

MLton Guide (20130715)

Contents

1	MLton	1
2	AdamGoode	2
3	AdmitsEquality	3
4	Alice	6
5	AllocateRegisters	7
6	AndreiFormiga	8
7	ArrayLiteral	9
8	AST	11
9	BasisLibrary	13
10	Bug	22
11	Bugs20041109	23
12	Bugs20051202	25
13	Bugs20070826	27
14	Bugs20100608	29
15	Bugs20130715	30
16	CallGraph	31
17	CallingFromCToSML	33
18	CallingFromSMLToC	36
19	CallingFromSMLToCFunctionPointer	38
20	Changelog	41
21	ChrisClearwater	83
22	Chunkify	84
23	CKitLibrary	85

24	Closure	86
25	ClosureConvert	87
26	CMinusMinus	88
27	CombineConversions	89
28	CommonArg	90
29	CommonBlock	95
30	CommonSubexp	97
31	CompilationManager	98
32	CompilerOverview	99
33	CompilerPassTemplate	100
34	CompileTimeOptions	101
35	CompilingWithSMLNJ	104
36	ConcurrentML	105
37	ConcurrentMLImplementation	106
38	ConstantPropagation	110
39	Contact	111
40	Contify	112
41	CoreML	113
42	CoreMLSimplify	114
43	Credits	115
44	CrossCompiling	117
45	CVS	118
46	DeadCode	119
47	DeepFlatten	120
48	DefineTypeBeforeUse	121

49	DefinitionOfStandardML	123
50	Defunctorize	124
51	Developers	126
52	Development	127
53	Documentation	128
54	Drawbacks	129
55	Eclipse	130
56	Elaborate	131
57	Emacs	135
58	EmacsBgBuildMode	136
59	EmacsDefUseMode	138
60	Enscript	140
61	EqualityType	141
62	EqualityTypeVariable	142
63	EtaExpansion	144
64	eXene	145
65	Experimental	146
66	FAQ	147
67	Features	148
68	FirstClassPolymorphism	151
69	Fixpoints	153
70	Flatten	156
71	Fold	157
72	Fold01N	169
73	ForeignFunctionInterface	171

74	ForeignFunctionInterfaceSyntax	172
75	ForeignFunctionInterfaceTypes	174
76	ForLoops	176
77	FrontEnd	180
78	FSharp	181
79	FunctionalRecordUpdate	182
80	fxp	185
81	GarbageCollection	186
82	GenerativeDatatype	187
83	GenerativeException	188
84	Git	190
85	Glade	191
86	Globalize	192
87	GnuMP	193
88	Google Summer of Code (2013)	194
89	HaMLet	198
90	HenryCejtin	199
91	History	200
92	HowProfilingWorks	201
93	Identifier	202
94	Immutable	203
95	ImperativeTypeVariable	204
96	ImplementExceptions	205
97	ImplementHandlers	206
98	ImplementProfiling	207

99	ImplementSuffix	208
100	InfixingOperators	209
101	Inline	212
102	InsertLimitChecks	213
103	InsertSignalChecks	214
104	Installation	215
105	IntermediateLanguage	216
106	IntroduceLoops	217
107	JesperLouisAndersen	218
108	JohnnyAndersen	219
109	KnownCase	220
110	LambdaCalculus	222
111	LambdaFree	223
112	LanguageChanges	224
113	Lazy	225
114	Libraries	226
115	LibrarySupport	228
116	License	231
117	LineDirective	232
118	LLVM	233
119	LocalFlatten	234
120	LocalRef	235
121	Logo	236
122	LoopInvariant	237
123	Machine	238

124	ManualPage	239
125	MatchCompilation	240
126	MatchCompile	241
127	MatthewFluet	243
128	mGTK	245
129	MichaelNorrish	246
130	MikeThomas	247
131	ML	248
132	MLAntlr	249
133	MLBasis	250
134	MLBasisAnnotationExamples	251
135	MLBasisAnnotations	252
136	MLBasisAvailableLibraries	253
137	MLBasisExamples	255
138	MLBasisPathMap	258
139	MLBasisSyntaxAndSemantics	259
140	MLj	260
141	MLKit	261
142	MLLex	262
143	MLLPTLibrary	263
144	MLmon	264
145	MLNLFFI	265
146	MLNLFFIGen	266
147	MLNLFFIImplementation	267
148	MLRISCLibrary	269

149	MLtonArray	271
150	MLtonBinIO	272
151	MLtonCont	273
152	MLtonContIsolateImplementation	274
153	MLtonCross	280
154	MLtonExn	281
155	MLtonFinalizable	282
156	MLtonGC	286
157	MLtonIntInf	287
158	MLtonIO	288
159	MLtonItimer	289
160	MLtonLibraryProject	290
161	MLtonMonoArray	291
162	MLtonMonoVector	292
163	MLtonPlatform	293
164	MLtonPointer	294
165	MLtonProcEnv	296
166	MLtonProcess	297
167	MLtonProfile	302
168	MLtonRandom	305
169	MLtonReal	306
170	MLtonRlimit	307
171	MLtonRusage	308
172	MLtonSignal	309
173	MLtonStructure	312

174	MLtonSyslog	316
175	MLtonTextIO	317
176	MLtonThread	318
177	MLtonVector	323
178	MLtonWeak	324
179	MLtonWord	325
180	MLtonWorld	326
181	MLULex	327
182	MLYacc	328
183	Monomorphise	329
184	MoscowML	331
185	Multi	332
186	Mutable	333
187	NeedsReview	334
188	NumericLiteral	335
189	ObjectOrientedProgramming	337
190	OCaml	342
191	OpenGL	343
192	OperatorPrecedence	344
193	OptionalArguments	345
194	OtherSites	348
195	Overloading	349
196	PackedRepresentation	350
197	ParallelMove	351
198	Performance	352

199 PhantomType	356
200 PlatformSpecificNotes	357
201 PolyEqual	358
202 PolyHash	359
203 PolyML	360
204 PolymorphicEquality	361
205 Polyvariance	364
206 Poplog	365
207 PortingMLton	366
208 PrecedenceParse	369
209 Printf	370
210 PrintfGentle	372
211 ProductType	377
212 Profiling	378
213 ProfilingAllocation	379
214 ProfilingCounts	380
215 ProfilingTheStack	382
216 ProfilingTime	383
217 Projects	385
218 Pronounce	386
219 PropertyList	387
220 Pygments	389
221 RayRacine	390
222 Reachability	391
223 Redundant	392

224	RedundantTests	393
225	References	394
226	RefFlatten	402
227	Regions	403
228	Release20041109	405
229	Release20051202	406
230	Release20070826	408
231	Release20100608	410
232	Release20130715	413
233	ReleaseChecklist	415
234	Releases	418
235	RemoveUnused	419
236	Restore	420
237	ReturnStatement	421
238	RSSA	424
239	RSSAShrink	425
240	RSSASimplify	426
241	RunningOnAIX	427
242	RunningOnAlpha	428
243	RunningOnAMD64	429
244	RunningOnARM	430
245	RunningOnCygwin	431
246	RunningOnDarwin	432
247	RunningOnFreeBSD	433
248	RunningOnHPPA	434

249	RunningOnHPUX	435
250	RunningOnIA64	436
251	RunningOnLinux	437
252	RunningOnMinGW	438
253	RunningOnNetBSD	440
254	RunningOnOpenBSD	441
255	RunningOnPowerPC	442
256	RunningOnPowerPC64	443
257	RunningOnS390	444
258	RunningOnSolaris	445
259	RunningOnSparc	446
260	RunningOnX86	447
261	RunTimeOptions	448
262	ScopeInference	450
263	SelfCompiling	451
264	Serialization	453
265	ShowBasis	454
266	ShowProf	456
267	Shrink	457
268	SimplifyTypes	458
269	SML3d	459
270	SMLNET	460
271	SMLNJ	461
272	SMLNJDeviations	462
273	SMLNJLibrary	467

274	SMLofNJStructure	469
275	SMLSharp	471
276	Sources	472
277	SpaceSafety	473
278	SSA	474
279	SSA2	475
280	SSA2Simplify	476
281	SSASimplify	477
282	Stabilizers	478
283	StandardML	480
284	StandardMLBooks	482
285	StandardMLGotchas	483
286	StandardMLHistory	486
287	StandardMLImplementations	487
288	StandardMLPortability	488
289	StandardMLTutorials	489
290	StaticSum	490
291	StephenWeeks	494
292	StyleGuide	495
293	Subversion	496
294	SuccessorML	497
295	SureshJagannathan	498
296	Swerve	499
297	SXML	500
298	SXMLShrink	501

299	SXMLSimplify	502
300	SyntacticConventions	503
301	Talk	510
302	TalkDiveIn	511
303	TalkFolkLore	512
304	TalkFromSMLTo	513
305	TalkHowHigherOrder	514
306	TalkHowModules	515
307	TalkHowPolymorphism	516
308	TalkMLtonApproach	517
309	TalkMLtonFeatures	518
310	TalkMLtonHistory	519
311	TalkStandardML	520
312	TalkTemplate	521
313	TalkWholeProgram	522
314	TILT	523
315	TipsForWritingConciseSML	524
316	ToMachine	527
317	TomMurphy	528
318	ToRSSA	529
319	ToSSA2	530
320	TypeChecking	531
321	TypeConstructor	534
322	TypeIndexedValues	535
323	TypeVariableScope	544

324 Unicode	546
325 UniversalType	547
326 UnresolvedBugs	549
327 UnsafeStructure	551
328 Useless	553
329 Users	554
330 Utilities	556
331 ValueRestriction	557
332 VariableArityPolymorphism	561
333 Variant	563
334 VesaKarvonen	564
335 WarnUnusedAnomalies	566
336 WesleyTerpstra	568
337 WholeProgramOptimization	569
338 WishList	570
339 XML	571
340 XMLShrink	573
341 XMLSimplify	575
342 XMLSimplifyTypes	576
343 Zone	577
344 ZZZOrphanedPages	578

Abstract

This is the guide for MLton, an open-source, whole-program, optimizing Standard ML compiler.

This guide was generated automatically from the MLton website, available online at <http://mlton.org>. It is up to date for MLton 20130715.

1 MLton

1.1 What is MLton?

MLton is an open-source, whole-program, optimizing [Standard ML](#) compiler.

1.2 What's new?

- 20130715: Please try out our latest release, [MLton 20130715](#).
- 20130308: Subversion repository on [SourceForge](#) converted and migrated to a Git repository on [GitHub](#); see [Sources](#) for current details.
- 20130227: [SourceForge](#) hosted resources have been upgraded to the [new SourceForge platform](#), resulting in a migration of the Subversion repository; see [Sources](#) for current details.
- 20120422: [www.mlton.org](#) content converted to be rendered as HTML using [AsciiDoc](#).
- 20120129: Subversion repository migrated to [SourceForge](#); see [Sources](#) for current details.
- 20120129: Mailing lists migrated to [SourceForge](#); see [Contact](#) for current details.

1.3 Next steps

- Read about MLton's [Features](#).
 - Look at [Documentation](#).
 - See some [Users](#) of MLton.
 - [Download](#) MLton.
 - Meet the MLton [Developers](#).
 - Get involved with MLton [Development](#).
 - User-maintained [FAQ](#).
 - [Contact](#) us.
-

2 AdamGoode

- I maintain the Fedora package of MLton, in [Fedora](#).
 - I have contributed some patches for Makefiles and PDF documentation building.
-

3 AdmitsEquality

A `TypeConstructor` admits equality if whenever it is applied to equality types, the result is an `EqualityType`. This notion enables one to determine whether a type constructor application yields an equality type solely from the application, without looking at the definition of the type constructor. It helps to ensure that `PolymorphicEquality` is only applied to sensible values.

The definition of admits equality depends on whether the type constructor was declared by a `type` definition or a `datatype` declaration.

3.1 Type definitions

For type definition

```
type ('a1, ..., 'an) t = ...
```

type constructor `t` admits equality if the right-hand side of the definition is an equality type after replacing `'a1, ..., 'an` by equality types (it doesn't matter which equality types are chosen).

For a nullary type definition, this amounts to the right-hand side being an equality type. For example, after the definition

```
type t = bool * int
```

type constructor `t` admits equality because `bool * int` is an equality type. On the other hand, after the definition

```
type t = bool * int * real
```

type constructor `t` does not admit equality, because `real` is not an equality type.

For another example, after the definition

```
type 'a t = bool * 'a
```

type constructor `t` admits equality because `bool * int` is an equality type (we could have chosen any equality type other than `int`).

On the other hand, after the definition

```
type 'a t = real * 'a
```

type constructor `t` does not admit equality because `real * int` is not equality type.

We can check that a type constructor admits equality using an `eqtype` specification.

```
structure Ok: sig eqtype 'a t end =
  struct
    type 'a t = bool * 'a
  end
```

```
structure Bad: sig eqtype 'a t end =
  struct
    type 'a t = real * int * 'a
  end
```

On structure `Bad`, MLton reports the following error.

```
Type t admits equality in signature but not in structure.
  not equality: [real] * _ * _
```

The `not equality` section provides an explanation of why the type did not admit equality, highlighting the problematic component (`real`).

3.2 Datatype declarations

For a type constructor declared by a datatype declaration to admit equality, every **variant** of the datatype must admit equality. For example, the following datatype admits equality because `bool` and `char * int` are equality types.

```
datatype t = A of bool | B of char * int
```

Nullary constructors trivially admit equality, so that the following datatype admits equality.

```
datatype t = A | B | C
```

For a parameterized datatype constructor to admit equality, we consider each **variant** as a type definition, and require that the definition admit equality. For example, for the datatype

```
datatype 'a t = A of bool * 'a | B of 'a
```

the type definitions

```
type 'a tA = bool * 'a
type 'a tB = 'a
```

both admit equality. Thus, type constructor `t` admits equality.

On the other hand, the following datatype does not admit equality.

```
datatype 'a t = A of bool * 'a | B of real * 'a
```

As with type definitions, we can check using an `eqtype` specification.

```
structure Bad: sig eqtype 'a t end =
  struct
    datatype 'a t = A of bool * 'a | B of real * 'a
  end
```

MLton reports the following error.

```
Type t admits equality in signature but not in structure.
  not equality: B of [real] * _
```

MLton indicates the problematic constructor (B), as well as the problematic component of the constructor's argument.

3.2.1 Recursive datatypes

A recursive datatype like

```
datatype t = A | B of int * t
```

introduces a new problem, since in order to decide whether `t` admits equality, we need to know for the `B` **variant** whether `t` admits equality. The **Definition** answers this question by requiring a type constructor to admit equality if it is consistent to do so. So, in our above example, if we assume that `t` admits equality, then the **variant** `B of int * t` admits equality. Then, since the `A` **variant** trivially admits equality, so does the type constructor `t`. Thus, it was consistent to assume that `t` admits equality, and so, `t` does admit equality.

On the other hand, in the following declaration

```
datatype t = A | B of real * t
```

if we assume that `t` admits equality, then the `B` **variant** does not admit equality. Hence, the type constructor `t` does not admit equality, and our assumption was inconsistent. Hence, `t` does not admit equality.

The same kind of reasoning applies to mutually recursive datatypes as well. For example, the following defines both `t` and `u` to admit equality.

```
datatype t = A | B of u
and u = C | D of t
```

But the following defines neither `t` nor `u` to admit equality.

```
datatype t = A | B of u * real
and u = C | D of t
```

As always, we can check whether a type admits equality using an `eqtype` specification.

```
structure Bad: sig eqtype t eqtype u end =
  struct
    datatype t = A | B of u * real
    and u = C | D of t
  end
```

MLton reports the following error.

```
Error: z.sml 1.16.
  Type t admits equality in signature but not in structure.
    not equality: B of [u] * [real]
Error: z.sml 1.16.
  Type u admits equality in signature but not in structure.
    not equality: D of [t]
```

4 Alice

Alice ML is an extension of SML with concurrency, dynamic typing, components, distribution, and constraint solving.

5 AllocateRegisters

[AllocateRegisters](#) is an analysis pass for the [RSSA IntermediateLanguage](#), invoked from [ToMachine](#).

5.1 Description

Computes an allocation of [RSSA](#) variables as [Machine](#) register or stack operands.

5.2 Implementation

- [allocate-registers.sig](#)
- [allocate-registers.fun](#)

5.3 Details and Notes

6 AndreiFormiga

I'm a graduate student just back in academia. I study concurrent and parallel systems, with a great deal of interest in programming languages (theory, design, implementation). I happen to like functional languages.

I use the nickname tautologico on #sml and my email is andrei DOT formiga AT gmail DOT com.

7 ArrayLiteral

Standard ML does not have a syntax for array literals or vector literals. The only way to write down an array is like

```
Array.fromList [w, x, y, z]
```

No SML compiler produces efficient code for the above expression. The generated code allocates a list and then converts it to an array. To alleviate this, one could write down the same array using `Array.tabulate`, or even using `Array.array` and `Array.update`, but that is syntactically unwieldy.

Fortunately, using [Fold](#), it is possible to define constants `A`, and ``` so that one can write down an array like:

```
A `w `x `y `z $
```

This is as syntactically concise as the `fromList` expression. Furthermore, MLton, at least, will generate the efficient code as if one had written down a use of `Array.array` followed by four uses of `Array.update`.

Along with `A` and ```, one can define a constant `V` that makes it possible to define vector literals with the same syntax, e.g.,

```
V `w `x `y `z $
```

Note that the same element indicator, ```, serves for both array and vector literals. Of course, the `$` is the end-of-arguments marker always used with [Fold](#). The only difference between an array literal and vector literal is the `A` or `V` at the beginning.

Here is the implementation of `A`, `V`, and ```. We place them in a structure and use signature abstraction to hide the type of the accumulator. See [Fold](#) for more on this technique.

```
structure Literal:>
  sig
    type 'a z
    val A: ('a z, 'a z, 'a array, 'd) Fold.t
    val V: ('a z, 'a z, 'a vector, 'd) Fold.t
    val `: ('a, 'a z, 'a z, 'b, 'c, 'd) Fold.step1
  end =
  struct
    type 'a z = int * 'a option * ('a array -> unit)

    val A =
      fn z =>
        Fold.fold
          ((0, NONE, ignore),
           fn (n, opt, fill) =>
             case opt of
               NONE =>
                 Array.tabulate (0, fn _ => raise Fail "array0")
             | SOME x =>
                 let
                   val a = Array.array (n, x)
                   val () = fill a
                 in
                   a
                 end)
          z

    val V = fn z => Fold.post (A, Array.vector) z

    val ` =
      fn z =>
        Fold.step1
          (fn (x, (i, opt, fill)) =>
             (i + 1,
              SOME x,
```

```
fn a => (Array.update (a, i, x); fill a))  
  z  
end
```

The idea of the code is for the fold to accumulate a count of the number of elements, a sample element, and a function that fills in all the elements. When the fold is complete, the finishing function allocates the array, applies the fill function, and returns the array. The only difference between \bar{A} and \bar{V} is at the very end; \bar{A} just returns the array, while \bar{V} converts it to a vector using post-composition, which is further described on the [Fold](#) page.

8 AST

AST is the [IntermediateLanguage](#) produced by the [FrontEnd](#) and translated by [Elaborate](#) to [CoreML](#).

8.1 Description

The abstract syntax tree produced by the [FrontEnd](#).

8.2 Implementation

- [ast-programs.sig](#)
- [ast-programs.fun](#)
- [ast-modules.sig](#)
- [ast-modules.fun](#)
- [ast-core.sig](#)
- [ast-core.fun](#)
- [ast](#)

8.3 Type Checking

The [AST IntermediateLanguage](#) has no independent type checker. Type inference is performed on an AST program as part of [Elaborate](#).

8.4 Details and Notes

8.4.1 Source locations

MLton makes use of a relatively clean method for annotating the abstract syntax tree with source location information. Every source program phrase is "wrapped" with the `WRAPPED` interface:

```
signature WRAPPED =
  sig
    type node'
    type obj

    val dest: obj -> node' * Region.t
    val makeRegion': node' * SourcePos.t * SourcePos.t -> obj
    val makeRegion: node' * Region.t -> obj
    val node: obj -> node'
    val region: obj -> Region.t
  end
```

The key idea is that `node'` is the type of an unannotated syntax phrase and `obj` is the type of its annotated counterpart. In the implementation, every `node'` is annotated with a `Region.t` ([region.sig](#), [region.sml](#)), which describes the syntax phrase's left source position and right source position, where `SourcePos.t` ([source-pos.sig](#), [source-pos.sml](#)) denotes a particular file, line, and column. A typical use of the `WRAPPED` interface is illustrated by the following code:

```

datatype node =
  App of Longcon.t * t
| Const of Const.t
| Constraint of t * Type.t
| FlatApp of t vector
| Layered of {constraint: Type.t option,
              fixop: Fixop.t,
              pat: t,
              var: Var.t}
| List of t vector
| Record of {flexible: bool,
             items: (Record.Field.t * Item.t) vector}
| Tuple of t vector
| Var of {fixop: Fixop.t,
          name: Longvid.t}
| Wild

include WRAPPED sharing type node' = node
                sharing type obj = t

```

Thus, AST nodes are cleanly separated from source locations. By way of contrast, consider the approach taken by [SML/NJ](#) (and also by the [CKit Library](#)). Each datatype denoting a syntax phrase dedicates a special constructor for annotating source locations:

```

datatype pat = WildPat (* empty pattern *)
| AppPat of {constr:pat, argument:pat} (* application *)
| MarkPat of pat * region (* mark a pattern *)

```

The main drawback of this approach is that static type checking is not sufficient to guarantee that the AST emitted from the front-end is properly annotated.

9 BasisLibrary

The [Standard ML Basis Library](#) is a collection of modules dealing with basic types, input/output, OS interfaces, and simple datatypes. It is intended as a portable library usable across all implementations of SML. For the official online version of the Basis Library specification, see <http://www.standardml.org/Basis>. [The Standard ML Basis Library](#) is a book version that includes all of the online version and more. For a reverse chronological list of changes to the specification, see <http://www.standardml.org/Basis/history.html>.

MLton implements all of the required portions of the Basis Library. MLton also implements many of the optional structures. You can obtain a complete and current list of what's available using `mlton -show-basis` (see [ShowBasis](#)). By default, MLton makes the Basis Library available to user programs. You can also [access the Basis Library](#) from [ML Basis](#) files.

Below is a complete list of what MLton implements.

9.1 Top-level types and constructors

```
eqtype 'a array
datatype bool = false | true
eqtype char
type exn
eqtype int
datatype 'a list = nil | :: of ('a * 'a list)
datatype 'a option = NONE | SOME of 'a
datatype order = EQUAL | GREATER | LESS
type real
datatype 'a ref = ref of 'a
eqtype string
type substring
eqtype unit
eqtype 'a vector
eqtype word
```

9.2 Top-level exception constructors

```
Bind
Chr
Div
Domain
Empty
Fail of string
Match
Option
Overflow
Size
Span
Subscript
```

9.3 Top-level values

MLton does not implement the optional top-level value `use: string -> unit`, which conflicts with whole-program compilation because it allows new code to be loaded dynamically.

MLton implements all other top-level values:

!, :=, <>, =, @, ^, app, before, ceil, chr, concat, exnMessage, exnName, explode, floor, foldl, foldr, getOpt, hd, ignore, implode, isSome, length, map, not, null, o, ord, print, real, rev, round, size, str, substring, tl, trunc, valOf, vector

9.4 Overloaded identifiers

*, +, -, /, <, <=, >, >=, ~, abs, div, mod

9.5 Top-level signatures

ARRAY

ARRAY2

ARRAY_SLICE

BIN_IO

BIT_FLAGS

BOOL

BYTE

CHAR

COMMAND_LINE

DATE

GENERAL

GENERIC SOCK

IEEE_REAL

IMPERATIVE_IO

INET SOCK

INTEGER

INT_INF

IO

LIST

LIST_PAIR

MATH

MONO_ARRAY

MONO_ARRAY2

MONO_ARRAY_SLICE

MONO_VECTOR

MONO_VECTOR_SLICE

NET_HOST_DB

NET_PROT_DB
NET_SERV_DB
OPTION
OS
OS_FILE_SYS
OS_IO
OS_PATH
OS_PROCESS
PACK_REAL
PACK_WORD
POSIX
POSIX_ERROR
POSIX_FILE_SYS
POSIX_IO
POSIX_PROCESS
POSIX_PROC_ENV
POSIX_SIGNAL
POSIX_SYS_DB
POSIX_TTY
PRIM_IO
REAL
SOCKET
STREAM_IO
STRING
STRING_CVT
SUBSTRING
TEXT
TEXT_IO
TEXT_STREAM_IO
TIME
TIMER
UNIX
UNIX SOCK
VECTOR
VECTOR_SLICE
WORD

9.6 Top-level structures

```
structure Array: ARRAY
structure Array2: ARRAY2
structure ArraySlice: ARRAY_SLICE
structure BinIO: BIN_IO
structure BinPrimIO: PRIM_IO
structure Bool: BOOL
structure BoolArray: MONO_ARRAY
structure BoolArray2: MONO_ARRAY2
structure BoolArraySlice: MONO_ARRAY_SLICE
structure BoolVector: MONO_VECTOR
structure BoolVectorSlice: MONO_VECTOR_SLICE
structure Byte: BYTE
structure Char: CHAR
```

- Char characters correspond to ISO-8859-1. The Char functions do not depend on locale.

```
structure CharArray: MONO_ARRAY
structure CharArray2: MONO_ARRAY2
structure CharArraySlice: MONO_ARRAY_SLICE
structure CharVector: MONO_VECTOR
structure CharVectorSlice: MONO_VECTOR_SLICE
structure CommandLine: COMMAND_LINE
structure Date: DATE
```

- Date.fromString and Date.scan accept a space in addition to a zero for the first character of the day of the month. The Basis Library specification only allows a zero.

```
structure FixedInt: INTEGER
structure General: GENERAL
structure GenericSock: GENERIC SOCK
structure IEEEReal: IEEE_REAL
structure INetSock: INET SOCK
structure IO: IO
structure Int: INTEGER
structure Int1: INTEGER
structure Int2: INTEGER
structure Int3: INTEGER
structure Int4: INTEGER
...
structure Int31: INTEGER
```

```
structure Int32: INTEGER
structure Int64: INTEGER
structure IntArray: MONO_ARRAY
structure IntArray2: MONO_ARRAY2
structure IntArraySlice: MONO_ARRAY_SLICE
structure IntVector: MONO_VECTOR
structure IntVectorSlice: MONO_VECTOR_SLICE
structure Int8: INTEGER
structure Int8Array: MONO_ARRAY
structure Int8Array2: MONO_ARRAY2
structure Int8ArraySlice: MONO_ARRAY_SLICE
structure Int8Vector: MONO_VECTOR
structure Int8VectorSlice: MONO_VECTOR_SLICE
structure Int16: INTEGER
structure Int16Array: MONO_ARRAY
structure Int16Array2: MONO_ARRAY2
structure Int16ArraySlice: MONO_ARRAY_SLICE
structure Int16Vector: MONO_VECTOR
structure Int16VectorSlice: MONO_VECTOR_SLICE
structure Int32: INTEGER
structure Int32Array: MONO_ARRAY
structure Int32Array2: MONO_ARRAY2
structure Int32ArraySlice: MONO_ARRAY_SLICE
structure Int32Vector: MONO_VECTOR
structure Int32VectorSlice: MONO_VECTOR_SLICE
structure Int64Array: MONO_ARRAY
structure Int64Array2: MONO_ARRAY2
structure Int64ArraySlice: MONO_ARRAY_SLICE
structure Int64Vector: MONO_VECTOR
structure Int64VectorSlice: MONO_VECTOR_SLICE
structure IntInf: INT_INF
structure LargeInt: INTEGER
structure LargeIntArray: MONO_ARRAY
structure LargeIntArray2: MONO_ARRAY2
structure LargeIntArraySlice: MONO_ARRAY_SLICE
structure LargeIntVector: MONO_VECTOR
structure LargeIntVectorSlice: MONO_VECTOR_SLICE
structure LargeReal: REAL
structure LargeRealArray: MONO_ARRAY
```

```
structure LargeRealArray2: MONO_ARRAY2
structure LargeRealArraySlice: MONO_ARRAY_SLICE
structure LargeRealVector: MONO_VECTOR
structure LargeRealVectorSlice: MONO_VECTOR_SLICE
structure LargeWord: WORD
structure LargeWordArray: MONO_ARRAY
structure LargeWordArray2: MONO_ARRAY2
structure LargeWordArraySlice: MONO_ARRAY_SLICE
structure LargeWordVector: MONO_VECTOR
structure LargeWordVectorSlice: MONO_VECTOR_SLICE
structure List: LIST
structure ListPair: LIST_PAIR
structure Math: MATH
structure NetHostDB: NET_HOST_DB
structure NetProtDB: NET_PROT_DB
structure NetServDB: NET_SERV_DB
structure OS: OS
structure Option: OPTION
structure PackReal32Big: PACK_REAL
structure PackReal32Little: PACK_REAL
structure PackReal64Big: PACK_REAL
structure PackReal64Little: PACK_REAL
structure PackRealBig: PACK_REAL
structure PackRealLittle: PACK_REAL
structure PackWord16Big: PACK_WORD
structure PackWord16Little: PACK_WORD
structure PackWord32Big: PACK_WORD
structure PackWord32Little: PACK_WORD
structure PackWord64Big: PACK_WORD
structure PackWord64Little: PACK_WORD
structure Position: INTEGER
structure Posix: POSIX
structure Real: REAL
structure RealArray: MONO_ARRAY
structure RealArray2: MONO_ARRAY2
structure RealArraySlice: MONO_ARRAY_SLICE
structure RealVector: MONO_VECTOR
structure RealVectorSlice: MONO_VECTOR_SLICE
structure Real32: REAL
```

```

structure Real32Array: MONO_ARRAY
structure Real32Array2: MONO_ARRAY2
structure Real32ArraySlice: MONO_ARRAY_SLICE
structure Real32Vector: MONO_VECTOR
structure Real32VectorSlice: MONO_VECTOR_SLICE
structure Real64: REAL
structure Real64Array: MONO_ARRAY
structure Real64Array2: MONO_ARRAY2
structure Real64ArraySlice: MONO_ARRAY_SLICE
structure Real64Vector: MONO_VECTOR
structure Real64VectorSlice: MONO_VECTOR_SLICE
structure Socket: SOCKET

```

- The Basis Library specification requires functions like `Socket.sendVec` to raise an exception if they fail. However, on some platforms, sending to a socket that hasn't yet been connected causes a `SIGPIPE` signal, which invokes the default signal handler for `SIGPIPE` and causes the program to terminate. If you want the exception to be raised, you can ignore `SIGPIPE` by adding the following to your program.

```

let
  open MLton.Signal
in
  setHandler (Posix.Signal.pipe, Handler.ignore)
end

```

```

structure String: STRING

```

- The `String` functions do not depend on locale.

```

structure StringCvt: STRING_CVT
structure Substring: SUBSTRING
structure SysWord: WORD
structure Text: TEXT
structure TextIO: TEXT_IO
structure TextPrimIO: PRIM_IO
structure Time: TIME
structure Timer: TIMER
structure Unix: UNIX
structure UnixSock: UNIX SOCK
structure Vector: VECTOR
structure VectorSlice: VECTOR_SLICE
structure Word: WORD
structure Word1: WORD
structure Word2: WORD
structure Word3: WORD

```

```
structure Word4: WORD
...
structure Word31: WORD
structure Word32: WORD
structure Word64: WORD
structure WordArray: MONO_ARRAY
structure WordArray2: MONO_ARRAY2
structure WordArraySlice: MONO_ARRAY_SLICE
structure WordVectorSlice: MONO_VECTOR_SLICE
structure WordVector: MONO_VECTOR
structure Word8Array: MONO_ARRAY
structure Word8Array2: MONO_ARRAY2
structure Word8ArraySlice: MONO_ARRAY_SLICE
structure Word8Vector: MONO_VECTOR
structure Word8VectorSlice: MONO_VECTOR_SLICE
structure Word16Array: MONO_ARRAY
structure Word16Array2: MONO_ARRAY2
structure Word16ArraySlice: MONO_ARRAY_SLICE
structure Word16Vector: MONO_VECTOR
structure Word16VectorSlice: MONO_VECTOR_SLICE
structure Word32Array: MONO_ARRAY
structure Word32Array2: MONO_ARRAY2
structure Word32ArraySlice: MONO_ARRAY_SLICE
structure Word32Vector: MONO_VECTOR
structure Word32VectorSlice: MONO_VECTOR_SLICE
structure Word64Array: MONO_ARRAY
structure Word64Array2: MONO_ARRAY2
structure Word64ArraySlice: MONO_ARRAY_SLICE
structure Word64Vector: MONO_VECTOR
structure Word64VectorSlice: MONO_VECTOR_SLICE
```

9.7 Top-level functors

ImperativeIO
PrimIO
StreamIO

- MLton's `StreamIO` functor takes structures `ArraySlice` and `VectorSlice` in addition to the arguments specified in the Basis Library specification.

9.8 Type equivalences

The following types are equivalent.

```
FixedInt = Int64.int
LargeInt = IntInf.int
LargeReal.real = Real64.real
LargeWord = Word64.word
```

The default `int`, `real`, and `word` types may be set by the `-default-type type` [compile-time option](#). By default, the following types are equivalent:

```
int = Int.int = Int32.int
real = Real.real = Real64.real
word = Word.word = Word32.word
```

9.9 Real and Math functions

The `Real`, `Real32`, and `Real64` modules are implemented using the C math library, so the SML functions will reflect the behavior of the underlying library function. We have made some effort to unify the differences between the math libraries on different platforms, and in particular to handle exceptional cases according to the Basis Library specification. However, there will be differences due to different numerical algorithms and cases we may have missed. Please submit a [bug report](#) if you encounter an error in the handling of an exceptional case.

On x86, real arithmetic is implemented internally using 80 bits of precision. Using higher precision for intermediate results in computations can lead to different results than if all the computation is done at 32 or 64 bits. If you require strict IEEE compliance, you can compile with `-ieee-fp true`, which will cause intermediate results to be stored after each operation. This may cause a substantial performance penalty.

10 Bug

To report a bug, please send mail to mlton-devel@mlton.org. Please include the complete SML program that caused the problem and a log of a compile of the program with `-verbose 2`. For large programs (over 256K), please send an email containing the discussion text and a link to any large files.

There are some [UnresolvedBugs](#) that we don't plan to fix.

We also maintain a list of bugs found with each release.

- [Bugs20100608](#)
 - [Bugs20070826](#)
 - [Bugs20051202](#)
 - [Bugs20041109](#)
-

11 Bugs20041109

Here are the known bugs in [MLton 20041109](#), listed in reverse chronological order of date reported.

- `MLton.Finalizable.touch` doesn't necessarily keep values alive long enough. Our SVN has a patch to the compiler. You must rebuild the compiler in order for the patch to take effect.

Thanks to Florian Weimer for reporting this bug.

- A bug in an optimization pass may incorrectly transform a program to flatten ref cells into their containing data structure, yielding a type-error in the transformed program. Our CVS has a [patch](#) to the compiler. You must rebuild the compiler in order for the patch to take effect.

Thanks to [VesaKarvonen](#) for reporting this bug.

- A bug in the front end mistakenly allows unary constructors to be used without an argument in patterns. For example, the following program is accepted, and triggers a large internal error.

```
fun f x = case x of SOME => true | _ => false
```

We have fixed the problem in our CVS.

Thanks to William Lovas for reporting this bug.

- A bug in `Posix.IO.{getlk,setlk,setlkw}` causes a link-time error: `undefined reference to Posix_IO_FLock_typ` Our CVS has a [patch](#) to the Basis Library implementation.

Thanks to Adam Chlipala for reporting this bug.

- A bug can cause programs compiled with `-profile alloc` to segfault. Our CVS has a [patch](#) to the compiler. You must rebuild the compiler in order for the patch to take effect.

Thanks to John Reppy for reporting this bug.

- A bug in an optimization pass may incorrectly flatten ref cells into their containing data structure, breaking the sharing between the cells. Our CVS has a [patch](#) to the compiler. You must rebuild the compiler in order for the patch to take effect.

Thanks to Paul Govereau for reporting this bug.

- Some arrays or vectors, such as `(char * char) vector`, are incorrectly implemented, and will conflate the first and second components of each element. Our CVS has a [patch](#) to the compiler. You must rebuild the compiler in order for the patch to take effect.

Thanks to Scott Cruzen for reporting this bug.

- `Socket.Ctl.getLINGER` and `Socket.Ctl.setLINGER` mistakenly raise `Subscript`. Our CVS has a [patch](#) to the Basis Library implementation.

Thanks to Ray Racine for reporting the bug.

- `CML Mailbox.send` makes a call in the wrong atomic context. Our CVS has a [patch](#) to the CML implementation.

- `OS.Path.joinDirFile` and `OS.Path.toString` did not raise `InvalidArc` when they were supposed to. They now do. Our CVS has a [patch](#) to the Basis Library implementation.

Thanks to Andreas Rossberg for reporting the bug.

- The front end incorrectly disallows sequences of expressions (separated by semicolons) after a topdec has already been processed. For example, the following is incorrectly rejected.

```
val x = 0;  
ignore x;  
ignore x;
```

We have fixed the problem in our CVS.

Thanks to Andreas Rossberg for reporting the bug.

- The front end incorrectly disallows expansive `val` declarations that bind a type variable that doesn't occur in the type of the value being bound. For example, the following is incorrectly rejected.

```
val 'a x = let exception E of 'a in () end
```

We have fixed the problem in our CVS.

Thanks to Andreas Rossberg for reporting this bug.

- The x86 codegen fails to account for the possibility that a 64-bit move could interfere with itself (as simulated by 32-bit moves). We have fixed the problem in our CVS.

Thanks to Scott Cruzen for reporting this bug.

- `NetHostDB.scan` and `NetHostDB.fromString` incorrectly raise an exception on internet addresses whose last component is a zero, e.g `0.0.0.0`. Our CVS has a [patch](#) to the Basis Library implementation.

Thanks to Scott Cruzen for reporting this bug.

- `StreamIO.inputLine` has an off-by-one error causing it to drop the first character after a newline in some situations. Our CVS has a [patch](#). to the Basis Library implementation.

Thanks to Scott Cruzen for reporting this bug.

- `BinIO.getInstream` and `TextIO.getInstream` are implemented incorrectly. This also impacts the behavior of `BinIO.scanStream` and `TextIO.scanStream`. If you (directly or indirectly) realize a `TextIO.StreamIO.instream` and do not (directly or indirectly) call `TextIO.setInstream` with a derived stream, you may lose input data. We have fixed the problem in our CVS.

Thanks to [WesleyTerpstra](#) for reporting this bug.

- `Posix.ProcEnv.setpgid` doesn't work. If you compile a program that uses it, you will get a link time error

```
undefined reference to `Posix_ProcEnv_setpgid'
```

The bug is due to `Posix_ProcEnv_setpgid` being omitted from the MLton runtime. We fixed the problem in our CVS by adding the following definition to `runtime/Posix/ProcEnv/ProcEnv.c`

```
Int Posix_ProcEnv_setpgid (Pid p, Gid g) {  
    return setpgid (p, g);  
}
```

Thanks to Tom Murphy for reporting this bug.

12 Bugs20051202

Here are the known bugs in [MLton 20051202](#), listed in reverse chronological order of date reported.

- Bug in the `Real<N>.fmt`, `Real<N>.fromString`, `Real<N>.scan`, and `Real<N>.toString` functions of the [Basis Library](#) implementation. These functions were using `TO_NEAREST` semantics, but should obey the current rounding mode. (Only `Real<N>.fmt StringCvt.EXACT`, `Real<N>.fromDecimal`, and `Real<N>.toDecimal` are specified to override the current rounding mode with `TO_NEAREST` semantics.)

Thanks to Sean McLaughlin for the bug report.

Fixed by revision [r5827](#).

- Bug in the treatment of floating-point operations. Floating-point operations depend on the current rounding mode, but were being treated as pure.

Thanks to Sean McLaughlin for the bug report.

Fixed by revision [r5794](#).

- Bug in the `Real32.toInt` function of the [Basis Library](#) implementation could lead incorrect results when applied to a `Real32.real` value numerically close to `valOf(Int.maxInt)`.

Fixed by revision [r5764](#).

- The `Socket` structure of the [Basis Library](#) implementation used `andb` rather than `orb` to unmarshal socket options (for `Socket.Ctl.get<OPT>` functions).

Thanks to Anders Petersson for the bug report and patch.

Fixed by revision [r5735](#).

- Bug in the `Date` structure of the [Basis Library](#) implementation yielded some functions that would erroneously raise `Date` when applied to a year before 1900.

Thanks to Joe Hurd for the bug report.

Fixed by revision [r5732](#).

- Bug in monomorphisation pass could exhibit the error `Type error: type mismatch`.

Thanks to Vesa Karvonen for the bug report.

Fixed by revision [r5731](#).

- The `PackReal<N>.toBytes` function in the [Basis Library](#) implementation incorrectly shared (and mutated) the result vector.

Thanks to Eric McCorkle for the bug report and patch.

Fixed by revision [r5281](#).

- Bug in elaboration of FFI forms. Using a unary FFI types (e.g., `array`, `ref`, `vector`) in places where `MLton.Pointer.t` was required would lead to an internal error `TypeError`.

Fixed by revision [r4890](#).

- The `MONO_VECTOR` signature of the [Basis Library](#) implementation incorrectly omits the specification of `find`.

Fixed by revision [r4707](#).

- The optimizer reports an internal error (`TypeError`) when an imported C function is called but not used.

Thanks to "jq" for the bug report.

Fixed by revision [r4690](#).

- Bug in pass to flatten data structures.

Thanks to Joe Hurd for the bug report.

Fixed by revision [r4662](#).

- The native codegen's implementation of the C-calling convention failed to widen 16-bit arguments to 32-bits.
Fixed by revision [r4631](#).
 - The `PACK_REAL` structures of the [Basis Library](#) implementation used byte, rather than element, indexing.
Fixed by revision [r4411](#).
 - `MLton.share` could cause a segmentation fault.
Fixed by revision [r4400](#).
 - The SSA simplifier could eliminate an irredundant test.
Fixed by revision [r4370](#).
 - A program with a very large number of functors could exhibit the error `ElaborateEnv.functorClosure: firstTycons`.
Fixed by revision [r4344](#).
-

13 Bugs20070826

Here are the known bugs in [MLton 20070826](#), listed in reverse chronological order of date reported.

- Bug in the mark-compact garbage collector where the C library's `memcpy` was used to move objects during the compaction phase; this could lead to heap corruption and segmentation faults with newer versions of gcc and/or glibc, which assume that `src` and `dst` in a `memcpy` do not overlap.
Fixed by revision [r7461](#).
 - Bug in elaboration of `datatype` declarations with `withtype` bindings.
Fixed by revision [r7434](#).
 - Performance bug in [RefFlatten](#) optimization pass.
Thanks to Reactive Systems for the bug report.
Fixed by revision [r7379](#).
 - Performance bug in [SimplifyTypes](#) optimization pass.
Thanks to Reactive Systems for the bug report.
Fixed by revisions [r7377](#) and [r7378](#).
 - Bug in amd64 codegen register allocation of indirect C calls.
Thanks to David Hansel for the bug report.
Fixed by revision [r7368](#).
 - Bug in `IntInf.scan` and `IntInf.fromString` where leading spaces were only accepted if the stream had an explicit sign character.
Thanks to David Hansel for the bug report.
Fixed by revisions [r7227](#) and [r7230](#).
 - Bug in `IntInf.~>>` that could cause a glibc assertion.
Fixed by revisions [r7083](#), [r7084](#), and [r7085](#).
 - Bug in the return type of `MLton.Process.reap`.
Thanks to Risto Saarelma for the bug report.
Fixed by revision [r7029](#).
 - Bug in `MLton.size` and `MLton.share` when tracing the current stack.
Fixed by revisions [r6978](#), [r6981](#), [r6988](#), [r6989](#), and [r6990](#).
 - Bug in nested `_export/_import` functions.
Fixed by revision [r6919](#).
 - Bug in the name mangling of `_import-ed` functions with the `stdcall` convention.
Thanks to Lars Bergstrom for the bug report.
Fixed by revision [r6672](#).
 - Bug in Windows code to page the heap to disk when unable to grow the heap to a desired size.
Thanks to Sami Evangelista for the bug report.
Fixed by revisions [r6600](#) and [r6624](#).
 - Bug in `*NIX` code to page the heap to disk when unable to grow the heap to a desired size.
Thanks to Nicolas Bertolotti for the bug report and patch.
Fixed by revisions [r6596](#) and [r6600](#).
-

-
- Space-safety bug in pass to [flatten refs](#) into containing data structure.
Thanks to Daniel Spoonhower for the bug report and initial diagnosis and patch.
Fixed by revision [r6395](#).
 - Bug in the frontend that rejected `op longvid` patterns and expressions.
Thanks to Florian Weimer for the bug report.
Fixed by revision [r6347](#).
 - Bug in the `IMPERATIVE_IO.canInput` function of the [Basis Library](#) implementation.
Thanks to Ville Laurikari for the bug report.
Fixed by revision [r6261](#).
 - Bug in algebraic simplification of real primitives. `REAL<N>. <= (x, x)` is `false` when `x` is NaN.
Fixed by revision [r6242](#).
 - Bug in the FFI visible representation of `Int16.int ref` (and references of other primitive types smaller than 32-bits) on big-endian platforms.
Thanks to Dave Herman for the bug report.
Fixed by revision [r6267](#).
 - Bug in type inference of flexible records. This would later cause the compiler to raise the `TypeError` exception.
Thanks to Wesley Terpstra for the bug report.
Fixed by revision [r6229](#).
 - Bug in cross-compilation of `gdt oa` library.
Thanks to Wesley Terpstra for the bug report and patch.
Fixed by revision [r6620](#).
 - Bug in pass to [flatten refs](#) into containing data structure.
Thanks to Ruy Ley-Wild for the bug report.
Fixed by revision [r6191](#).
 - Bug in the handling of weak pointers by the mark-compact garbage collector.
Thanks to Sean McLaughlin for the bug report and Florian Weimer for the initial diagnosis.
Fixed by revision [r6183](#).
 - Bug in the elaboration of structures with signature constraints. This would later cause the compiler to raise the `TypeError` exception.
Thanks to Vesa Karvonen for the bug report.
Fixed by revision [r6046](#).
 - Bug in the interaction of `_export-ed` functions and signal handlers.
Thanks to Sean McLaughlin for the bug report.
Fixed by revision [r6013](#).
 - Bug in the implementation of `_export-ed` functions using the `char` type, leading to a linker error.
Thanks to Katsuhiko Ueno for the bug report.
Fixed by revision [r5999](#).
-

14 Bugs20100608

Here are the known bugs in [MLton 20100608](#), listed in reverse chronological order of date reported.

- Bug in `REAL.signBit`, `REAL.copySign`, and `REAL.toDecimal/REAL.fromDecimal`.
Thanks to Phil Clayton for the bug report and examples.
Fixed by revisions [r7571](#), [r7572](#), and [r7573](#).
 - Bug in elaboration of type variables with and without equality status.
Thanks to Rob Simmons for the bug report and examples.
Fixed by revision [r7565](#).
 - Bug in [redundant SSA](#) optimization.
Thanks to Lars Magnusson for the bug report and example.
Fixed by revision [r7561](#).
 - Bug in [SSA/SSA2 shrinker](#) that could erroneously turn a non-tail function call with a `Bug` transfer as its continuation into a tail function call.
Thanks to Lars Bergstrom for the bug report.
Fixed by revision [r7546](#).
 - Bug in translation from [SSA2](#) to [RSSA](#) with `case` expressions over non-primitive-sized words.
Fixed by revision [r7544](#).
 - Bug with [SSA/SSA2](#) type checking of `case` expressions over words.
Fixed by revision [r7542](#).
 - Bug with treatment of `as`-patterns, which should not allow the redefinition of constructor status.
Thanks to Michael Norrish for the bug report.
Fixed by revision [r7530](#).
 - Bug with treatment of `nan` in [common subexpression elimination SSA](#) optimization.
Thanks to Alexandre Hamez for the bug report.
Fixed by revision [r7503](#).
 - Bug in translation from [SSA2](#) to [RSSA](#) with weak pointers.
Thanks to Alexandre Hamez for the bug report.
Fixed by revision [r7502](#).
 - Bug in amd64 codegen calling convention for `varargs C` calls.
Thanks to [HenryCejtin](#) for the bug report and [WesleyTerpstra](#) for the initial diagnosis.
Fixed by revision [r7501](#).
 - Bug in comment-handling in lexer for [MLYacc](#)'s input language.
Thanks to Michael Norrish for the bug report and patch.
Fixed by revision [r7500](#).
 - Bug in elaboration of function clauses with different numbers of arguments that would raise an uncaught `Subscript exception`.
Fixed by revision [r75497](#).
-

15 Bugs20130715

Here are the known bugs in [MLton 20130715](#), listed in reverse chronological order of date reported.

16 CallGraph

For easier visualization of [profiling](#) data, `mlprof` can create a call graph of the program in dot format, from which you can use the [graphviz](#) software package to create a PostScript or PNG graph. For example,

```
mlprof -call-graph foo.dot foo mlmon.out
```

will create `foo.dot` with a complete call graph. For each source function, there will be one node in the graph that contains the function name (and source position with `-show-line true`), as well as the percentage of ticks. If you want to create a call graph for your program without any profiling data, you can simply call `mlprof` without any `mlmon.out` files, as in

```
mlprof -call-graph foo.dot foo
```

Because SML has higher-order functions, the call graph is dependent on MLton's analysis of which functions call each other. This analysis depends on many implementation details and might display spurious edges that a human could conclude are impossible. However, in practice, the call graphs tend to be very accurate.

Because call graphs can get big, `mlprof` provides the `-keep` option to specify the nodes that you would like to see. This option also controls which functions appear in the table that `mlprof` prints. The argument to `-keep` is an expression describing a set of source functions (i.e. graph nodes). The expression *e* should be of the following form.

- `all`
- `"s"`
- `(and e ...)`
- `(from e)`
- `(not e)`
- `(or e)`
- `(pred e)`
- `(succ e)`
- `(thresh x)`
- `(thresh-gc x)`
- `(thresh-stack x)`
- `(to e)`

In the grammar, `all` denotes the set of all nodes. `"s"` is a regular expression denoting the set of functions whose name (followed by a space and the source position) has a prefix matching the regexp. The `and`, `not`, and `or` expressions denote intersection, complement, and union, respectively. The `pred` and `succ` expressions add the set of immediate predecessors or successors to their argument, respectively. The `from` and `to` expressions denote the set of nodes that have paths from or to the set of nodes denoted by their arguments, respectively. Finally, `thresh`, `thresh-gc`, and `thresh-stack` denote the set of nodes whose percentage of ticks, gc ticks, or stack ticks, respectively, is greater than or equal to the real number *x*.

For example, if you want to see the entire call graph for a program, you can use `-keep all` (this is the default). If you want to see all nodes reachable from function `foo` in your program, you would use `-keep '(from "foo")'`. Or, if you want to see all the functions defined in subdirectory `bar` of your project that used at least 1% of the ticks, you would use

```
-keep '(and ".*bar/" (thresh 1.0))'
```

To see all functions with ticks above a threshold, you can also use `-thresh x`, which is an abbreviation for `-keep '(thresh x)'`. You can not use multiple `-keep` arguments or both `-keep` and `-thresh`. When you use `-keep` to display a subset of the functions, `mlprof` will add dashed edges to the call graph to indicate a path in the original call graph from one function to another.

When compiling with `-profile-stack true`, you can use `mlprof -gray true` to make the nodes darker or lighter depending on whether their stack percentage is higher or lower.

MLton's optimizer may duplicate source functions for any of a number of reasons (functor duplication, monomorphisation, polyvariance, inlining). By default, all duplicates of a function are treated as one. If you would like to treat the duplicates separately, you can use `mlprof -split regexp`, which will cause all duplicates of functions whose name has a prefix matching the regular expression to be treated separately. This can be especially useful for higher-order utility functions like `General.o`.

16.1 Caveats

Technically speaking, `mlprof` produces a call-stack graph rather than a call graph, because it describes the set of possible call stacks. The difference is in how tail calls are displayed. For example if `f` nontail calls `g` and `g` tail calls `h`, then the call-stack graph has edges from `f` to `g` and `f` to `h`, while the call graph has edges from `f` to `g` and `g` to `h`. That is, a tail call from `g` to `h` removes `g` from the call stack and replaces it with `h`.

17 CallingFromCToSML

MLton's [ForeignFunctionInterface](#) allows programs to *export* SML functions to be called from C. Suppose you would like export from SML a function of type `real * char -> int` as the C function `foo`. MLton extends the syntax of SML to allow expressions like the following:

```
_export "foo": (real * char -> int) -> unit;
```

The above expression exports a C function named `foo`, with prototype

```
Int32 foo (Real64 x0, Char x1);
```

The `_export` expression denotes a function of type `(real * char -> int) -> unit` that when called with a function `f`, arranges for the exported `foo` function to call `f` when `foo` is called. So, for example, the following exports and defines `foo`.

```
val e = _export "foo": (real * char -> int) -> unit;
val _ = e (fn (x, c) => 13 + Real.floor x + Char.ord c)
```

The general form of an `_export` expression is

```
_export "C function name" attr... : cFuncTy -> unit;
```

The type and the semicolon are not optional. As with `_import`, a sequence of attributes may follow the function name.

MLton's `-export-header` option generates a C header file with prototypes for all of the functions exported from SML. Include this header file in your C files to type check calls to functions exported from SML. This header file includes `typedefs` for the [types that can be passed between SML and C](#).

17.1 Example

Suppose that `export.sml` is

```
val e = _export "f": (int * real * char -> char) -> unit;
val _ = e (fn (i, r, _) =>
    (print (concat ["i = ", Int.toString i,
                  " r = ", Real.toString r, "\n"]);
     # "g"))
val g = _import "g" public: unit -> unit;
val _ = g ()
val _ = g ()

val e = _export "f2": (Word8.word -> word array) -> unit;
val _ = e (fn w =>
    Array.tabulate (10, fn _ => Word.fromLargeWord (Word8.toLargeWord w)))
val g2 = _import "g2" public: unit -> word array;
val a = g2 ()
val _ = print (concat ["0wx", Word.toString (Array.sub (a, 0)), "\n"])

val e = _export "f3": (unit -> unit) -> unit;
val _ = e (fn () => print "hello\n");
val g3 = _import "g3" public: unit -> unit;
val _ = g3 ()

(* This example demonstrates mutual recursion between C and SML. *)
val e = _export "f4": (int -> unit) -> unit;
val g4 = _import "g4" public: int -> unit;
val _ = e (fn i => if i = 0 then () else g4 (i - 1))
val _ = g4 13

val (_, zzzSet) = _symbol "zzz" alloc: (unit -> int) * (int -> unit);
```

```

val () = zzzSet 42
val g5 = _import "g5" public: unit -> unit;
val _ = g5 ()

val _ = print "success\n"

```

Create the header file with `-export-header`.

```

% mlton -default-ann 'allowFFI true' \
        -export-header export.h \
        -stop tc \
        export.sml

```

`export.h` now contains the following C prototypes.

```

Int8 f (Int32 x0, Real64 x1, Int8 x2);
Pointer f2 (Word8 x0);
void f3 ();
void f4 (Int32 x0);
extern Int32 zzz;

```

Use `export.h` in a C program, `ffi-export.c`, as follows.

```

#include <stdio.h>
#include "export.h"

/* Functions in C are by default PUBLIC symbols */
void g () {
    Char8 c;

    fprintf (stderr, "g starting\n");
    c = f (13, 17.15, 'a');
    fprintf (stderr, "g done char = %c\n", c);
}

Pointer g2 () {
    Pointer res;
    fprintf (stderr, "g2 starting\n");
    res = f2 (0xFF);
    fprintf (stderr, "g2 done\n");
    return res;
}

void g3 () {
    fprintf (stderr, "g3 starting\n");
    f3 ();
    fprintf (stderr, "g3 done\n");
}

void g4 (Int32 i) {
    fprintf (stderr, "g4 (%d)\n", i);
    f4 (i);
}

void g5 () {
    fprintf (stderr, "g5 ()\n");
    fprintf (stderr, "zzz = %i\n", zzz);
    fprintf (stderr, "g5 done\n");
}

```

Compile `ffi-export.c` and `export.sml`.

```
% gcc -c ffi-export.c
% mlton -default-ann 'allowFFI true' \
    export.sml ffi-export.o
```

Finally, run `export`.

```
% ./export
g starting
...
g4 (0)
success
```

17.2 Download

- [export.sml](#)
- [ffi-export.c](#)

18 CallingFromSMLToC

MLton's [ForeignFunctionInterface](#) allows an SML program to *import* C functions. Suppose you would like to import from C a function with the following prototype:

```
int foo (double d, char c);
```

MLton extends the syntax of SML to allow expressions like the following:

```
_import "foo": real * char -> int;
```

This expression denotes a function of type `real * char -> int` whose behavior is implemented by calling the C function whose name is `foo`. Thinking in terms of C, imagine that there are C variables `d` of type `double`, `c` of type `unsigned char`, and `i` of type `int`. Then, the C statement `i = foo (d, c)` is executed and `i` is returned.

The general form of an `_import` expression is:

```
_import "C function name" attr... : cFuncTy;
```

The type and the semicolon are not optional.

The function name is followed by a (possibly empty) sequence of attributes, analogous to C `__attribute__` specifiers. For now, the only attributes supported are `cdecl` and `stdcall`. These specify the calling convention of the C function on Cygwin/Windows, and are ignored on all other platforms. The default is `cdecl`. You must use `stdcall` in order to correctly call Windows API functions.

18.1 Example

`import.sml` imports the C function `ffi` and the C variable `FFI_INT` as follows.

```
(* main.sml *)

(* Declare ffi to be implemented by calling the C function ffi. *)
val ffi = _import "ffi" public: real array * int * int ref * char ref * int -> char;
open Array

val size = 10
val a = tabulate (size, fn i => real i)
val ri = ref 0
val rc = ref #"0"
val n = 17

(* Call the C function *)
val c = ffi (a, Array.length a, ri, rc, n)

(* FFI_INT is declared as public in ffi-import.c *)
val (nGet, nSet) = _symbol "FFI_INT" public: (unit -> int) * (int -> unit);

val _ = print (concat [Int.toString (nGet ()), "\n"])

val _ =
  print (if c = #"c" andalso !ri = 45 andalso !rc = c
         then "success\n"
         else "fail\n")
```

`ffi-import.c` is

```
#include "export.h"

Int32 FFI_INT = 13;
```

```
Word32 FFI_WORD = 0xFF;
Bool FFI_BOOL = 1;
Real64 FFI_REAL = 3.14159;

Char8 ffi (Pointer a1, Int32 allen, Pointer a2, Pointer a3, Int32 n) {
    double *ds = (double*)a1;
    int *pi = (int*)a2;
    char *pc = (char*)a3;
    int i;
    double sum;

    sum = 0.0;
    for (i = 0; i < allen; ++i) {
        sum += ds[i];
        ds[i] += n;
    }
    *pi = (int)sum;
    *pc = 'c';
    return 'c';
}
```

Compile and run the program.

```
% mlton -default-ann 'allowFFI true' import.sml ffi-import.c
% ./import
13
success
```

18.2 Download

- [import.sml](#)
- [ffi-import.c](#)

18.3 Next Steps

- [CallingFromSMLToCFunctionPointer](#)

19 CallingFromSMLToCFunctionPointer

Just as MLton can [directly call C functions](#), it is possible to make indirect function calls; that is, function calls through a function pointer. MLton extends the syntax of SML to allow expressions like the following:

```
_import * : MLton.Pointer.t -> real * char -> int;
```

This expression denotes a function of type

```
MLton.Pointer.t -> real * char -> int
```

whose behavior is implemented by calling the C function at the address denoted by the `MLton.Pointer.t` argument, and supplying the C function two arguments, a double and an int. The C function pointer may be obtained, for example, by the dynamic linking loader (`dlopen`, `dlsym`, ...).

The general form of an indirect `_import` expression is:

```
_import * attr... : cPtrTy -> cFuncTy;
```

The type and the semicolon are not optional.

19.1 Example

This example uses `dlopen` and `friends` (imported using normal `_import`) to dynamically load the math library (`libm`) and call the `cos` function. Suppose `iimport.sml` contains the following.

```
signature DYN_LINK =
  sig
    type hndl
    type mode
    type fptr

    val dlopen : string * mode -> hndl
    val dlsym  : hndl * string -> fptr
    val dlclose : hndl -> unit

    val RTLD_LAZY : mode
    val RTLD_NOW  : mode
  end

structure DynLink :> DYN_LINK =
  struct
    type hndl = MLton.Pointer.t
    type mode = Word32.word
    type fptr = MLton.Pointer.t

    (* These symbols come from a system library, so the default import scope
     * of external is correct.
     *)
    val dlopen =
      _import "dlopen" : string * mode -> hndl;
    val dlerror =
      _import "dlerror": unit -> MLton.Pointer.t;
    val dlsym =
      _import "dlsym" : hndl * string -> fptr;
    val dlclose =
      _import "dlclose" : hndl -> Int32.int;

    val RTLD_LAZY = 0wx00001 (* Lazy function call binding. *)
    val RTLD_NOW  = 0wx00002 (* Immediate function call binding. *)
```

```

val dlerror = fn () =>
  let
    val addr = dlerror ()
  in
    if addr = MLton.Pointer.null
    then NONE
    else let
      fun loop (index, cs) =
        let
          val w = MLton.Pointer.getWord8 (addr, index)
          val c = Byte.byteToChar w
        in
          if c = #"\000"
          then SOME (implode (rev cs))
          else loop (index + 1, c::cs)
        end
      in
        loop (0, [])
      end
    end
end

val dlopen = fn (filename, mode) =>
  let
    val filename = filename ^ "\000"
    val hndl = dlopen (filename, mode)
  in
    if hndl = MLton.Pointer.null
    then raise Fail (case dlerror () of
      NONE => "???"
      | SOME s => s)
    else hndl
  end
end

val dlsym = fn (hndl, symbol) =>
  let
    val symbol = symbol ^ "\000"
    val fptr = dlsym (hndl, symbol)
  in
    case dlerror () of
      NONE => fptr
    | SOME s => raise Fail s
  end
end

val dlclose = fn hndl =>
  if MLton.Platform.OS.host = MLton.Platform.OS.Darwin
  then () (* Darwin reports the following error message if you
    * try to close a dynamic library.
    * "dynamic libraries cannot be closed"
    * So, we disable dlclose on Darwin.
    *)
  else
    let
      val res = dlclose hndl
    in
      if res = 0
      then ()
      else raise Fail (case dlerror () of
        NONE => "???"
        | SOME s => s)
      end
    end
end
end

```

```
val dll =
  let
    open MLton.Platform.OS
  in
    case host of
      Cygwin => "cygwin1.dll"
    | Darwin => "libm.dylib"
    | _ => "libm.so"
  end

val hndl = DynLink.dlopen (dll, DynLink.RTLD_LAZY)

local
  val double_to_double =
    _import * : DynLink.fptr -> real -> real;
  val cos_fptr = DynLink.dlsym (hndl, "cos")
in
  val cos = double_to_double cos_fptr
end

val _ = print (concat ["    Math.cos(2.0) = ", Real.toString (Math.cos 2.0), "\n",
                      "libm.so::cos(2.0) = ", Real.toString (cos 2.0), "\n"])

val _ = DynLink.dlclose hndl
```

Compile and run `iimport.sml`.

```
% mlton -default-ann 'allowFFI true' \
        -target-link-opt linux -ldl \
        -target-link-opt solaris -ldl \
        iimport.sml
% iimport
  Math.cos(2.0) = ~0.416146836547
libm.so::cos(2.0) = ~0.416146836547
```

This example also shows the `-target-link-opt` option, which uses the switch when linking only when on the specified platform. Compile with `-verbose 1` to see in more detail what's being passed to `gcc`.

19.2 Download

- [iimport.sml](#)

20 Changelog

- [changelog](#)

Here are the changes from version 2010608 to version 20130715.

Summary:

- + Compiler.
 - o Cosmetic improvements to type-error messages.
 - o Removed features:
 - * Bytecode codegen: The bytecode codegen had not seen significant use and it was not well understood by any of the active developers.
 - * Support for .cm files as input: The ML Basis system provides much better infrastructure for "programming in the very large" than the (very) limited support for CM. The cm2mlb tool (available in the source distribution) can be used to convert CM projects to MLB projects, preserving the CM scoping of module identifiers.
 - o Bug fixes: see changelog
- + Runtime.
 - o Bug fixes: see changelog
- + Language.
 - o Interpret (`#![line line:col "file" *]` directives as relative file names.
 - o ML Basis annotations.
 - * Added: `resolveScope`
- + Libraries.
 - o Basis Library.
 - * Improved performance of `String.concatWith`.
 - * Use bit operations for `REAL.class` and other low-level operations.
 - * Support additional variables with `Posix.ProcEnv.sysconf`.
 - * Bug fixes: see changelog
 - o MLton structure.
 - * Removed: `MLton.Socket`
 - o Other libraries.
 - * Updated: `ckit` library, `MLRISC` library, `SML/NJ` library
 - * Added: `MLLPT` library
- + Tools.
 - o `mllex`
 - * Generate (`#![line line:col "file.lex" *]` directives with simple (relative) file names, rather than absolute paths.
 - o `mlyacc`
 - * Generate (`#![line line:col "file.grm" *]` directives with simple (relative) file names, rather than absolute paths.
 - * Fixed bug in comment-handling in lexer.
- * 2013-07-06
 - Update `SML/NJ` libraries to `SML/NJ 110.76`.
- * 2013-06-19
 - Upgrade `gdtoa.tgz`; fixed bug in `'Real32.{fmt,toDecimal,toString}'`, which in some cases produced too many digits
- * 2013-06-18
 - Removed `MLton.Socket` structure (deprecated in last release).
- * 2013-06-10
 - Improved performance of `String.concatWith`.

- * 2013-05-22
 - Update SML/NJ libraries to SML/NJ 110.75.
- * 2013-04-30
 - Detect PowerPC 64 architecture.
- * 2012-10-09
 - Fixed bug in elaboration that erroneously accepted the following:
signature S = sig structure A : sig type t end
and B : sig type t end where type t = A.t end
- * 2012-09-04
 - Introduce an MLB annotation to control overload and flex record resolution scope: "resolveScope {strdec|dec|topdec|program}".
- * 2012-07-04
 - Simplify use of 'getsockopt' and 'setsockopt' in Basis Library.
 - Direct implementation of Socket.Ctl.{getATMARK,getNREAD} in runtime system, rather than indirect implementation in Basis Library via 'ioctl'.
 - Replace use of casting through a union with memcpy in runtime.
- * 2012-06-11
 - Use bit operations for REAL.class and other low-level operations.
 - Fixed bugs in REAL.copySign, REAL.signBit, and REAL.{to,from}Decimal.
- * 2012-06-01
 - Cosmetic improvements to type-error messages.
 - Fixed bug in elaboration that erroneously rejected the following:
datatype ('a, ''a) t = T
type ('a, ''a) u = unit
and erroneously accepted the following:
fun f (x: 'a) : ''a = x
fun g (x: 'a) : ''a = if x = x then x else x
- * 2012-02-24
 - Fixed bug in redundant SSA optimization.
- * 2011-06-20
 - Support additional variables with Posix.ProcEnv.sysconf.
- * 2011-06-17
 - Change mlex and mlyacc to generate #line directives with simple file names, rather than absolute paths.
 - Interpret "#line" directives as relative file names.
- * 2011-06-14
 - Fixed bug in SSA/SSA2 shrinker that could erroneously turn a non-tail function call with a Bug transfer as its continuation into a tail function call.
- * 2011-06-11
 - Update SML/NJ libraries to SML/NJ 110.73 and add ML-LPT library.
- * 2011-06-10
 - Fixed bug in translation from SSA2 to RSSA with case expressions over non-primitive-sized words.
 - Fixed bug in SSA/SSA2 type checking of case expressions over words.
- * 2011-06-04
 - Upgrade gdtoa.tgz.

- Remove bytecode codegen.
- Remove support for .cm files as input.
- * 2011-05-03
 - Fixed a bug with the treatment of as-patterns, which should not allow the redefinition of constructor status.
- * 2011-02-18
 - Fixed bug with treatment of nan in common subexpression elimination SSA optimization.
- * 2011-02-18
 - Fixed bug in translation from SSA2 to RSSA with weak pointers.
- * 2011-02-05
 - Fixed bug in amd64 codegen calling convention for varargs C calls.
- * 2011-01-17
 - Fixed bug in comment-handling in lexer for mlyacc's input language.
- * 2010-06-22
 - Fixed bug in elaboration of function clauses with different numbers of arguments that would raise an uncaught Subscript exception.

Here are the changes from version 20070826 to version 20100608.

Summary:

- + New platforms.
 - o ia64-hpux
 - o powerpc64-aix
- + Compiler.
 - o Command-line switches.
 - * Added: `-mlb-path-var '<name> <value>'`
 - * Removed: `-keep sml, -stop sml`
 - o Improved constant folding of floating-point operations.
 - o Experimental: Support for compiling to a C library; see wiki documentation.
 - o Extended `-show-def-use` output to include types of variable definitions.
 - o Deprecated features (to be removed in a future release)
 - * Bytecode codegen: The bytecode codegen has not seen significant use and it is not well understood by any of the active developers.
 - * Support for .cm files as input: The ML Basis system provides much better infrastructure for "programming in the very large" than the (very) limited support for CM. The `cm2mlb` tool (available in the source distribution) can be used to convert CM projects to MLB projects, preserving the CM scoping of module identifiers.
 - o Bug fixes: see changelog
- + Runtime.
 - o @MLton switches.
 - * Added: `may-page-heap {false|true}`
 - o `may-page-heap`: By default, MLton will not page the heap to disk when unable to grow the heap to accommodate an allocation. (Previously, this behavior was the default, with no means to disable, with security an least-surprise issues.)
 - o Bug fixes: see changelog
- + Language.
 - o Allow numeric characters in ML Basis path variables.

- + Libraries.
 - o Basis Library.
 - * Bug fixes: see changelog.
 - o MLton structure.
 - * Added: MLton.equal, MLton.hash, MLton.Cont.isolate, MLton.GC.Statistics, MLton.Pointer.sizeofPointer, MLton.Socket.Address.toVector
 - * Changed:
 - * Deprecated: MLton.Socket
 - o Unsafe structure.
 - * Added versions of all of the monomorphic array and vector structures.
 - o Other libraries.
 - * Updated: ckit library, MLRISC library, SML/NJ library.
- + Tools.
 - o mllex
 - * Eliminated top-level 'type int = Int.int' in output.
 - * Include (*#line line:col "file.lex" *) directives in output.
 - * Added %posint command, to set the yypoint type and allow the lexing of multi-gigabyte files.
 - o mlnlffigen
 - * Added command-line switches -linkage archive and -linkage shared.
 - * Deprecated command-line switch -linkage static.
 - * Added support for ia64 and hppa targets.
 - o mlyacc
 - * Eliminated top-level 'type int = Int.int' in output.
 - * Include (*#line line:col "file.grm" *) directives in output.
- * 2010-05-12
 - Fixed bug in the mark-compact garbage collector where the C library's memcpy was used to move objects during the compaction phase; this could lead to heap corruption and segmentation faults with newer versions of gcc and/or glibc, which assume that src and dst in a memcpy do not overlap.
- * 2010-03-12
 - Fixed bug in elaboration of datatype declarations with withtype bindings.
- * 2009-12-11
 - Fixed performance bug in ref flatten SSA2 optimization.
- * 2009-12-09
 - Fixed performance bug in simplify types SSA optimization.
- * 2009-12-02
 - Fixed bug in amd64 codegen register allocation of indirect C calls.
- * 2009-09-17
 - Fixed bug in IntInf.scan and IntInf.fromString where leading spaces were only accepted if the stream had an explicit sign character.
- * 2009-07-10
 - Added combine conversions SSA optimization.
- * 2009-06-09
 - Removed deprecated command line switch -show-anns {false, true}.
- * 2009-04-18
 - Removed command line switches -keep sml and -stop sml. Their meaning was unclear with .mlb files; their effect with .cm files

can be achieved with `-stop f`.

- * 2009-04-16
 - Fixed bug in `IntInf.~>>` that could cause a `glibc` assertion failure.
- * 2009-04-01
 - Fixed exported type of `MLton.Process.reap`.
- * 2009-01-27
 - Added `MLton.Socket.Address.toVector` to get the network-byte-order representation of an IP address.
- * 2008-11-10
 - Fixed bug in `MLton.size` and `MLton.share` when tracing the current stack.
- * 2008-10-27
 - Fixed phantom typing of sockets by hiding the representation of socket types. Previously the representation of sockets was revealed rendering the phantom types useless.
- * 2008-10-10
 - Fixed bug in nexted `_export/_import` functions.
- * 2008-09-12
 - Improved constant folding of floating point operations.
- * 2008-08-20
 - Store the card/cross map at the end of the allocated ML heap; avoids possible out of memory errors when resizing the ML heap cannot be followed by a card/cross map allocation.
- * 2008-07-24
 - Added support for compiling to a C library. The relevant new compiler options are `'-ar'` and `'-format'`. Libraries are named based on the name of the `-export-header` file. Libraries have two extra methods:
 - * `NAME_open(argc, argv)` initializes the library and runs the SML code until it reaches the end of the program. If the SML code exits or raises an uncaught exception, the entire program will terminate.
 - * `NAME_close()` will execute any registered `atExit` functions, any outstanding finalizers, and frees the ML heap.
- * 2008-07-16
 - Fixed bug in the name mangling of `_import`-ed functions with the `stdcall` convention.
- * 2008-06-12
 - Added `MLton.Pointer.sizeofPointer`.
- * 2008-06-06
 - Added expert command line switch `-emit-main {true|false}`.
- * 2008-05-17
 - Fixed bug in Windows code to page the heap to disk when unable to grow the heap to a desired size.
Thanks to Sami Evangelista for the bug report.
- * 2008-05-10
 - Implemented `MLton.Cont.isolate`.
- * 2008-04-20

- Fixed bug in *NIX code to page the heap to disk when unable to grow the heap to a desired size.
Thanks to Nicolas Bertolotti for the bug report and patch.
- * 2008-04-07
 - More flexible active/paused stack resizing policy.
Removed thread-shrink-ratio runtime option.
Added stack-current-grow-ratio, stack-current-max-reserved-ratio, stack-current-permit-ratio, stack-current-shrink-ratio, stack-max-reserved-ratio, and stack-shrink-ratio runtime options.
- * 2008-04-07
 - Fixed bugs in Basis Library where the representations of OS.IO.iodesc, Posix.IO.file_desc, Posix.Signal.signal, Socket.sock, Socket.SOGK.sock_type as integers were exposed.
- * 2008-03-14
 - Added unsafe versions of all of the monomorphic array and vector structures.
- * 2008-03-02
 - Fixed bug in Basis Library where the representation of OS.Process.status as an integer was exposed.
- * 2008-02-13
 - Fixed space-safety bug in pass to flatten refs into containing data structure. Thanks to Daniel Spoonhower for the bug report and initial diagnosis and patch.
- * 2008-01-25
 - Various updates to GC statistics gathering. Some basic GC statistics can be accessed from SML by MLton.GC.Statistics.* functions.
- * 2008-01-24
 - Added primitive (structural) polymorphic hash.
- * 2008-01-21
 - Fixed frontend to accept "op longvid" patterns and expressions.
Thanks to Florian Weimer for the bug report.
- * 2008-01-17
 - Extended -show-def-use output to include types of variable definitions.
- * 2008-01-09
 - Extended MLton_equal to be a structural equality on all types, including real and -> types.
- * 2007-12-18
 - Changed ML-Yacc and ML-Lex to output line directives so that MLton's def-use information points to the source files (.grm and .lex) instead of the generated implementations (.grm.sml and .lex.sml).
- * 2007-12-14
 - Added runtime option 'may-page-heap {false|true}'. By default, MLton will not page the heap to disk when unable to grow the heap to a desired size. (Previously, this behavior was the default, with no means to disable, with security and least-surprise concerns.)
Thanks to Wesley Terpstra for the patch.

- Fixed bug the FFI visible representation of `Int16.int ref` (and references of other primitive types smaller than 32-bits) on big-endian platforms.
Thanks to Dave Herman for the bug report.
- * 2007-12-13
 - Fixed bug in `ImperativeIOExtra.canInput` (`TextIO.canInput`).
Thanks to Ville Laurikari for the bug report.
- * 2007-12-09
 - Better constant folding of `IntInf` operations.
- * 2007-12-07
 - Fixed bug in algebraic simplification of real primitives.
`Real.<= (x, x)` is false when `x` is `NaN`.
- * 2007-11-29
 - Fixed bug in type inference of flexible records. This would later cause the compiler to raise the `TypeError` exception.
Thanks to Wesley Terpstra for the bug report.
- * 2007-11-28
 - Fixed bug in cross-compilation of `gdtoa` library. Thanks to Wesley Terpstra for the bug report and patch.
- * 2007-11-20
 - Fixed bug in pass to flatten refs into containing data structure.
Thanks to Ruy LeyWild for the bug report.
- * 2007-11-19
 - Fixed bug in the handling of weak pointers by the mark-compact garbage collector. Thanks to Sean McLaughlin for the bug report and Florian Weimer for the initial diagnosis.
- * 2007-11-07
 - Added `%posint` command to `ml-lex`, to set the `yypos` type and allow the lexing of multi-gigabyte input files. Thanks to Florian Weimer for the feature concept and original patch.
- * 2007-11-07
 - Added command-line switch `-mlb-path-var '<name> <value>'` for specifying MLB path variables.
- * 2007-11-06
 - Allow numeric characters in MLB path variables.
- * 2007-09-20
 - Fixed bug in elaboration of structures with signature constraints. This would later cause the compiler to raise the `TypeError` exception. Thanks to Vesa Karvonen for the bug report.
- * 2007-09-11
 - Fixed bug in interaction of `_export-ed` functions and signal handlers. Thanks to Sean McLaughlin for the bug report.
- * 2007-09-03
 - Fixed bug in implementation of `_export-ed` functions using `'char'` type. Thanks to Katsuhiko Ueno for the bug report.

Here are the changes from version 20051202 to version 20070826.

Summary:

- + New platforms:
 - o amd64-linux, amd64-freebsd
 - o hppa-hpux
 - o powerpc-aix
 - o x86-darwin (Mac OS X)
 - + Compiler.
 - o Support for 64-bit platforms.
 - * Native amd64 codegen.
 - o Command-line switches.
 - * Added: `-codegen amd64`, `-codegen x86`, `-default-type <type>`, `-profile-val {false|true}`.
 - * Changed: `-stop f` (file listing now includes `.mlb` files)
 - o Bytecode codegen.
 - * Support for profiling.
 - * Support for exception history.
 - + Language.
 - o ML Basis annotations.
 - * Removed: `allowExport`, `allowImport`, `sequenceUnit`, `warnMatch`.
 - + Libraries.
 - o Basis Library.
 - * Added: `PackWord16Big`, `PackWord16Little`, `PackWord64Big`, `PackWord64Little`.
 - * Bug Fixes: see changelog.
 - o MLton structure.
 - * Added: `MLTON_MONO_ARRAY`, `MLTON_MONO_VECTOR`, `MLTON_REAL`, `MLton.BinIO.tempPrefix`, `MLton.CharArray`, `MLton.CharVector`, `MLton.IntInf.BigWord`, `MLton.IntInf.SmallInt`, `MLton.Exn.defaultTopLevelHandler`, `MLton.Exn.getTopLevelHandler`, `MLton.Exn.setTopLevelHandler`, `MLton.LargeReal`, `MLton.LargeWord`, `MLton.Real`, `MLton.Real32`, `MLton.Real64`, `MLton.Rlimit.Rlim`, `MLton.TextIO.tempPrefix`, `MLton.Vector.create`, `MLton.Word.bswap`, `MLton.Word8.bswap`, `MLton.Word16`, `MLton.Word32`, `MLton.Word64`, `MLton.Word8Array`, `MLton.Word8Vector`.
 - * Changed: `MLton.Array.unfoldi`, `MLton.IntInf.rep`, `MLton.Rlimit`, `MLton.Vector.unfoldi`.
 - * Deprecated: `MLton.Socket`
 - o Other libraries.
 - * Added: MLRISC library.
 - * Updated: ckit library, SML/NJ library.
 - + Tools.
- * 2007-08-12
- Removed deprecated ML Basis annotations.
- * 2007-08-06
- Fixed bug in treatment of `Real<N>.{scan,fromString}` operations. `Real<N>.{scan,fromString}` were using `TO_NEAREST` semantics, but should obey current rounding mode. (Only `Real<N>.fromDecimal` is specified to always have `TO_NEAREST` semantics.) Thanks to Sean McLaughlin for the bug report.
- * 2007-07-27
- Fixed bugs in constant-folding of floating-point operations with C codegen.
- * 2007-07-26
- Fixed bug in treatment of floating-point operations. Floating-point operations depend on the current rounding mode, but were being treated as pure. Thanks to Sean McLaughlin for the bug report.

- * 2007-07-13
 - Added `MLton.Exn.{default,get,set}TopLevelHandler`.
- * 2007-07-12
 - Restored native option to `-codegen` flag.
- * 2007-07-11
 - Fixed bug in `Real32.toInt`: conversion of real values close to `Int.maxInt` could be incorrect.
- * 2007-07-07
 - Updates to bytecode code generator: support for amd64-* targets, support for profiling (including exception history).
 - Fixed bug in `Socket` module of Basis Library; unmarshalling of socket options (for `get*` functions) used `andb` rather than `orb`. Thanks to Anders Petersson for the bug report (and patch).
- * 2007-07-06
 - Fixed bug in `Date` module of Basis Library; some functions would erroneously raise `Date` when given a year ≤ 1900 . Thanks to Joe Hurd for the bug report.
 - Fixed a long-standing bug in monomorphisation pass. Thanks to Vesa Karvonen for the bug report.
- * 2007-05-18
 - Native amd64 code generator for amd64-* targets.
 - Eliminate native option from `-codegen` flag.
 - Add x86 and amd64 options to `-codegen` flag.
- * 2007-04-29
 - Improved type checking of `RSSA` and `Machine` ILs.
- * 2007-04-14
 - Fixed aliasing issues with `basis/Real/*.c` files.
 - Added `real/word` casts in `MLton` structure.
- * 2007-04-12
 - Added primitives for bit cast of word to/from real.
 - Implement `PackReal<N>{Big, Little}` using `PackWord<N>{Big, Little}` and bit casts.
- * 2007-04-11
 - Move all system header `#include-s` to `platform/ os` headers.
 - Use C99 `<assert.h>`, rather than custom `"assert.{h,c}"`.
- * 2007-03-13
 - Implement `PackWord<N>{Big, Little}` entirely in `ML`, using an `ML` byte swap function.
- * 2007-02-25
 - Change amd64-* target platforms from 32-bit compatibility mode (i.e., `-m32`) to 64-bit mode (i.e., `-m64`). Currently, only the `C` codegen is able to generate 64-bit executables.
- * 2007-02-23
 - Removed expert command line switch `-coalesce <n>`.
 - Added expert command line switch `-chunkify {coalesce<n>|func|one}`.
- * 2007-02-20
 - Fixed bug in `PackReal<N>.toBytes`. Thanks to Eric McCorkle for the bug report (and patch).

- * 2007-02-18
 - Added command line switch `-profile-val`, to profile the evaluation of val bindings; this is particularly useful with exception history for debugging uncaught exceptions at the top-level.
- * 2006-12-29
 - Added command line switch `-show {anns|path-map}` and deprecated command line switch `-show-anns {false|true}`. Use `-show path-map` to see the complete MLB path map as seen by the compiler.
- * 2006-12-20
 - Changed the output of command line switch `-stop f` to include `mlb-files`. This is useful for generating Makefile dependencies. The old output is easy to recover if necessary (e.g. `grep -v '\.mlb$'`).
- * 2006-12-8
 - Added command line switches `-{,target}-{as,cc,link}-opt-quote`, which pass their argument as a single argument to gcc (i.e., without tokenization at spaces). These options support using headers and libraries (including the MLton runtime headers and libraries) from a path with spaces.
- * 2006-12-02
 - Extensive reorganization of garbage collector, runtime system, and Basis Library implementation. (This is in preparation for future 64bit support.) They should be more C standards compliant and easier to port to new systems.
 - FFI revisions
 - Disallow nested indirect types (e.g., `int array array`).
- * 2006-11-30
 - Fixed a bug in elaboration of FFI forms; unary FFI types (e.g., `array`, `ref`, `vector`) could be used in places where `MLton.Pointer.t` was required. This would later cause the compiler to raise the `TypeError` exception, along with a lot of XML IL.
- * 2006-11-19
 - On `*-darwin`, work with GnuMP installed via Fink or MacPorts.
- * 2006-10-30
 - Ported to `x86-darwin`.
- * 2006-09-23
 - Added missing specification of `find` to the `MONO_VECTOR` signature.
- * 2006-08-03
 - Fixed a bug in the "useless" SSA optimization, caused by calling an imported C function and then ignoring the result.
- * 2006-06-24
 - Fixed a bug in `pass` to flatten data structures. Thanks to Joe Hurd for the bug report.
- * 2006-06-08
 - Fixed a bug in the native codegen's implementation of the C-calling convention.
- * 2006-05-11
 - Ported to PowerPC-AIX.
 - Fixed a bug in the runtime for the cases where nonblocking IO with sockets was implemented using `MSG_DONTWAIT`. This flag does not

exist on AIX, Cygwin, HP/UX, and MinGW and was previously just ignored. Now the runtime simulates the flag for these platforms (except MinGW, yet, where it's still ignored).

- * 2006-05-06
 - Added `-default-type '<ty><N>'` for specifying the binding of default types in the Basis Library (e.g., `Int.int`).
- * 2006-04-25
 - Ported to HPPA-HP/UX.
 - Fixed `PackReal{,32,64}{Big,Little}` to follow the Basis Library specification.
- * 2006-04-19
 - Fixed a bug in `MLton.share` that could cause a segfault.
- * 2006-03-30
 - Changed `MLton.Vector.unfoldi` to return the state in addition to the result vector.
- * 2006-03-30
 - Added `MLton.Vector.create`, a more powerful vector-creation function than is available in the basis library.
- * 2006-03-04
 - Added MLRISC from SML/NJ 110.57 to standard distribution.
- * 2006-03-03
 - Fixed bug in SSA simplifier that could eliminate an irredundant test.
- * 2006-03-02
 - Ported a bugfix from SML/NJ for a bug with the combination of `withNack` and `never` in CML.
- * 2006-02-09
 - Support compiler specific annotations in ML Basis files. If an annotation contains `:"`, then the text preceding the `:"` is meant to denote a compiler. For MLton, if the text preceding the `:"` is equal to `"mlton"`, then the remaining annotation is scanned as a normal annotation. If the text preceding the `:"` is not-equal to `"mlton"`, then the annotation is ignored, and no warning is issued.
- * 2006-02-04
 - Fixed bug in elaboration of functors; a program with a very large number of functors could exhibit the error `"ElaborateEnv.functorClosure: firstTycons"`.

 Here are the changes from version 20041109 to version 20051202.

Summary:

- + New license: BSD-style instead of GPL.
- + New platforms:
 - o hppa: Debian Linux.
 - o x86: MinGW.
- + Compiler.
 - o improved exception history.
 - o Command-line switches.
 - * Added: `-as-opt`, `-mlb-path-map`, `-target-as-opt`, `-target-cc-opt`.
 - * Deprecated: none.
 - * Removed: `-native`, `-sequence-unit`, `-warn-match`, `-warn-unused`.

- + Language.
 - o FFI syntax changes and extensions.
 - * Added: `_symbol`.
 - * Changed: `_export`, `_import`.
 - * Removed: `_ffi`.
 - o ML Basis annotations.
 - * Added: `allowFFI`, `nonexhaustiveExnMatch`, `nonexhaustiveMatch`, `redundantMatch`, `sequenceNonUnit`.
 - * Deprecated: `allowExport`, `allowImport`, `sequenceUnit`, `warnMatch`.
- + Libraries.
 - o Basis Library.
 - * Added: `Int1`, `Word1`.
 - o MLton structure.
 - * Added: `Process.create`, `ProcEnv.setgroups`, `Rusage.measureGC`, `Socket.fdToSock` `Socket.Ctl.getError`.
 - * Changed: `MLton.Platform.Arch`.
 - o Other libraries.
 - * Added: `ckit` library, `ML-NLFFI` library, `SML/NJ` library.
- + Tools.
 - o updates of `mlex` and `mlyacc` from `SML/NJ`.
 - o added `mlnlffigen`.
 - o profiling supports better inclusion/exclusion of code.
- * 2005-11-19
 - Updated `SML/NJ` Library and `CKit` Library from `SML/NJ 110.57`.
- * 2005-11-15
 - Fixed a bug in `MLton.ProcEnv.setgroups`.
- * 2005-11-11
 - Fixed a bug in the interleaving of lexing/parsing and elaborating of ML Basis files, which would raise an unhandled `Force` exception on cyclic basis references. Thanks to John Dias for the bug report.
- * 2005-11-10
 - Fixed two bugs in `Time.scan`. One would raise `Time` on a string with a large fractional component. Thanks to Carsten Varming for the bug report. The other failed to scan strings with an explicit sign followed by a decimal point.
- * 2005-11-03
 - Removed `MLton.GC.setRusage`.
 - Added `MLton.Rusage.measureGC`.
- * 2005-09-11
 - Fixed bug in display of types with large numbers of type variables, which could cause unhandled exception `Chr`.
- * 2005-09-08
 - Fixed bug in type inference of flexible records that would show up as "Type error: variable applied to wrong number of type args".
- * 2005-09-06
 - Fixed bug in `Real.signBit`, which had assumed that the underlying C signbit returned 0 or 1, when in fact any nonzero value is allowed to indicate the signbit is set.
- * 2005-09-05
 - Added `-mlb-path-map` switch.
- * 2005-08-25
 - Fixed bug in `MLton.Finalizable.touch`, which was not keeping alive

finalizable values in all cases.

- * 2005-08-18
 - Added SML/NJ Library and CKit Library from SML/NJ 110.55 to standard distribution.
 - Fixed bug in Socket.Ctl.*, which got the endianness wrong on big-endian machines. Thanks to Wesley Terpstra for the bug report and fix.
 - Added MLton.GC.setRusage.
 - Fixed bug in mllex, which had file positions starting at 2. They now start at zero.
- * 2005-08-15
 - Fixed bug in LargeInt.scan, which should skip leading "0x" and "0X". Thanks to Wesley Terpstra for the bug report and fix.
- * 2005-08-06
 - Additional revisions of FFI.
 - Deprecated `_export` with incomplete annotation.
 - Added `_address` for address of C objects.
 - Eliminated address component of `_symbol`.
 - Changed the type of the `_symbol*` expression.
 - See documentation for more detail.
- * 2005-08-06
 - Annotation changes.
 - Deprecated: `sequenceUnit`
 - Added: `sequenceNonUnit`
- * 2005-08-03
 - Annotation changes.
 - Deprecated: `allowExport`, `allowImport`, `warnMatch`
 - Added: `allowFFI`, `nonexhaustiveExnMatch`, `nonexhaustiveMatch`, `redundantMatch`
- * 2005-08-01
 - Update mllex and mlyacc with SML/NJ 110.55+ versions. This incorporates a small number of minor bug fixes.
- * 2005-07-23
 - Fixed bug in pass to flatten refs into containing data structure.
- * 2005-07-23
 - Overhaul of FFI.
 - Deprecated `_import` of C base types.
 - Added `_symbol` for address, getter, and setter of C base types.
 - See documentation for more detail.
- * 2005-07-21
 - Update mllex and mlyacc with SML/NJ 110.55 versions. This incorporates a small number of minor bug fixes.
- * 2005-07-20
 - Fixed bug in front end that allowed unary constructors to be used without an argument in patterns.
- * 2005-07-19
 - Eliminated `_ffi`, which has been deprecated for some time.
- * 2005-07-14
 - Fixed bug in runtime that caused `getRusage` to be called on every GC, even if timing info isn't needed.

- * 2005-07-13
 - Fixed bug in closure conversion tickled by making a weak pointer to a closure.
- * 2005-07-12
 - Changed `{OS,Posix}.Process.sleep` to call `nanosleep()` instead of `sleep()`.
 - Added `MLton.ProcEnv.setgroups`.
- * 2005-07-11
 - `InetSock.{any,toAddr}` raise `SysErr` if port is not in `[0, 2^16)`.
- * 2005-07-02
 - Fixed bug in `Socket.recvVecFrom{,','NB,NB'}`. The type was too polymorphic and allowed the creation of a bogus `sock_addr`.
- * 2005-06-28
 - The front end now reports errors on encountering undefined or cyclicly defined MLB path variables.
- * 2005-05-22
 - Fixed bug in `Posix.IO.{getlk,setlk,setlkw}` that caused a link-time error: undefined reference to `Posix_IO_FLock_typ`.
 - Improved exception history so that the first entry in the history is the source position of the raise, and the rest is the call stack.
- * 2005-05-19
 - Improved exception history for Overflow exceptions.
- * 2005-04-20
 - Fixed a bug in `pass` to flatten refs into containing data structure.
- * 2005-04-14
 - Fixed a front-end bug that could cause an internal bug message of the form "missing flexInst".
- * 2005-04-13
 - Fixed a bug in the representation of flat arrays/vectors that caused incorrect behavior when the element size was 2 or 4 bytes and there were multiple components to the element (e.g. `(char * char) vector`).
- * 2005-04-01
 - Fixed a bug in `GC_arrayAllocate` that could cause a segfault.
- * 2005-03-22
 - Added structures `Int1, Word1`.
- * 2005-03-19
 - Fixed a bug that caused `Socket.Ctl.{get,set}LINGER` to raise `Subscript`. The problem was in the use of `PackWord32Little.update`, which scales the supplied index by `bytesPerElem`.
- * 2005-03-13
 - Fixed a bug in CML mailboxes.
- * 2005-02-26
 - Fixed an off-by-one error in `mkstemp` defined in `mingw.c`.
- * 2005-02-13

- Added mlnlffigen tool (heavily adapted from SML/NJ).
- * 2005-02-12
 - Added MLNLFFI Library (heavily adapted from SML/NJ) to standard distribution.
- * 2005-02-04
 - Fixed a bug in OS.path.toString, which did not raise InvalidArc when needed.
- * 2005-02-03
 - Fixed a bug in OS.Path.joinDirFile, which did not raise InvalidArc when passed a file that was not an arc.
- * 2005-01-26
 - Fixed a front end bug that incorrectly rejected expansive valbinds with useless bound type variables.
- * 2005-01-22
 - Fixed x86 codegen bug which failed to account for the possibility that a 64-bit move could interfere with itself (as simulated by 32-bit moves).
- * 2004-12-22
 - Fixed Real32.fmt StringCvt.EXACT, which had been producing too many digits of precision because it was converting to a Real64.real.
- * 2004-12-15
 - Replaced MLB path variable MLTON_ROOT with SML_LIB, to use a more compiler-independent name. We will keep MLTON_ROOT aliased to SML_LIB until after the next release.
- * 2004-12-02
 - Unix.create now works on all platforms (including Cygwin and MinGW).
- * 2004-11-24
 - Added support for MLton.Process.create, which works on all platforms (including Windows-based ones like Cygwin and MinGW) and allows better control over std{in,out,err} for child process.

Here are the changes from version 20040227 to 20041109.

Summary:

- + New platforms:
 - o x86: FreeBSD 5.x, OpenBSD
 - o PowerPC: Darwin (MacOSX)
- + Support for MLBasis files.
- + Support for dynamic libraries.
- + Support for Concurrent ML (CML).
- + New structures: Int2, Int3, ..., Int31 and Word2, Word3, ..., Word31.
- + A new form of profiling, -profile count.
- + A bytecode generator.
- + Data representation improvements.
- + MLton structure changes.
 - o Added: share, shareAll
 - o Changed: Exn, IntInf, Signal, Thread.
- + Command-line switch changes.
 - o Deprecated:
 - native (use -codegen)

```
-sequence-unit (use -default-ann)
-warn-match (use -default-ann)
-warn-unused (use -default-ann)
o Removed:
  -detect-overflow
  -exn-history (use -const)
  -safe
  -show-basis-used
o Added:
  -codegen
  -const
  -default-ann
  -disable-ann
  -profile-branch
  -target-link-opt

* 2004-09-22
- Extended _import to support indirect function calls.

* 2004-09-13
- Made Date.{fromString,scan} accept a space (treated as zero) in
  the first character of the day of the month.

* 2004-09-12
- Fixed bug in IntInf that could cause a seg fault.
- Remove MLton.IntInf.size.

* 2004-09-05
- Made -detect-overflow and -safe expert options.

* 2004-08-30
- Added val MLton.share: 'a -> unit, which maximizes sharing in a
  heap object.

* 2004-08-27
- Fixed bug in Real.toLargeInt. It would incorrectly raise Option
  instead of Overflow in the case when the real was not an INF, but
  rounding produced an INF.
- Fixed bugs in Date.{fmt,fromString,scan,toString}. They
  incorrectly allowed a space for the first character in the day of
  the month.

* 2004-08-18
- Changed MLton.{Thread,Signal,World} to distinguish between
  implicitly and explicitly paused threads.

* 2004-07-28
- Added support for programming in the large using the ML Basis
  system.

* 2004-07-11
- Fixed bugs in ListPair.*Eq functions, which incorrectly raised
  the UnequalLengths exception.

* 2004-07-01
- Added val MLton.Exn.addExnMessenger: (exn -> string option) -> unit

* 2004-06-23
- Runtime system options that take memory sizes now accept a "g"
  suffix indicating gigabytes. They also now take a real instead of
  an integer, e.g. fixed-heap 0.5g. They also now accept uppercase,
  e.g. 150M.
```

- * 2004-06-12
 - Added support for OpenBSD.
- * 2004-06-10
 - Added support for FreeBSD 5.x.
- * 2004-05-28
 - Deprecated the `-native` flag. Instead, use the new flag `-codegen {native|bytecode|C}`. This is in anticipation of adding a bytecode compiler.
- * 2004-05-26
 - Fixed a front-end bug that could cause cascading error to print a very large and unreadable internal bug message of the form `"datatype ... realized with scheme Unknown"`.
- * 2004-05-17
 - Automatically restart functions in the Basis Library that correspond directly to interruptable system calls.
- * 2004-05-13
 - Added `-profile count`, for dynamic counts of function calls and branches.
 - Equate the types `Posix.Signal.signal` and `Unix.signal`.
- * 2004-05-11
 - Fixed a bug with `-basis 1997` that would cause type errors due to differences between types in the MLton structure and types in the rest of the basis library.
- * 2004-05-01
 - Fixed a bug with sharing constraints in signatures that would sometimes mistakenly treat two structures as identical when they shouldn't have been. This would cause some programs to be mistakenly rejected.
- * 2004-04-30
 - Added `MLton.Signal.{handled,restart}`.
- * 2004-04-23
 - Added `Timer.checkCPUTimes`, and updated the `Timer` structure to match the latest basis spec. Also fixed `totalCPUTimer` and `totalRealTimer`, which were wrong.
- * 2004-04-13
 - Added `MLton.Signal.Mask.{getBlocked,isMember}`.
- * 2004-04-12
 - Fix bug that mistakenly generalized variable types containing unknown types when matching against a signature.
 - Reasonable front-end error message when unification causes recursive (circular) type.
- * 2004-04-03
 - Fixed bug in sharing constraints so that `A = B = C` means that all pairs `AB`, `AC`, `BC` are shared, not just `AB` and `BC`. This matters in some situations.
- * 2004-03-20
 - Fixed `Time.now` which was treating microseconds as nanoseconds.
- * 2004-03-14

- Fixed SSA optimizer bug that could cause the error "<type> has no tyconInfo property".
- * 2004-03-11
 - Fixed Time.fromReal to raise Time, not Overflow, on unrepresentable times.
- * 2004-03-04
 - Added structures Word2, Word3, ..., Word31.
- * 2004-03-03
 - Added structures Int2, Int3, ..., Int31.
 - Fixed bug in elaboration of "and" with signatures, structures, and functors so that it now evaluates all right-hand sides before binding any left-hand sides.

 Here are the changes from version 20030716 to 20040227.

Summary:

- + The front end now follows the Definition of SML and produces readable error messages.
- + Added support for NetBSD.
- + Basis library changes tracking revisions to the specification.
- + Added structures: Int64, Real32, Word64.
- + File positions use Int64.
- + Major improvements to -show-basis, which now displays the basis in a very readable way with full type information.
- + Command-line switch changes.
 - o Deprecated: -basis.
 - o Removed: -lib-search, -link, -may-load-world, -static.
 - o Added: -link-opt, -runtime, -sequence-unit, -show-def-use, -stop tc, -warn-match, -warn-unused.
 - o Changed: -export-header, -show-basis, -show-basis-used.
 - o Renamed: -host to -target.
- + FFI changes.
 - o Renamed _ffi as _import.
 - o Added cdecl and stdcall attributes to _import and _export expressions.
- + MLton structure changes.
 - o Added: Pointer.
 - o Removed: Ptrace.
 - o Changed: Finalizable, IntInf, Platform, Random, Signal, Word.
- * 2004-02-16
 - Changed -export-header, -show-basis, -show-basis-used to take a file name argument, and they no longer force compilation to halt.
 - Added -show-def-use and -warn-unused, which deal with def-use information.
- * 2004-02-13
 - Added flag -sequence-unit, which imposes the constraint that in the sequence expression (e1; e2), e1 must be of type unit.
- * 2004-02-10
 - Lots of changes to MLton.Signal: name changes, removal of superfluous functions, additional functions.
- * 2004-02-09
 - Extended -show-basis so that when used with an input program, it shows the basis defined by the input program.
 - Added "stop" runtime argument.

- Made `-call-graph {false|true}` an option to `mlprof` that determines whether or not a call graph file is written.
- * 2004-01-20
 - Fixed a bug in `IEEEReal.{fromString,scan}`, which would improperly return `INF` instead of `ZERO` for things like `"0.0000e123456789012345"`.
 - Fixed a bug in `Real.{fromDecimal,fromString,scan}`, which didn't return an appropriately signed zero for `~0.0`.
 - Fixed a bug in `Real.{toDecimal,fmt}`, which didn't correctly handle `~0.0`.
 - Report a compile-time error on unrepresentable real constants.
- * 2004-01-05
 - Removed option `-may-load-world`. You can now use `-runtime no-load-world` instead.
 - Removed option `-static`. You can now use `-link-opt -static` instead.
 - Changed `MLton.IntInf.size` to return 0 instead of 1 on small ints.
- * 2003-12-28
 - Fixed horrible bug in `MLton.Random.alphaNumString` that caused it to return 0 for all characters beyond position 11.
- * 2003-12-17
 - Removed `-basis` as a normal flag. It is still available as an expert flag, but its use is deprecated. It will almost certainly disappear after the next release.
- * 2003-12-10
 - Allow multiple `@MLton -- runtime args` in sequence. This makes it easier for scripts to prefix `@MLton` args without having to splice them with other ones.
- * 2003-12-04
 - Added support for files larger than 2G. This included changing `Position` from `Int32` to `Int64`.
- * 2003-12-01
 - Added structure `MLton.Pointer`, which includes a type `t` for pointers (memory addresses, not SML heap pointers) and operations for loading from and storing to memory.
- * 2003-11-03
 - Fixed `Timer.checkGCtime` so that only the GC user time is included, not GC system time.
- * 2003-10-13
 - Added `-warn-match` to control display nonexhaustive and redundant match warnings.
 - Fixed space leak in `StreamIO` causing the entire stream to be retained. Thanks to Jared Showalter for the bug report and fix.
- * 2003-10-10
 - Added `"-stop tc"` switch to stop after type checking.
- * 2003-09-25
 - Fixed `Posix.IO.getfl`, which had mistakenly called `fcntl` with `F_GETFD` instead of `F_GETFL`.
 - Tracking basis library changes:
 - o Socket module datagram functions no longer return amount written, since they always write the entire amount or fail. So, `send{Arr,Vec}To{,}'` now return unit instead of int.

- o Added nonblocking versions of all the send and recv functions, as well as accept and connect. So, we now have:
acceptNB, connectNB, recv{Arr,Vec}{,From}NB{,'},
send{Arr,Vec}{,To}NB{,'}

* 2003-09-24

- Tracking basis library changes:
 - o TextIO.inputLine now returns a string option.
 - o Slices used in Byte, PRIM_IO, PrimIO, Posix.IO, StreamIO
 - o Posix.IO.readVec raises Size, not Subscript, with negative argument.

* 2003-09-22

- Fixed Real.toManExp so that the mantissa is in [0.5, 1), not [1, 2). The spec says that $1.0 \leq \text{man} * \text{radix} < \text{radix}$, which since radix is 2, implies that the mantissa is in [0.5, 1).
- Added Time.{from,to}Nanoseconds.

* 2003-09-11

- Added Real.realRound.
- Added Char{Array,Vector}Slice to Text.

* 2003-09-11

- OS.IO.poll and Socket.select now raise errors on negative timeouts.
- Time.time is now implemented using IntInf instead of Int, which means that a much larger range of time values is representable.

* 2003-09-10

- Word64 is now there.

* 2003-09-09

- Replaced Pack32{Big,Little} with PackWord32{Big,Little}.
- Fixed bug in OS.FileSys.fullPath, which mistakenly stopped as soon as it hit a symbolic link.

* 2003-09-08

- Fixed @MLton max-heap, which was mistakenly ignored. Cleaned up @MLton fixed-heap. Both fixed-heap and max-heap can use copying or mark-compact collection.

* 2003-09-06

- Int64 is completely there.
- Fixed OS.FileSys.tmpName so that it creates the file, and doesn't use tmpnam. This eliminates an annoying linker warning message.

* 2003-09-05

- Added structures {LargeInt, LargeReal, LargeWord, Word}
{Array, Array2, ArraySlice, Vector, VectorSlice}
- Fixed bug in Real.toDecimal, which return class NORMAL for subnormals.
- Fixed bug in Real.toLargeInt, which didn't return as precise an integer as possible.

* 2003-09-03

- Lots of fixes to Real functions.
 - o Real32 is now completely in place, except for Real32.nextAfter on SunOS.
 - o Fixed Real.Math.exp on x86 to return the right value when applied to posInf and negInf.
 - o Changed Real.Math.{cos, sin, tan} on x86 to always use a call to the C math library instead of using the x86 instruction. This

- eliminates some anomalies between compiling `-native false` and `-native true`.
- o Change `Real.Math.pow` to handle exceptional cases in the SML code.
- o Fixed `Real.signBit` on Sparcs.

* 2003-08-28

- Fixed `PackReal{,64}Little` to work correctly on Sparc.
- Added `PackReal{,64}Big`, `PackReal32{Big,Little}`.
- Added `-runtime` switch, which passes arguments to the runtime via `@MLton`. These arguments are processed before command line switches.
- Eliminated MLton switch `-may-load-world`. Can use `-runtime` combined with new runtime switch `-no-load-world` to disable load world in an executable.

* 2003-08-26

- Changed `-host` to `-target`.
- Split `MLton.Platform.{arch,os}` into `MLton.Platform.{Arch,OS}.t`.

* 2003-08-21

- Fixed bug in C codegen that would cause undefined references to `Real_{fetch,move,store}` when compiling on Sparcs with `-align 4`.

* 2003-08-17

- Eliminated `-link` and `-lib-search`, which are no longer needed. Eliminated support for passing `-l*`, `-L*`, and `*.a` on the command line. Use `-link-opt` instead.

* 2003-08-16

- Added `-link-opt`, for passing options to gcc when linking.

* 2003-07-19

- Renamed `_ffi` as `_import`. The old `_ffi` will remain for a while, but is deprecated and should be replaced with `_import`.
- Added attributes to `_export` and `_import`. For now, the only attributes are `"cdecl"` and `"stdcall"`.

Here are the changes from version 20030711 to 20030716.

Summary:

- + Fixed several serious bugs with the 20030711 release.

* 2003-07-15

- Fixed bug that caused a segfault when attempting to create an array that was too large, e.g


```
1 + Array.sub (Array.tabulate (valOf Int.maxInt, fn i => i), 0)
```
- mlton now checks the command line arguments following the file to compile that are passed to the linker to make sure they are reasonable.

* 2003-07-14

- Fixed packaging for Cygwin and Sparc to include `libgmp.a`.
- Eliminated bootstrap target. The Makefile automatically determines whether to bootstrap or not.
- Fixed XML type checker bug that could cause error: empty tyvars in `PolyVal dec`.

* 2003-07-12

- Turned off `FORCE_GENERATIONAL` in gc. It had been set, which caused the gc to always use generational collection. This could

seriously slow apps down that don't need it.

Here are the changes from version 20030312 to 20030711.

Summary:

- + Added support for Sparc/SunOS using the C code generator.
- + Completed the basis library implementation. At this point, the only missing basis library function is "use".
- + Added `_export`, which allows one to call SML functions from C.
- + Added weak pointers (via `MLton.Weak`) and finalization (via `MLton.Finalizable`).
- + Added new integer modules: `Int8`, `Int16`.
- + Better profiling call graphs
- + Fixed conversions between reals and their decimal representations to be correct using the `gdtoa` library.

* 2003-07-07

- Profiling improvements
 - o Eliminated `mlton -profile-split`. Added `mlprof -split`. Now the profiling infrastructure keeps track of the splits and allows one to decide which splits to make (if any) when `mlprof` is run, which is much better than having to decide at compile time.
 - o Changed `mlprof -graph` to `mlprof -keep`, and changed the behavior so that `-keep` also controls which functions are displayed in the table.
 - o Eliminated `mlprof -ignore`: it's behavior is now subsumed by `-keep`, whose meaning has changed to be more like `-ignore` on nodes that are not kept.
- When calling `gcc` for linking, put `-link` args in same order as they appeared on the MLton command line (they used to be reversed).

* 2003-07-03

- Making `OS.Process.{atExit,exit}` conform to the basis library spec in that exceptions raised during cleaners are caught and ignored. Also, calls to `exit` from cleaners cause the rest of cleaners to run.

* 2003-07-02

- Fixed bug with negative `IntInf` constants that could cause compile time error message:
"x86Translate.translateChunk ... strange Offset: base: ..."
- Changed argument type of `MLton.IntInf.Small` from `word` to `int`.
- Added fix to profiling so that the `mlmon.out` file is written even when the program terminates due to running out of memory.

* 2003-06-25

- Added `{Int{8,16},Word8}{,Array,ArraySlice,Vector,VectorSlice,Array2}` structures.

* 2003-06-25

- Fixed bug in `IntInf.sign`, which returned the wrong value for zero.

* 2003-06-24

- Added `_export`, for calling from C to SML.

* 2003-06-18

- Regularization of options.
 - `-diag --> -diag-pass`
 - `-drop-pass` takes a regexp

* 2003-06-06

- Fixed bug in OS.IO.poll that caused it to return the input event types polled for instead of what was actually available.
- * 2003-06-04
 - Fixed bug in known case SSA optimization that could cause incorrect results in compiled programs.
- * 2003-06-03
 - Fixed bug in SSA optimizer that could cause the error message
Type error: Type.equals
{from = char vector, to = unit vector}
Type error: analyze raised exception loopStatement: ...
unhandled exception: TypeError
- * 2003-06-02
 - Fixed Real.rem to work correctly on infs and nans.
 - Fixed bug in profiling that caused the function name to be omitted on functions defined by val rec.
- * 2003-05-31
 - Fixed Real.{fmt,fromString,scan,toString} to match the basis library spec.
 - Added IEEEReal.{fromString,scan}.
 - Added Real.{from,to}Decimal.
- * 2003-05-25
 - Added Real.nextAfter.
 - Added OS.Path.{from,to}UnixPath, which are the identity function on Unix.
- * 2003-05-20
 - Added type MLton.pointer, the type of C pointers, for use with the FFI.
- * 2003-05-18
 - Fixed two bugs in type inference that could cause the compiler to raise the TypeError exception, along with a lot of XML IL. The type-check.sml regression contains simple examples of what failed.
 - Fixed a bug in the simplifier that could cause the message: shrinker raised Prim.apply raised assertion failure: SmallIntInf.fromWord
- * 2003-05-15
 - Fixed bug in Real.class introduced on 04-28 that cause many regression failures with reals when using newer gccs.
 - Replaced MLton.Finalize with MLton.Finalizable, which has a more robust approach to finalization.
- * 2003-05-13
 - Fixed bug in MLton.FFI on Cygwin that caused Thread_returnToC to be undefined.
- * 2003-05-12
 - Added support for finalization with MLton.Finalize.
- * 2003-05-09
 - Fixed a runtime system bug that could cause a segfault. This bug would happen after a GC during heap resizing when copying a heap, if the heap was allocated at a very low (<10M) address. The bug actually showed up on a Cygwin system.
- * 2003-05-08

- Fixed bug in HashType that raised "Vector.forall2" when the arity of a type constructor is changed by simplifyTypes, but a newly constructed type has the same hash value.
- * 2003-05-02
 - Switched over to new layered IO implementation, which completes the implementation of the BinIO and TextIO modules.
- * 2003-04-28
 - Fixed bug that caused an assertion failure when generating a jump table for a case dispatch on a non-word sized index with non-zero lower bound on the range.
- * 2003-04-24
 - Added -align {4|8}, which controls alignment of objects. With -align 8, memory accesses to doubles are guaranteed to be aligned mod 8, and so don't need special routines to load or store.
- * 2003-04-22
 - Fixed bug that caused a total failure of time profiling with -native false. The bug was introduced with the C codegen improvements that split the C into multiple files. Now, the C codegen declares all profile labels used in each file so that they are global symbols.
- * 2003-04-18
 - Added MLton.Weak, which supports weak pointers.
- * 2003-04-10
 - Replaced the basis library's MLton.hostType with MLton.Platform.arch and MLton.Platform.os.
- * 2003-04
 - Added support for SPARC/SunOS using the C codegen.
- * 2003-03-25
 - Added MLton.FFI, which allows callbacks to SML from C.
- * 2003-03-21
 - Fixed mlprof so that the default -graph arg for data from -profile-stack true is '(thresh-stack x)', not '(thresh x)'.

 Here are the changes from version 20020923 to 20030312.

Summary:

- + Added source-level profiling of both time and allocation.
- + Updated basis library to 2002 specification. To obtain the old library, compile with -basis 1997.
- + Added many modules to basis library: BinPrimIO, GenericSock, ImperativeIO, INetSock, NetHostDB, NetProtDB, NetServDB, Socket, StreamIO, TextPrimIO, UnixSock.
- + Completed implementation of IntInf and OS.IO.
- * 2003-02-23
 - Replaced -profile-combine with -profile-split.
- * 2003-02-11
 - Regularization of options.
 - l --> -link
 - L --> -lib-search
 - o --> -output

-v --> -verbose

- * 2003-02-10
 - Added option to mlton: `-profile-combine {false|true}`
- * 2003-02-09
 - Added options to mlprof: `-graph-title`, `-gray`, `-ignore`, `-mlmon`, `-tolerant`.
- * 2002-11 - 2003-01
 - Added source-level allocation and time profiling. This includes the new options to mlton: `-profile` and `-profile-stack`.
- * 2002-12-28
 - Added `NetHostDB`, `NetProtDB`, `NetServDB` structures.
 - Added `Socket`, `GenericSock`, `INetSock`, `UnixSock` structures.
- * 2002-12-19
 - Fixed bug in signal check insertion that could cause some signals to be missed. The fix was to add a signal check on entry to each function in addition to at each loop header.
- * 2002-12-10
 - Fixed bug in runtime that might cause the message `Unable to set cardMapForMutator`.
- * 2002-11-23
 - Added support for the latest Basis Library specification.
 - Added option `-basis` to choose Basis Library version. Currently available basis libraries are 2002, 2002-strict, 1997, and none.
 - Added `IntInf.{orb,xorb,anb,notb,<<,~>>}` values.
 - Added `OS.IO.{poll_desc,poll_info}` types.
 - Added `OS.IO.{pollDesc,pollToIODesc,infoToPollDesc,Poll}` values.
 - Added `OS.IO.{pollIn,pollOut,pollPri,poll,isIn,isOut,isPri}` values.
 - Added `BinPrimIO`, `TextPrimIO` structures.
 - Added `StreamIO`, `ImperativeIO` functors.
- * 2002-11-22
 - Fixed bug that caused time profiling to fail (with a segfault) when resuming a saved world.
- * 2002-11-07
 - Fixed bug in `MLton.eq` that could arise when using `eq` on functions.
- * 2002-11-05
 - Improvements to polymorphic equality. Equality on `IntInfs`, vectors, and datatypes all do an `eq` test first before a more expensive comparison.
- * 2002-11-01
 - Added allocation profiling. Now, can compile with either `-profile alloc` or `-profile time`. Renamed `MLton.Profile` as `MLton.ProfileTime`. Added `MLton.ProfileAlloc`. Cleaned up and changed most mlprof option names.
- * 2002-10-31
 - Eliminated `MLton.debug`.
 - Fixed bug in the optimizer that affected `IntInf.fmt`. The optimizer had been always using base 10, instead of the passed in radix.
- * 2002-10-22
 - Fixed `Real.toManExp` so that the mantissa is in `[1, 2)`, not `[0.5, 1)`.
 - Added `Real.fromLargeInt`, `Real.toLargeInt`.
 - Fixed `Real.split`, which would return an incorrect whole part due to

the underlying primitive, `Real_modf`, being treated as functional instead of side-effecting.

- * 2002-09-30
 - Fixed `rpath` problem with packaging. All executables in packages previously made had included a setting for `RPATH`.

Here are the changes from version 20020410 to 20020923.

Summary:

- + MLton now runs on FreeBSD.
 - + Major runtime system improvements. The runtime now implements mark-compact and generational collection, in addition to the copying collection that was there before. It automatically switches between the the collection strategies to improve performance and to try to avoid paging.
 - + Performance when compiling "`-exn-history true`" has been improved.
 - + Added `IntInf.log2`, `MLton.GC.pack`, `MLton.GC.unpack`.
 - + Fixed bug in load world that could cause "sread failed" on Cygwin.
 - + Fixed optimizer bug that could cause "no analyze var value property" message.
- * 2002-09
 - Integrated Sam Rushing's changes to port MLton to FreeBSD.
 - * 2002-08-25
 - Changed the implementation of exception history to be completely functional. Now, the extra field in exceptions (when compiling `-exn-history true`) is a string list instead of a string list ref, and `raise` conses a new exception with a new element in the list instead of assigning to the list. This changes the semantics of exception history (for the better) on some programs. See `regression/exnHistory3.sml` for an example. It also significantly improves performance when compiling `-exn-history true`.
 - * 2002-07 and 2002-08
 - Added generational GC, and code to the runtime that automatically turns it on and off.
 - * 2002-08-20
 - Fixed SSA optimizer bug that could cause the following error message
 `x_0 has no analyze var value property`
 - * 2002-07-28
 - Added `MLton.GC.{pack,unpack}`. `pack` shrinks the heap so that other processes can use the RAM, and its dual, `unpack`, resizes the heap to the desired size.
 - * 2002-06 and 2002-07
 - Added mark compact GC.
 - Changed array layout so that arrays have three, not two header words.
 The new word is a counter word that precedes the array length and header.
 - Changed all header words to be indices into an array of object descriptors.
 - * 2002-06-27
 - Added patches from Michael Neumann to port runtime to FreeBSD 4.5.
 - * 2002-06-05
 - Output file and intermediate file are now saved in the current directory instead of in the directory containing the input file.
 - * 2002-05-31

- Fixed bug in overloading of / so that the following now type checks:
`fun f (x, y) = x + y / y`

- * 2002-04-26
 - Added back max-heap runtime option.
- * 2002-04-25
 - Fixed load/save world so that they use binary mode. This should fix the "sread failed" problem that Byron Hale saw on Cygwin that caused mlton to fail to start.
 - Added IntInf.log2.
 - Changed call to linker to use libgmp.a (if it exists) instead of libgmp.so. This is because the linker adds a dependency to a shared library even if there are no references to it
- * 2002-04-23
 - Rewrote heap resizing code. This fixed bug that was triggered with large heaps and could cause a spurious out of memory error.
 - Removed gmp from MLton sources (again :-).

 Here are the changes from version 20011006 to version 20020410.

- * 2002-03-28
 - Added BinIO.
- * 2002-03-27
 - Regularization of options
 - g --> -degug {false|true}
 - h n --> -fixed-heap n
 - p --> -profile {false|true}
- * 2002-03-22
 - Set up the stubs so that MLton can be compiled in the standard basis library, with no MLton structure. Thus it is now easy to compile MLton with an older (or newer) version of itself that has a different MLton structure.
- * 2002-03-17
 - Added MLton.Process.{spawn,spawnne,spawnp}, which use primitives when running on Cygwin and fork/exec when running on Linux.
- * 2002-02 - 2002-03
 - Added the ability to cross-compile to Cygwin/Windows.
- * 2002-02-24
 - Added gmp back for use with Cygwin.
- * 2002-02-10
 - Reworked object header words so that Array.maxLen = valOf Int.maxInt. Also fixed a long-standing minor bug in MLton, where Array.array (Array.maxLen, ...) would raise Size instead of attempting to allocate the array. It was an off-by-one error in the meaning of Array.maxLen.
- * 2002-02-08
 - Modifications to runtime to behave better in situations where the amount of live data is a signifant fraction of the amount of RAM, based on code from PolySpace. MLton executables by default can now use more than the available amount of RAM. Executables will still respect the max-heap runtime arg if it is set.

- * 2002-02-04
 - Improvements to runtime so that it fails to get space, it attempts to get less space instead of failing. Based on PolySpace's modifications.
 - Added MLton.eq.
- * 2002-02-03
 - Added MLton.IntInf.gcd.
 - Removed gmp from MLton sources. We now link with /usr/lib/gmp.a.
 - Added TextIO.getPosOut.
 - Renamed type MLton.Itimer.which to MLton.Itimer.t and MLton.Itimer.whichSignal to MLton.Itimer.signal.
 - Added -coalesce flag, for use with the C backend.
- * 2002-01-26
 - Added -show-basis-used, which prints out the parts of the basis library that the input program uses.
 - Changed several other flags (-print-at-fun-entry, -show-basis, -static) to follow the {false|true} convention.
- * 2002-01-22
 - Improved MLton.profile so that multiple profile arrays can exist simultaneously and so that the current one being used can be set from the SML side.
- * 2002-01-18
 - The MACHINE IL has been replaced with an RSSA (representation explicit SSA) IL and an improved MACHINE IL.
- * 2002-01-16
 - Added known case SSA optimization
- * 2002-01-14
 - Added rudimentary profiling control from with a MLton compile program via the MLton.Profile structure.
- * 2002-01-09
 - Fixed bug in match compiler that caused case expressions on datatypes with redundant cases to be compiled incorrectly.
- * 2002-01-08
 - Added redundant tuple construction elimination to SSA shrinker.
 - Improved flatten SSA optimization.
- * 2001-12-06
 - Changed the interface for MLton.Signal. There is no longer a separate Handler substructure. This was done so that programs that just use default and ignore signal handlers don't bring in the entire thread mechanism.
- * 2001-12-05
 - Added local ref elimination SSA optimization.
- * 2001-11-19
 - The CPS IL has been replaced with an SSA (static-single assignment) IL. All of the optimizations have been ported from CPS to SSA.
- * 2001-10-24
 - Fixed bug in Thread_atomicEnd -- limit was mistakenly set to base instead of to 0. This caused assertion failures when for executables compiled -g because GC_enter didn't reset the limit.
 - Fixed bug in register allocation of byte registers.

- * 2001-10-23
 - Added -D option to cmcat for preprocessor defines. Thanks to Anog for sending the code.
 - Changed limit check insertion so that limit checks are only coalesced within a single basic block -- not across blocks. This slows many benchmarks down, but is needed to fix a bug in the way that limit checks were coalesced across blocks. Hopefully we will figure out a better fix soon.
- * 2001-10-18
 - Fixed type inference of flexrecord so that it now follows the Definition. Many programs containing flexrecords were incorrectly rejected. Added many new tests to regression/flexrecord.sml.
 - Changed the behavior of -keep dot combined with -keep pass for SSA passes. Dot files are now saved for the program before and after, instead of just after.
- * 2001-10-11
 - Fixed a bug in the type inference that caused type variables to be mistakenly generalized. The bug was exposed in Norman Ramsey's sled.sml. Added a test to regression/flexrecord.sml to catch the problem.

Here are the changes from version 20010806 to version 20011006.

Summary:

- + Added MLton.Exn.history, which is similar to SMLofNJ.exnHistory.
 - + Support for #line directives of the form (*#line line.col "file"*).
 - + Performance improvements in native codegenerator.
 - + Bug fixes in front-end, optimizer, register allocator, Real.{maxFinite, minPos, toManExp}, and in heap save and restore.
- * 2001-10-5
 - Fixed a bug in polymorphic layered patterns, like


```
val 'a a as b = []
```

 These would always fail due to the variable "a" not being handled correctly.
 - Fixed the syntax of "val rec" so that a pattern is allowed on the left-hand side of the =. Thus, we used to reject, but now accept, the following.


```
val rec a as b as c = fn _ => ()
val rec a : unit -> unit : unit -> unit = fn () => ()
```

 Thanks again to Andreas Rossberg's test files. This is now tested for in valrec.sml.
 - Fixed dynamic semantics of "val rec" so that if "val rec" is used to override constructor status, then at run time, the Bind exception is raised as per rule 126 of the Definition. So, for example, the following program type checks and compiles, but raises Bind at run time.


```
val rec NONE = fn () => ()
val _ = NONE ()
```

 Again, this is checked in valrec.sml.
 - Added '\r\n' to ml.lex so that Windows style newlines are acceptable in input files.
 - * 2001-10-4
 - Fixed bug in the implementation of "open" declarations, which in the case of "open A B" had opened A and then looked up B in the resulting environment. The correct behaviour (see rule 22 of the Definition) is to lookup each longstrid in the current environment, and then open them all in sequence. This is now checked for in the open.sml regression test. Thanks to Andreas Rossberg for pointing this bug out.
 - Fixed bug that caused tyvars of length 1 (i.e. ') to be rejected. This is now checked in the id.sml regression test. Again, thanks to Andreas

Rossberg for the test.

- * 2001-10-2
 - Fixed bugs in Real.toManExp (which always returned the wrong result because the call to frexp was not treated as side-effecting by the optimizer) and in Real.minPos, which was zero because of a mistake with extra precision bits.
- * 2001-10-1
 - Added MLton.Exn.history.
 - Fixed register allocation bug with fucom instruction. Was allowing fucomp when the first source was not removable.
 - Changed Real.isFinite to use the C math.h finite function. This fixed the nontermination bug which occurred in any program that used Real.maxFinite.
- * 2001-9-22
 - Bug fixes found from Ramsey's lrtl in contify.fun and unused-args.fun, both of which caused compile-time exceptions to be raised.
- * 2001-9-21
 - Fixed MLton.World.{load,save} so that the saved world does not store the max heap size. Instead, the max heap size is computed upon load world in exactly the same way as at program startup. This fixes a long-standing (but only recently noticed) problem in which mlton (which uses a saved world) would attempt to use as much memory as was on the machine used to build world.mlton.
- * 2001-8-29
 - Overflow checking is now on by default in the C backend. This is a huge performance hit, but who cares, since we never use the C backend except for testing anyways.
- * 2001-8-22
 - Added support for #line directives of the form


```
(#line line.col "file"*)
```

 These directives only affect error messages produced by the parser and elaborator.
- * 2001-8-17
 - Fixed bug in removeUnused optimization that caused the following program to fail to compile.


```
fun f l = case l of [] => f l | _ :: l => f l
val _ = f [13]
```
- * 2001-8-14
 - New x86-codegen infrastructure.
 - o support for tracking liveness of stack slots and carrying them in registers across basic blocks
 - o more specific Entry and Transfer datatypes to make calling convention distinctions more explicit
 - o new heuristic for carrying values in registers across basic blocks (look Ma, no Overflows!)
 - o new "predict" model for generating register allocation hints
 - o additional bug fixes
- * 2001-8-7
 - MLton.Socket.shutdownWrite flushes the outstream.

 Here are the changes from version 20010706 to version 20010806.

Summary:

- + Word.andb (w, 0xFF) now works correctly
 - + MLton.Rusage.rusage has a patch to work around a linux kernel bug
 - + Programs of the form "exp ; program" are now accepted
 - + Added the "MLton.Rlimit" structure
 - + Added the "-keep dot" flag, which produces call graphs, intraprocedural control-flow graphs, and dominator trees
- * 2001-8-6
 - Added simple common block elimination CPS optimization.
 - * 2001-8-2
 - Took out -keep il.
 - * 2001-7-31
 - Performance improvements to TextIO.{input, output, output1}
 - * 2001-7-25
 - Added redundant-test elimination CPS optimization.
 - * 2001-7-21
 - Added common-subexpression elimination CPS optimization.
 - * 2001-7-20
 - Bug fix to x86 codegen. The commuteBinALMD peephole optimization would rewrite `mov 2,Y; add Y,Y` as `mov Y,Y; add 2,Y`. Now the appropriate interference checks are made.
 - Added intraprocedural unused argument removal.
 - Added intraprocedural flattener. This avoids some stupid tuple allocations in loops. Decent speedup on a few benchmarks (count-graphs, psdes-random, wc-scanStream) and no noticeable slowdowns.
 - Added -keep dot flag.
 - * 2001-7-17
 - Modified grammar to properly handle `val rec`. There were several problems.
 - o MLton had accepted "`val rec 'a ...`" instead of "`val 'a rec ...`"
 - o MLton had not accepted "`val x = 13 and rec f = fn () => ()`"
 - o MLton had not accepted "`val rec rec f = fn () => ()`"
 - o MLton had not accepted "`val rec f = fn () => () and rec g = fn () => ()`"
 - * 2001-7-16
 - Workaround for Linux kernel bug that can cause `getrusage` to return a wrong system time value (low by one second). See `fixedGetrusage` in `gc.c`.
 - Bug fix to x86 codegen. The register allocator could get confused when doing comparisons of floating point numbers and use the wrong operand. The bug seems to have never been detected because it only happens when both of the operands are already on the floating point stack, which is rare, since one is almost always in memory since we don't carry floating point values in the stack across basic blocks.
 - Added production to the grammar on page 58 of the Definition that had been missing from MLton since day one.


```

          program ::= exp ; <program>
          
```

 Also updated docs to reflect change.
 - Modified grammar to accept the empty program.
 - Added `-type-check expert` flag to turn on type checking in ILs.
 - * 2001-7-15
 - Bug fix to the algebraic simplifier. It had been rewriting `Word32.andb (w, 0xFF)` to `w` instead of `Word32.andb (w, 0xFFFFFFFF)` to `w`.

- * 2001-7-13
 - Improved CPS shrinker so that if-tests where the then and else branch jump to the same label is turned into a direct jump.
 - Improved CPS shrinker (Prim.apply) to handle constructors
 - A = A --> true
 - A = B --> false
 - A x = B y --> false
 - Rewrote a lot of loops in the basis library to use inequalities instead of equality for the loop termination test so that the (forthcoming) overflow detection elimination will work on the loop index variable.
- * 2001-7-11
 - Fixed minor bugs in Array2.{array,tabulate}, Substring.{slice} that caused the Overflow exception to be raised instead of Size or Subscript
 - Fixed bug in Pack32Big.update that caused the wrong location to be updated.
 - Fixed several bugs in Pack32{Big, Little}.{subArr, subVec, update} that caused Overflow to be raised instead of Subscript. Also, improved the implementation so that bounds checking only occurs once per call (instead of four times, which was sometimes happening).
 - Fixed bugs in Time.{toMilliseconds, toMicroseconds} that could cause a spurious Overflow exception.
 - Fixed bugs in Time.{fromMilliseconds, fromMicroseconds} that could cause a spurious Time exception.
 - Improved Pack32.sub* by reordering the orbs.
 - Improved {Int, IntInf}.mod to increase chances of constant folding.
 - Switched many uses of +, -, * in basis library to the non-overflow checked versions. Modules changed were: Array, Array2, Byte, Char, Int, IntInf, List, Pack32{Big, Little}, Util, String, StringCvt, Substring, TextIO, Time, Vector.
 - Added regression tests for Array2, Int (overflow checking), Pack32, Substring, Time.
 - Changed CPS output so that it includes a dot graph for each CPS function.
- * 2001-7-9
 - Change OS.Process.exit so that it raises an exception if the exit status is not in [0, 256).
 - Added MLton.Rlimit to provide access to getrlimit and setrlimit.

Here are the changes from the 20000906 version to the 20010706 version.

Summary:

- + Native X86 code generator (instead of using gcc)
- + Significantly improved compile times
- + Significantly improved run times for generated executables
- + Many bug fixes
- + Correct raising of the Overflow exception for integer arithmetic
- + New modules in the MLton structure

- * 2001-7-6
 - GC mods from Henry. Mostly adding inline declarations.
- * 2001-7-5
 - Fixed several runtime bugs involving threads, critical sections, and signals.
- * 2001-6-29
 - Fixed performance bug in cps/two-point-lattice.fun that caused quadratic behavior. This affects the raise-to-jump and useless analyses. In particular, the useless analysis was blowing up when compiling fxp.

- * 2001-6-27
 - Henry improved wordAlign -- this sped up GC by 27% (during a self compile).

- * 2001-6-20
 - Moved MLton.random to MLton.Random.rand and added other stuff to MLton.Random
 - Added MLton.TextIO.mkstemp.
 - Made Int.{div,quot} respect the -detect-overflow switch.

- * 2001-6-20
 - Added MLton.Syslog.

- * 2001-6-7
 - Fixed bug in MLton.Socket.accept that was in the runtime implementation Socket_accept. It did a setsockopt SO_REUSEADDR after the accept. It should have been after the call to socket in Socket_listen. Thanks to Doug Bagley for the fix.

- * 2001-5-30
 - Fixed bug in remove-unused that caused polymorphic equality to return true sometimes when constructors were never used in a pattern match. For example, the following (in which A and B are not used as patterns):

```
datatype t = A | B
datatype u = C of t
val _ = if C A = C B then raise Fail "bug" else ()
```

- * 2001-3-27
 - Fixed bug that caused all of the following to fail
{LargeWord,Word,SysWord}.{toLargeInt,toLargeIntX,fromLargeInt}
The problem was the basis library file integer/patch.sml which fixed Word32 but not the other structures that are the same.

- * 2001-2-12
 - Fixed bug in match compiler that caused it to spend a lot of extra time in deep patterns. It still could be exponential however. Hopefully this will get fixed in the release after next.
This bug could cause very slow compile times in some cases.
Anyways, this fix cut the "finish infer" time of a self compile down from 22 to under 4 seconds. I.E. most of the time used to be spent due to this bug.

- * 2001-2-6
 - Fixed bug in frontend that caused the wrong file and line number to be reported with errors in functor bodys.

- * 2001-1-3 - 2000-2-5
 - Changes to CoreML, XML, SXML, and CPS ILs to replace lists by vectors in order to decrease space usage.

- * 2001-1-16
 - Fixed a bug in constant propagation where the length of vectors was not propagated properly.

- * 2000-12-11 - 2001-1-3
 - Major rewrite of elaborator to use a single hash table for each namespace instead of a hash table for every environment.

- * 2000-12-20
 - Fixed some bugs in the SML/NJ compatibility library,
src/lib/mlton-subs-in-smlnj.

- * 2000-12-8

- More careful removal of tracing code when compiling `MLton_debug=0`. This cut down self compile data size by 100k and compile time by a few seconds.
- Added built in character and word cases propagated throughout all ILs.
- * 2000-12-6
 - Added max stack size information to `gc-summary`.
- * 2000-12-5
 - Added `src/benchmark`, which contains an SML program that benchmarks all of the SML compilers I have my hands on. The script has lots of hardwired paths for now.
- * 2000-12-4
 - Fixed bug in `Posix.ProcEnv.environ`, which did not work correctly in a saved world (the original `environ` was saved). In fact, it did not work at all because the ML primitive expected a constant and the C was a nullary function. This caused a segfault with any program using `Posix.ProcEnv.environ`.
 - Added `MLton.ProcEnv.setenv`, since there doesn't seem to be any `setenv` in the basis library.
- * 2000-11-29
 - Changed backend so that it should no longer generate machine programs with void operands.
 - Added `-detect-overflow` and `-safe` flags.
- * 2000-11-27 - 2000-11-28
 - Changes in many places to use `List.revMap` instead of `List.map` to cut down on allocation.
- * 2000-11-21
 - Added `MLton.Word.~` and `MLton.Word8.~` to the MLton basis library structure.
- * 2000-11-20
 - Fixed a bug in the CPS shrinker that could cause a compile-time failure. It was maintaining occurrence counts incorrectly.
- * 2000-11-15
 - Fixed a (performance) bug in constant propagation that caused the hashing to be bad.
 - Improved translation to XML so that the match compiler isn't called on tuple or if expressions. This should speed up the translation and make the output smaller.
 - Fixed a bug in the match compiler that caused it to not generate integer case statements. This should speed up the `mlyacc` benchmark and the MLton front end.
- * 2000-11-9
 - Added `IntInf_equal` and `IntInf_compare` primitives.
 - Took out the automatic `-keep c` when compiling `-g`.
- * 2000-11-8
 - Added a whole bunch of algebraic laws to the CPS shrinker, including some specifically targeted to `IntInf` primitives.
- * 2000-11-3
 - Improved implementation of properties so that sets don't allocate.
 - Improved implementation of type homomorphism in type inference. What was there before appears to have been a bug -- it didn't use the property on types.

- * 2000-11-2
 - Fixed timers used with `-v` option to use user + sys time.

- * 2000-10-27
 - Split the runtime basis library C files into many separate files so that only the needed code would be included by the linker.
 - Fixed several bugs in the front end grammar and elaborator that caused type specifications to be handled incorrectly. The following three programs used to be handled incorrectly, but are now handled correctly.
 - signature S = sig type t and u = int end (* reject *)
 - signature S = sig type t = int and u = t end (* accept *)
 - signature S = sig eqtype t and u = int end (* reject *)

- * 2000-10-25
 - Changes to main.sml to run complete compiles with `-native` switch.

- * 2000-10-24
 - Removed defunctorizer.

- * 2000-10-20
 - Fixed bug in `cps-tree.fun PrimExp.maySideEffect`. This bug could cause no operand failures in the backend.
 - Fixed bug in the runtime implementation of `MLton.size`. The size for stack objects was using the used instead of reserved, and so was too low.

- * 2000-10-19
 - Replaced automatically generated dependencies in `src/runtime/Makefile` with hand generated ones. Took out `make depend` from `src/Makefile`. `makedepend` was behaving really badly on RHAT 7.0.
 - Tweaked compiler to shorten width of C output lines to work around bug in RHAT 7.0 `cpp` which silently truncates (very) long lines.
 - Fixed bug in grammar that didn't allow "op" to occur in datatype and exception bindings, causing the following to fail
 - datatype t = op T
 - exception op E = op Fail
 - Improved error messages in CM processor. Fixed bug in CM Alias handling.

- * 2000-10-18
 - Fixed two bugs in the gc that did comparisons with `(s->limit - s->frontier)`, which of course doesn't work if `frontier` is beyond `limit`, since these are unsigned. This could have caused segfaults, except that the mutator checks the `frontier` upon return from the GC.

- * 2000-10-17
 - Fixed bug in backend in the calculation of `maxFrameSize`. It could be wrong (low) in some situations.
 - Improved CPS inliner's estimate of function sizes. The size of a function now takes into account other inlined functions that the function calls. This also changed the meaning of the `size` argument to the `-inline` switch. It now corresponds (roughly) to the product of the size of the function and the number of calls. In general, it should be larger than before.

- * 2000-10-13
 - Made some calls to `Array.sub unsafe` in the implementation of `Array2`.
 - Integrated Matthew's new x86 backend with floating point support.

- * 2000-10-9
 - Fixed CM file processor so that MLton works if it is run from a different directory than the main CM file.

- * 2000-10-4
 - Changed `LimitCheck` so it loops on the `frontier > limit` check. This fixed

a potential bug in threads caused when there is enough space available for a thread, `t`, before switching to another thread but not enough space when it resumes. This could have caused a segfault.

- * 2000-10-3
 - More rewrites of `TextIO.StreamIO` to improve speed.
 - Changed `TextIO` so that only `TextIO.stdErr` is unbuffered.
 - Changed `TextIO` so that FIFOs and sockets are buffered.
- * 2000-10-2
 - Combined `remove-unused-constructors`, `remove-unused-functions`, and `remove-unused-globals` into a single pass that runs to fixed-point and produces results at least as good as running the previous three in (any) sequence.
- * 2000-9-29
 - Added `GC_FIRST_CHECK`, which does a gc at each limit check the first time it reached.
 - Reimplemented `TextIO.StreamIO` (from 2000-9-12) to use lists of strings instead of lists of characters so that the per char space overhead is small.
- * 2000-9-21
 - Fixed bug in profiling labels in C code. The label was always the basic block label instead of the cps function label.
 - Added `-b` switch to `mlprof` to gather data at the basic block level.
 - Improved performance of `TextIO.input1` by about 3X.
- * 2000-9-15 - 2000-9-19
 - Added overflow exceptions to CPS and Machine ILs.
- * 2000-9-12
 - Fixed `TextIO.scanStream`. It was very broken.
 - Added `TextIO.{getInstream,mkInstream,setInstream}`
`TextIO.StreamIO.{canInput,closeIn,endOfStream,input1,input,inputAll,`
`inputLine,inputN}`
- * 2000-9-11
 - Fixed `Real_qequal` in `mlton-lib.h`. It was missing a paren that caused code using it to not even compile. It was also semantically incorrect.
 - Noted that `Real_{equal,lt,le,gt,ge}` may not follow basis library spec, since ANSI does not require IEEE compliance, and hence these could return wrong results when nans are involved.

 Here are the changes from the 20000712 version to the 20000906 version.

Version 20000906 is mostly a bugfix release over 20000712. The other major changes are that `mllex` and `mlyacc` are now included and that `mlton` can now process a limited subset of CM files as input.

- * 2000-9-6
 - Fixed `Socket_listen` in `mlton-lib.c` so that it closes the socket if the `bind`, `listen`, or `getsockname` fails. This could have caused a file descriptor leak.
- * 2000-9-5
 - Added `-static` commandline switch.
 - Changed default max heap size to .85 RAM from .95 RAM.
 - Added `PackRealLittle` structure to basis library.
- * 2000-8-25

- Added cases on integers to ILs (instead of using sequences of tests) so that backend can emit more efficient test (jump table, binary tree, ...).
- * 2000-8-24
 - Fixed bug in gc.c. dfsInitializeStack would smummap a NULL pointer whenever toSpace was NULL. This could gauge MLton.size to segfault.
 - Fixed bug in Popt that caused -k to fail with no keeps.
- * 2000-8-22 - 2000-8-23
 - Ported mlex and mlyacc from SML/NJ
- * 2000-8-20 - 2000-8-21
 - Added ability to use a .cm file as input to MLton.
- * 2000-8-16
 - Ported mlprof to SML.
 - Fixed bug in library/basic/assert.sml that caused asserts to be run even when MLton.debug = false.
- * 2000-8-15
 - Fixed bug in backend -- computation of maxFrameSize was wrong. It didn't count slots in frames that didn't make nontail calls. This could lead to the stack being overwritten because a stack limit check didn't guarantee enough space, and lead to a seg fault.
 - Fixed bug in gc.c newThreadOfSize. If the thread allocation caused a gc, then the stack wasn't forwarded, leading to a seg fault. The solution was to ensure enough memory all at once, and then fill in both objects.
- * 2000-8-14
 - Changed limit checks so that checks < 512 bytes are replaced by a check for 0. The runtime also moves the limit down by 512. This is done so that the common case, a small limit check, has less code and is faster.
 - Fixed bug in cps/cps-tree.fun Program.hasPrim returned true for any program that had *any* primapp, not just programs satisfying the predicate. This caused cps/once.fun to be overly conservative, since it thought that every program used continuations.
- * 2000-8-10
 - Fixed bug in CPS typechecker. It didn't enforce that handlers should be defined before any reference to them -- including implicit references in HandlerPops. This caused an evil bug in the liveness analysis where a variable that was only live in the handler was missed in a continuation because the liveness for the handler wasn't computed yet.
 - Limited the size for moving up limit checks for arrays whose size is known at compile time to avoid huge limit checks getting moved into loops.
 - added -indent, -kp, -show-types switches.
 - Put optimization in CPS IL suggested by Neal Glew. It determines for each toplevel function if it can raise an exception to its caller. Also, it removes HanderPush and HandlerPop for handlers that are not on top of the stack for any nontail call.
- * 2000-8-8
 - Changed register allocator so that continuation formals can be allocated in pseudo registers -- they aren't necessarily forced to the stack.
- * 2000-8-3
 - Fixed bug in constant folding.
Word8.>> had been used to implement Word8.<~>>.
 - Fixed bug in allocate registers that was not forcing the size argument to Primitive.Array.array to be a stack slot. This could cause problems if there was a thread switch in the limit check, since upon return the size pseudo register would have a bogus value.

- * 2000-8-1
 - Turned back on Xml simplification after monomorphisation.
- * 2000-7-31
 - Fixed bug in MLton.Itimer.set that caused the time to be doubled.
 - Fixed bug in MLton.Thread that made it look like asynchronous exceptions were allowed by throw'ing an exception raising thunk to an interrupted thread obtained via a signal handler. Attempting asynchronous exceptions will now cause process death, with a helpful error message.
- * 2000-7-27
 - Updated docs to include structure World: MLTON_WORLD in MLton structure.
 - Added toplevel signatures MLTON_{CONT, ..., WORLD} to basis library.
 - Fixed broken link in docs to CM in cmcat section.
- * 2000-7-26
 - Eliminated GC_switchToThread and Thread_switchTo1, since the inlined version Thread_switchTo is all that's needed, and Matt's X86 backend now handles it.
 - Added MLton.Signal.vtalm, needed for Itimer.Set{which = Itimer.Virtual, ...}
- * 2000-7-25
 - Added MLton.Socket.shutdownWrite.
- * 2000-7-21
 - Updated mlton-lib.c MLton_bug with new email (MLton@sourcelight.com).
- * 2000-7-19
 - Fixed Posix.Process.kill to check for errors.
- * 2000-7-18
 - Fixed the following Posix.ProcEnv functions to check for errors:
 - setgid, setpgid, setsid, setuid.
 - Fixed doc/examples/callcc.sml.

Here are the changes from the 1999-7-12 to the 20000712 version.

- * 2000-6-10 - 2000-7-12
 - Too many changes to count: bug fixes, new basis library modules, optimizer improvements.
- * 2000-6-30
 - Fixed bug in monomorphiser that caused programs with non-value carrying exception declarations in polymorphic functions to have a compile-time error because of a duplicate label. The problem was that the exception constructor wasn't duplicated.
- * 2000-5-22 - 2000-6-10
 - Finished the changes for the new Cps IL.
- * 2000-1-1
 - Fixed some errors in the basis library.
 - Real.copySign
 - Posix.FileSys.fpathconf
 - Posix.IO.{lseek, getlk, setlk, setlkw}
 - Posix.ProcEnv.setpgid
 - Posix.TTY.getattr
 - System.FileSys.realpath
- * 1999-12-22
 - Fixed bug in src/closure-convert/abstract-value.fun that caused a compiler failure whenever a program had a vector where the element

```
type contained an ->.
* 1999-12-10
- Changed dead code elimination in core-ml/dead-code.fun so that
  wildcard declarations (val _ = ...) in the basis are kept.
  Changed places in the basis library to take advantage of this
- Added setTopLevelHandler primitive so that the basis library code
  can define the topLevel handler.
- Changed basis-library/misc/suffix.sml to call OS.Process.exit.
  Took out Halt transfer from Cps, since the program never should
  reach it.
- Cleaned up basis-library/system/{process.sml, unix.sml} to use
  the new signal handling stuff.
* 1999-11-28 - 1999-12-20
Added support for threads and cleaned up signal handling.
This involved a number of changes.
- The stack is now allocated as just another kind of heap object.
- Limit checks are inserted at all loop headers, whether or not there
  is any allocation. This is to ensure that the signal handler
  always has a chance to get called.
- The register allocator puts more variables in stack slots. The new
  rule is that a variable goes in a stack slot if it is ever live
  across a nontail call, in a handler, or (this is the new part)
  across a limit check.
- Arguments are passed on the stack, with the convention determined by
  argument types.
- The "locals" array of pointers that was copied to/from for GC is now
  gone, because no registers (in particular no pointer valued
  registers) can be live at a limit check point.
* 1999-11-21
- Runtime system
  o Fixed a bug introduced by the signal code
    (presumably on 1999-8-9) that caused a gc to *not* be performed when
    doing a save world. This caused the heaps created by save world to
    be the same size as the heap -- not the live data. This was quite
    bad.
  o Cleaned up the Makefile. Add make depend.
  o Added max gc pause to gc-summary info.
  o Move heap translation variables that had been file statics into
    the GC_state.
- Made structure Position available at toplevel.
- Basis Library
  o Added MLton.loadWorld
- Added Primitive.usesCallcc
- Added Primitive.safe
- Removed special size functions from cps/save-world -- they are no
  longer necessary since size doesn't do a gc.
- Fixed another (sigh) bug in cps/simplify-types.fun that could
  cause it to not terminate.
* 1999-11-16
- Cleaned up backend/machine.fun a bit so that it spits out macros
  for allocation of objects and bumping of frontier. Added macros
  MLTON_object and MLTON_incFrontier to include/mlton-lib.h.
- Fixed a bug in backend/limit-check.fun that caused loops to not be
  detected if they were only reached by a case branch. This could
  cause there to be loop that allocates with no limit check.
  Needless to say, this could cause a segfault if the loop ran for
  long enough.
* 1999-10-18
Added basis library function Array2.copy.
* 1999-8-15
Turned off globalization of ref cells
(closure-convert/globalize.fun) because it interacts badly with
```

```

serialization.
* 1999-8-13
Fixed bug in mlton-lib.h in MLTON_allocArrayNoPointers that was
triggered when bytesPerElt == 0. The problem was that it wasn't
reserving space for the forwarding pointer. This could cause a seg
fault.
* 1999-8-8 and 1999-8-9
Added support for signal handling.
* 1999-8-7
Fixed bugs in Array.tabulate (and other tabulate variants) caused if
the function argument used callcc.
* 1999-8-1
Added serialization, which was mostly code in src/runtime/gc.c.
GC_serialize converts an object to a Word8Vector.vector.
GC_deserialize undoes the conversion.
(de)Serialization should work for all objects except for functions,
because I haven't yet added the support in the flow analysis.
* 1999-7-31
- Cleaned up the GC. Changed headers, by stealing a bit from the
number of non pointers and making it a mark bit (used in GC_size).
- Rewrote GC_size so that it runs in time proportional to the number
of pointers in the object. It does a depth-first-search now,
using toSpace to hold the stack.
* 1999-7-30
- Fixed bug in SUBSTRING. getc had the wrong type. This bug wasn't
noticed because MLton doesn't do enough type checking.
- Fixed bug (seg fault) caused when a GC immediately followed a throw.
* 1999-7-29
Fixed bug in Date.fmt (basis-library/system/date.sml). It was not
setting Tm.buf, and hence the time was always 0 unless there had
been a previous call to setTmBuf.
* 1999-7-28
- Fixed bugs in Posix.IO.FLock.{getlk,setlk,setlkw}, which would
cause compilation to fail because FLock.toInt was defined as the C
castInt, which no longer exists. Instead, expand FLock.toInt to
MLTON_pointerToInt, which was added to include/mlton-lib.h.
- Changed Posix.Primitive.Flock to Posix.Primitive.FLock.
- Added MLTON_chown, MLTON_ftruncate to include/mlton-posix.h. They were
missing. This would cause compilation of any program using
Posix.FileSys.{chown,ftruncate} to fail. Also made it so all of the
primitives in basis-library/posix/primitive.sml use MLTON_ versions
of functions, even if a wrapper is unnecessary.
* 1999-7-25
Added some other missing signature definitions to toplevel.
* 1999-7-24
Added missing OS_* signature definitions to
basis-library/top-level/top-level.sml.
* 1999-7-19
Fixed bug in basis-library/arrays-and-vectors/mono-array.sml. Used
:> instead of : so that the monomorphic array types are abstract.
-----

```

Here are the changes from the 1999-3-19 version to the 1999-7-12 version.

```

* 1999-7-12
Changed src/backend/machine.fun so that the 'pointer locals' array is
only as large as necessary in order to copy all pointer-valued
locals, not as large as the number of pointer-valued locals.
* 1999-7-11
Rewrote src/backend/allocate-registers.fun so that it does a better
job of sharing "registers" (i.e. C local variables) and stack slots.

```

This should cut down on the amount of copying that has to happen before and after a gc. It should also cut down on the size of stack slots.

- * 1999-7-10
Fixed a bug in src/backend/parallel-move.fun that should have been triggered on most any parallel move. I guess parallel moves almost never happened due to the old register allocation strategy -- but, with the new one (see note for 1999-7-12) parallel moves will be frequent.
- * 1999-6-27
Fixed src/main.sml so that when compiling -p, the .c file is compiled -g and the .o is linked -p.
In bakend/machine.fun, added profiling comments before chunkswitches and put in an optimization to avoid printing repeated profiling comments. Also, profiling comments are only output when compiling -p.
- * 1999-6-17
Changed -i to -inline, -f to -flatten, -np to -no-polyvariance, -u to -unsafe.
Added -i, -I, -l, -L flags for includes and libraries.
Updated documentation for these options and for ffi.
- * 1999-6-16
Hardwired version number in src/control/control.sml. As it stood, the version number was computed when MLton was built after someone downloaded it, which was clearly wrong.
- * 1999-6-16
Fixed undefined variable time in GC_done in src/runtime/gc.c.
- * 1999-6-8
in include/mlton-lib.h,
removed #include <huge_val.h>
added #include <math.h>
and deleted all of the function signatures I had copied from math.h
Changed Real.{minNormalPos, minPos, maxFinite} so that they are computed in real.sml instead of appearing as constants in the C.
- * 1999-6-7
IntInf.pow added to basis library.
- * 1999-6-4
bin/mlton changed to use .arch-n-opsys if it exists.
- * 1999-6-3
src/Makefile changed to use sml-cm instead of sml
- * 1999-5-10
Patch to src/atoms/small-int-inf.fun to work around a bug in the SML/NJ implementation of bignums. This bug was causing some hex bignum constants to be lexed incorrectly.
- * 1999-4-15
Comments emitted in C code for profiling. The comments identify the Cps function responsible for each C statement.
- * 1999-4-15
callcc and throw added.
- * 1999-4-15
Bug in src/cps/simplify-types fixed. The bug caused nontermination whenever there was a circular datatype with a vector on the rhs.
E.g. datatype t = T of t vector

Here are the changes from the 1998-8-26 version to the 1999-3-19 version.

- * Compile time and code size have decreased.
- * Runtime performance of executables has improved.
- * Large programs can now be compiled.
- * MLton is self hosting.

- * The basis library is mostly complete and many bugs have been fixed.
 - * The monomorphiser (-m) is no longer available.
 - * The heap and stack are automatically resized.
 - * There are now facilities for heap checkpointing (MLton.saveWorld) and object size computation (MLton.size).
 - * MLton uses the GNU multiprecision library to provide a fast implementation of IntInf.
-

21 ChrisClearwater

22 Chunkify

[Chunkify](#) is an analysis pass for the [RSSA IntermediateLanguage](#), invoked from [ToMachine](#).

22.1 Description

It partitions all the labels (function and block) in an [RSSA](#) program into disjoint sets, referred to as chunks.

22.2 Implementation

- [chunkify.sig](#)
- [chunkify.fun](#)

22.3 Details and Notes

Breaking large [RSSA](#) functions into chunks is necessary for reasonable `gcc` compile times with the [CCodegen](#).

23 CKitLibrary

The **ckit Library** is a C front end written in SML that translates C source code (after preprocessing) into abstract syntax represented as a set of SML datatypes. The ckit Library is distributed with SML/NJ. Due to differences between SML/NJ and MLton, this library will not work out-of-the box with MLton.

As of 20130706, MLton includes a port of the ckit Library synchronized with SML/NJ version 110.76.

23.1 Usage

- You can import the ckit Library into an MLB file with:

MLB file	Description
<code>\$(SML_LIB)/ckit-lib/ckit-lib.mlb</code>	

- If you are porting a project from SML/NJ's [CompilationManager](#) to MLton's [ML Basis system](#) using `cm2mlb`, note that the following map is included by default:

```
# ckit Library
$ckit-lib.cm                $(SML_LIB)/ckit-lib
$ckit-lib.cm/ckit-lib.cm    $(SML_LIB)/ckit-lib/ckit-lib.mlb
```

This will automatically convert a `$/ckit-lib.cm` import in an input `.cm` file into a `$(SML_LIB)/ckit-lib/ckit-lib.mlb` import in the output `.mlb` file.

23.2 Details

The following changes were made to the ckit Library, in addition to deriving the `.mlb` file from the `.cm` file:

- `ast/ast-sig.sml` (modified): Rewrote use of `withtype` in signature.
- `ast/build-ast.sml` (modified): Rewrote use of `or-patterns`.
- `ast/initializer-normalizer.sml` (modified): Rewrote use of `or-patterns`.
- `ast/pp/pp-ast-adornment-sig.sml` (modified): Rewrote use of signature in local.
- `ast/pp/pp-ast-ext-sig.sml` (modified): Rewrote use of signature in local.
- `ast/pp/pp-lib.sml` (modified): Rewrote use of `or-patterns`.
- `ast/sizeof.sml` (modified): Rewrote use of `or-patterns`.
- `ast/type-util-sig.sml` (modified): Rewrote use of signature in local.
- `ast/type-util.sml` (modified): Rewrote use of `or-patterns`.
- `parser/grammar/c.lex.sml` (modified): Rewrote use of vector literal.
- `parser/parse-tree-sig.sml` (modified): Rewrote use of (sequential) `withtype` in signature.
- `parser/parse-tree.sml` (modified): Rewrote use of (sequential) `withtype`.

23.3 Patch

- `ckit.patch`

24 Closure

A closure is a data structure that is the run-time representation of a function.

24.1 Typical Implementation

In a typical implementation, a closure consists of a *code pointer* (indicating what the function does) and an *environment* containing the values of the free variables of the function. For example, in the expression

```
let
  val x = 5
in
  fn y => x + y
end
```

the closure for `fn y => x + y` contains a pointer to a piece of code that knows to take its argument and add the value of `x` to it, plus the environment recording the value of `x` as 5.

To call a function, the code pointer is extracted and jumped to, passing in some agreed upon location the environment and the argument.

24.2 MLton's Implementation

MLton does not implement closures traditionally. Instead, based on whole-program higher-order control-flow analysis, MLton represents a function as an element of a sum type, where the variant indicates which function it is and carries the free variables as arguments. See [ClosureConvert](#) and [CejinEtAl00](#) for details.

25 ClosureConvert

[ClosureConvert](#) is a translation pass from the [SXML IntermediateLanguage](#) to the [SSA IntermediateLanguage](#).

25.1 Description

It converts an [SXML](#) program into an [SSA](#) program.

[Defunctionalization](#) is the technique used to eliminate [Closures](#) (see [CejtinEtAl00](#)).

Uses [Globalize](#) and [LambdaFree](#) analyses.

25.2 Implementation

- [closure-convert.sig](#)
- [closure-convert.fun](#)

25.3 Details and Notes

26 CMinusMinus

C-- is a portable assembly language intended to make it easy for compilers for different high-level languages to share the same backend. An experimental version of MLton has been made to generate C--.

- <http://www.mlton.org/pipermail/mlton/2005-March/026850.html>

26.1 Also see

- [LLVM](#)
-

27 CombineConversions

`CombineConversions` is an optimization pass for the [SSA IntermediateLanguage](#), invoked from `SSASimplify`.

27.1 Description

This pass looks for and simplifies nested calls to (signed) extension/truncation.

27.2 Implementation

- `combine-conversions.fun`

27.3 Details and Notes

It processes each block in dfs order (visiting definitions before uses):

- If the statement is not a `PrimApp` with `Word_extdToWord`, skip it.
- After processing a conversion, it tags the `Var` for subsequent use.
- When inspecting a conversion, check if the `Var` operand is also the result of a conversion. If it is, try to combine the two operations. Repeatedly simplify until hitting either a non-conversion `Var` or a case where the conversion cannot be simplified.

The optimization rules are very simple:

```
x1 = ...
x2 = Word_extdToWord (W1, W2, {signed=s1}) x1
x3 = Word_extdToWord (W2, W3, {signed=s2}) x2
```

- If $W1 = W2$, then there is no conversions before x_1 .
This is guaranteed because $W2 = W3$ will always trigger optimization.

- Case $W1 \leq W3 \leq W2$:

```
x3 = Word_extdToWord (W1, W3, {signed=s1}) x1
```

- Case $W1 < W2 < W3$ AND $((\text{NOT } s1) \text{ OR } s2)$:

```
x3 = Word_extdToWord (W1, W3, {signed=s1}) x1
```

- Case $W1 = W2 < W3$:

unoptimized, because there are no conversions past $W1$ and $x2 = x1$

- Case $W3 \leq W2 \leq W1$ OR $W3 \leq W1 \leq W2$:

```
x_3 = Word_extdToWord (W1, W3, {signed=_}) x1
```

because $W3 \leq W1$ && $W3 \leq W2$, just clip $x1$

- Case $W2 < W1 \leq W3$ OR $W2 < W3 \leq W1$:

unoptimized, because $W2 < W1$ && $W2 < W3$, has truncation effect

- Case $W1 < W2 < W3$ AND $(s1 \text{ AND } (\text{NOT } s2))$:

unoptimized, because each conversion affects the result separately

28 CommonArg

`CommonArg` is an optimization pass for the [SSA IntermediateLanguage](#), invoked from `SSASimplify`.

28.1 Description

It optimizes instances of `Goto` transfers that pass the same arguments to the same label; e.g.

```
L_1 ()
  ...
  z1 = ?
  ...
  L_3 (x, y, z1)
L_2 ()
  ...
  z2 = ?
  ...
  L_3 (x, y, z2)
L_3 (a, b, c)
  ...
```

This code can be simplified to:

```
L_1 ()
  ...
  z1 = ?
  ...
  L_3 (z1)
L_2 ()
  ...
  z2 = ?
  ...
  L_3 (z2)
L_3 (c)
  a = x
  b = y
```

which saves a number of resources: time of setting up the arguments for the jump to `L_3`, space (either stack or pseudo-registers) for the arguments of `L_3`, etc. It may also expose some other optimizations, if more information is known about `x` or `y`.

28.2 Implementation

- [common-arg.fun](#)

28.3 Details and Notes

Three analyses were originally proposed to drive the optimization transformation. Only the *Dominator Analysis* is currently implemented. (Implementations of the other analyses are available in the [repository history](#).)

28.3.1 Syntactic Analysis

The simplest analysis I could think of maintains

```
varInfo: Var.t -> Var.t option list ref
```

initialized to `[]`.

- For each variable v bound in a `Statement.t` or in the `Function.t` args, then `List.push(varInfo v, NONE)`.
- For each `L (x1, ..., xn) transfer` where (a_1, \dots, a_n) are the formals of `L`, then `List.push(varInfo ai, SOME xi)`.
- For each block argument a used in an unknown context (e.g., arguments of blocks used as continuations, handlers, arith success, runtime return, or case switch labels), then `List.push(varInfo a, NONE)`.

Now, any block argument a such that `varInfo a = xs`, where all of the elements of `xs` are equal to `SOME x`, can be optimized by setting $a = x$ at the beginning of the block and dropping the argument from `Goto` transfers.

That takes care of the example above. We can clearly do slightly better, by changing the transformation criteria to the following: any block argument a such that `varInfo a = xs`, where all of the elements of `xs` are equal to `SOME x` or are equal to `SOME a`, can be optimized by setting $a = x$ at the beginning of the block and dropping the argument from `Goto` transfers. This optimizes a case like:

```
L_1 ()
  ... z1 = ? ...
  L_3 (x, y, z1)
L_2 ()
  ... z2 = ? ...
  L_3 (x, y, z2)
L_3 (a, b, c)
  ... w = ? ...
  case w of
    true => L_4 | false => L_5
L_4 ()
  ...
  L_3 (a, b, w)
L_5 ()
  ...
```

where a common argument is passed to a loop (and is invariant through the loop). Of course, the [LoopInvariant](#) optimization pass would normally introduce a local loop and essentially reduce this to the first example, but I have seen this in practice, which suggests that some optimizations after [LoopInvariant](#) do enough simplifications to introduce (new) loop invariant arguments.

28.3.2 Fixpoint Analysis

However, the above analysis and transformation doesn't cover the cases where eliminating one common argument exposes the opportunity to eliminate other common arguments. For example:

```
L_1 ()
  ...
  L_3 (x)
L_2 ()
  ...
  L_3 (x)
L_3 (a)
  ...
  L_5 (a)
L_4 ()
  ...
  L_5 (x)
L_5 (b)
  ...
```

One pass of analysis and transformation would eliminate the argument to `L_3` and rewrite the `L_5 (a)` transfer to `L_5 (x)`, thereby exposing the opportunity to eliminate the common argument to `L_5`.

The interdependency the arguments to `L_3` and `L_5` suggest performing some sort of fixed-point analysis. This analysis is relatively simple; maintain

```
varInfo: Var.t -> VarLattice.t
```

where

```
VarLattice.t ~== Bot | Point of Var.t | Top
```

(but is implemented by the [FlatLattice](#) functor with a `lessThan` list and value `ref` under the hood), initialized to `Bot`.

- For each variable `v` bound in a `Statement.t` or in the `Function.t` args, then `VarLattice.<= (Point v, varInfo v)`
- For each `L (x1, ..., xn)` transfer where `(a1, ..., an)` are the formals of `L`, then `VarLattice.<= (varInfo xi, varInfo ai)`.
- For each block argument `a` used in an unknown context, then `VarLattice.<= (Point a, varInfo a)`.

Now, any block argument `a` such that `varInfo a = Point x` can be optimized by setting `a = x` at the beginning of the block and dropping the argument from `Goto` transfers.

Now, with the last example, we introduce the ordering constraints:

```
varInfo x <= varInfo a
varInfo a <= varInfo b
varInfo x <= varInfo b
```

Assuming that `varInfo x = Point x`, then we get `varInfo a = Point x` and `varInfo b = Point x`, and we optimize the example as desired.

But, that is a rather weak assumption. It's quite possible for `varInfo x = Top`. For example, consider:

```
G_1 ()
  ... n = 1 ...
  L_0 (n)
G_2 ()
  ... m = 2 ...
  L_0 (m)
L_0 (x)
  ...
L_1 ()
  ...
  L_3 (x)
L_2 ()
  ...
  L_3 (x)
L_3 (a)
  ...
  L_5 (a)
L_4 ()
  ...
  L_5 (x)
L_5 (b)
  ...
```

Now `varInfo x = varInfo a = varInfo b = Top`. What went wrong here? When `varInfo x` went to `Top`, it got propagated all the way through to `a` and `b`, and prevented the elimination of any common arguments. What we'd like to do instead is when `varInfo x` goes to `Top`, propagate on `Point x`—we have no hope of eliminating `x`, but if we hold `x` constant, then we have a chance of eliminating arguments for which `x` is passed as an actual.

28.3.3 Dominator Analysis

Does anyone see where this is going yet? Pausing for a little thought, [MatthewFluet](#) realized that he had once before tried proposing this kind of "fix" to a fixed-point analysis—when we were first investigating the [Contify](#) optimization in light of John Reppy's CWS paper. Of course, that "fix" failed because it defined a non-monotonic function and one couldn't take the fixed point. But, [StephenWeeks](#) suggested a dominator based approach, and we were able to show that, indeed, the dominator analysis subsumed both the previous call based analysis and the cont based analysis. And, a moment's reflection reveals further parallels: when `varInfo: Var.t -> Var.t option list ref`, we have something analogous to the call analysis, and when `varInfo: Var.t -> VarLattice.t`, we have something analogous to the cont analysis. Maybe there is something analogous to the dominator approach (and therefore superior to the previous analyses).

And this turns out to be the case. Construct the graph G as follows:

```
nodes(G) = {Root} U Var.t
edges(G) = {Root -> v | v bound in a Statement.t or
            in the Function.t args} U
            {xi -> ai | L(x1, ..., xn) transfer where (a1, ..., an)
              are the formals of L} U
            {Root -> a | a is a block argument used in an unknown context}
```

Let $\text{idom}(x)$ be the immediate dominator of x in G with root `Root`. Now, any block argument a such that $\text{idom}(a) = x \llcorner \text{Root}$ can be optimized by setting $a = x$ at the beginning of the block and dropping the argument from `GotO` transfers.

Furthermore, experimental evidence suggests (and we are confident that a formal presentation could prove) that the dominator analysis subsumes the "syntactic" and "fixpoint" based analyses in this context as well and that the dominator analysis gets "everything" in one go.

28.3.4 Final Thoughts

I must admit, I was rather surprised at this progression and final result. At the outset, I never would have thought of a connection between [Contify](#) and [CommonArg](#) optimizations. They would seem to be two completely different optimizations. Although, this may not really be the case. As one of the reviewers of the ICFP paper said:

I understand that such a form of CPS might be convenient in some cases, but when we're talking about analyzing code to detect that some continuation is constant, I think it makes a lot more sense to make all the continuation arguments completely explicit.

I believe that making all the continuation arguments explicit will show that the optimization can be generalized to eliminating constant arguments, whether continuations or not.

What I think the common argument optimization shows is that the dominator analysis does slightly better than the reviewer puts it: we find more than just constant continuations, we find common continuations. And I think this is further justified by the fact that I have observed common argument eliminate some `env_X` arguments which would appear to correspond to determining that while the closure being executed isn't constant it is at least the same as the closure being passed elsewhere.

At first, I was curious whether or not we had missed a bigger picture with the dominator analysis. When we wrote the contification paper, I assumed that the dominator analysis was a specialized solution to a specialized problem; we never suggested that it was a technique suited to a larger class of analyses. After initially finding a connection between [Contify](#) and [CommonArg](#) (and thinking that the only connection was the technique), I wondered if the dominator technique really was applicable to a larger class of analyses. That is still a question, but after writing up the above, I'm suspecting that the "real story" is that the dominator analysis is a solution to the common argument optimization, and that the [Contify](#) optimization is specializing [CommonArg](#) to the case of continuation arguments (with a different transformation at the end). (Note, a whole-program, inter-procedural common argument analysis doesn't really make sense (in our [SSA IntermediateLanguage](#)), because the only way of passing values between functions is as arguments. (Unless of course in the case that the common argument is also a constant argument, in which case [ConstantPropagation](#) could lift it to a global.) The inter-procedural [Contify](#) optimization works out because there we move the function to the argument.)

Anyways, it's still unclear to me whether or not the dominator based approach solves other kinds of problems.

28.3.5 Phase Ordering

On the downside, the optimization doesn't have a huge impact on runtime, although it does predictably saved some code size. I stuck it in the optimization sequence after [Flatten](#) and (the third round of) [LocalFlatten](#), since it seems to me that we could have cases where some components of a tuple used as an argument are common, but the whole tuple isn't. I think it makes sense to add it after [IntroduceLoops](#) and [LoopInvariant](#) (even though [CommonArg](#) get some things that [LoopInvariant](#) gets, it doesn't get all of them). I also think that it makes sense to add it before [CommonSubexp](#), since identifying variables could expose more common subexpressions. I would think a similar thought applies to [RedundantTests](#).

29 CommonBlock

[CommonBlock](#) is an optimization pass for the [SSA IntermediateLanguage](#), invoked from [SSASimplify](#).

29.1 Description

It eliminates equivalent blocks in a [SSA](#) function. The equivalence criteria requires blocks to have no arguments or statements and transfer via `Raise`, `Return`, or `Goto` of a single global variable.

29.2 Implementation

- `common-block.fun`

29.3 Details and Notes

- Rewrites

```
L_X ()
  raise (global_Y)
```

to

```
L_X ()
  L_Y' ()
```

and adds

```
L_Y' ()
  raise (global_Y)
```

to the [SSA](#) function.

- Rewrites

```
L_X ()
  return (global_Y)
```

to

```
L_X ()
  L_Y' ()
```

and adds

```
L_Y' ()
  return (global_Y)
```

to the [SSA](#) function.

- Rewrites

```
L_X ()
  L_Z (global_Y)
```

to

```
L_X ()
  L_Y' ()
```

and adds

```
L_Y' ()  
  L_Z (global_Y)
```

to the [SSA](#) function.

The [Shrink](#) pass rewrites all uses of `L_X` to `L_Y'` and drops `L_X`.

For example, all uncaught `Overflow` exceptions in a [SSA](#) function share the same raising block.

30 CommonSubexp

`CommonSubexp` is an optimization pass for the [SSA IntermediateLanguage](#), invoked from [SSASimplify](#).

30.1 Description

It eliminates instances of common subexpressions.

30.2 Implementation

- [common-subexp.fun](#)

30.3 Details and Notes

In addition to getting the usual sorts of things like

- ```
(w + 0wx1) + (w + 0wx1)
```

rewritten to

```
let val w' = w + 0wx1 in w' + w' end
```

it also gets things like

- ```
val a = Array_array n
val b = Array_length a
```

rewritten to

```
val a = Array_array n
val b = n
```

`Arith` transfers are handled specially. The *result* of an `Arith` transfer can be used in *common* `Arith` transfers that it dominates:

- ```
val l = (n + m) + (n + m)

val k = (l + n) + ((l + m) handle Overflow => ((l + m)
 handle Overflow => l + n))
```

is rewritten so that `(n + m)` is computed exactly once, as are `(l + n)` and `(l + m)`.

## 31 CompilationManager

The **Compilation Manager** (CM) is SML/NJ's mechanism for supporting programming-in-the-very-large.

### 31.1 Porting SML/NJ CM files to MLton

To help in porting CM files to MLton, the MLton source distribution includes the sources for a utility, `cm2mlb`, that will print an **ML Basis** file with essentially the same semantics as the CM file—handling the full syntax of CM supported by your installed SML/NJ version and correctly handling export filters. When `cm2mlb` encounters a `.cm` import, it attempts to convert it to a corresponding `.mlb` import. CM anchored paths are translated to paths according to a default configuration file (`cm2mlb-map`). For example, the default configuration includes

```
Standard ML Basis Library
$SMLNJ-BASIS $(SML_LIB)/basis
$basis.cm $(SML_LIB)/basis
$basis.cm/basis.cm $(SML_LIB)/basis/basis.mlb
```

to ensure that a `$/basis.cm` import is translated to a `$(SML_LIB)/basis/basis.mlb` import. See `util/cm2mlb` for details. Building `cm2mlb` requires that you have already installed a recent version of SML/NJ.

## 32 CompilerOverview

The following table shows the overall structure of the compiler. [IntermediateLanguages](#) are shown in the center column. The names of compiler passes are listed in the left and right columns.

| <b>Compiler Overview</b>       |                             |                                |
|--------------------------------|-----------------------------|--------------------------------|
| <i>Translation Passes</i>      | <i>IntermediateLanguage</i> | <i>Optimization Passes</i>     |
|                                | Source                      |                                |
| <a href="#">FrontEnd</a>       |                             |                                |
|                                | AST                         |                                |
| <a href="#">Elaborate</a>      |                             |                                |
|                                | CoreML                      | <a href="#">CoreMLSimplify</a> |
| <a href="#">Defunctorize</a>   |                             |                                |
|                                | XML                         | <a href="#">XMLSimplify</a>    |
| <a href="#">Monomorphise</a>   |                             |                                |
|                                | SXML                        | <a href="#">SXMLSimplify</a>   |
| <a href="#">ClosureConvert</a> |                             |                                |
|                                | SSA                         | <a href="#">SSASimplify</a>    |
| <a href="#">ToSSA2</a>         |                             |                                |
|                                | SSA2                        | <a href="#">SSA2Simplify</a>   |
| <a href="#">ToRSSA</a>         |                             |                                |
|                                | RSSA                        | <a href="#">RSSASimplify</a>   |
| <a href="#">ToMachine</a>      |                             |                                |
|                                | Machine                     |                                |
| <a href="#">Codegen</a>        |                             |                                |

The `Compile` functor (`compile.sig`, `compile.fun`), controls the high-level view of the compiler passes, from [FrontEnd](#) to code generation.

## 33 CompilerPassTemplate

An analysis pass for the [ZZZ IntermediateLanguage](#), invoked from [ZZZOtherPass](#). An implementation pass for the [ZZZ IntermediateLanguage](#), invoked from [ZZZSimplify](#). An optimization pass for the [ZZZ IntermediateLanguage](#), invoked from [ZZZSimplify](#). A rewrite pass for the [ZZZ IntermediateLanguage](#), invoked from [ZZZOtherPass](#). A translation pass from the [ZZA IntermediateLanguage](#) to the [ZZB IntermediateLanguage](#).

### 33.1 Description

A short description of the pass.

### 33.2 Implementation

- [ZZZ.fun](#)

### 33.3 Details and Notes

Relevant details and notes.

---

## 34 CompileTimeOptions

MLton's compile-time options control the name of the output file, the verbosity of compile-time messages, and whether or not certain optimizations are performed. They also can specify which intermediate files are saved and can stop the compilation process early, at some intermediate pass, in which case compilation can be resumed by passing the generated files to MLton. MLton uses the input file suffix to determine the type of input program. The possibilities are `.c`, `.mlb`, `.o`, `.s`, and `.sml`.

With no arguments, MLton prints the version number and exits. For a usage message, run MLton with an invalid switch, e.g. `mlton -z`. In the explanation below and in the usage message, for flags that take a number of choices (e.g. `{true|false}`), the first value listed is the default.

### 34.1 Options

- `-align n`  
Aligns object in memory by the specified alignment (4 or 8). The default varies depending on architecture.
- `-as-opt option`  
Pass *option* to `gcc` when compiling assembler code. If you wish to pass an option to the assembler, you must use `gcc`'s `-Wa,` syntax.
- `-cc-opt option`  
Pass *option* to `gcc` when compiling C code.
- `-codegen {native|x86|amd64|c}`  
Generate native code or C code. With `-codegen native` (`-codegen x86` or `-codegen amd64`), MLton typically compiles more quickly and generates better code.
- `-const name value`  
Set the value of a compile-time constant. Here is a list of available constants, their default values, and what they control.
  - `Exn.keepHistory {false|true}`  
Enable `MLton.Exn.history`. See [MLtonExn](#) for details. There is a performance cost to setting this to `true`, both in memory usage of exceptions and in run time, because of additional work that must be performed at each exception construction, raise, and handle.
- `-default-ann ann`  
Specify default [ML Basis annotations](#). For example, `-default-ann 'warnUnused true'` causes unused variable warnings to be enabled by default. A default is overridden by the corresponding annotation in an ML Basis file.
- `-default-type type`  
Specify the default binding for a primitive type. For example, `-default-type word64` causes the top-level type `word` and the top-level structure `Word` in the [Basis Library](#) to be equal to `Word64.word` and `Word64:WORD`, respectively. Similarly, `-default-type intinf` causes the top-level type `int` and the top-level structure `Int` in the [Basis Library](#) to be equal to `IntInf.int` and `IntInf:INTEGER`, respectively.
- `-disable-ann ann`  
Ignore the specified [ML Basis annotation](#) in every ML Basis file. For example, to see *all* match and unused warnings, compile with
 

```
-default-ann 'warnUnused true'
-disable-ann forceUsed
-disable-ann nonexhaustiveMatch
-disable-ann redundantMatch
-disable-ann warnUnused
```
- `-export-header file`  
Write C prototypes to *file* for all of the functions in the program [exported from SML to C](#).

- `-ieee-fp {false|true}`

Cause the native code generator to be pedantic about following the IEEE floating point standard. By default, it is not, because of the performance cost. This only has an effect with `-codegen x86`.

- `-inline n`

Set the inlining threshold used in the optimizer. The threshold is an approximate measure of code size of a procedure. The default is 320.

- `-keep {g|o}`

Save intermediate files. If no `-keep` argument is given, then only the output file is saved.

|   |                                         |
|---|-----------------------------------------|
| g | generated .c and .s files passed to gcc |
| o | object (.o) files                       |

- `-link-opt option`

Pass *option* to gcc when linking. You can use this to specify library search paths, e.g. `-link-opt -Lpath`, and libraries to link with, e.g., `-link-opt -lfoo`, or even both at the same time, e.g. `-link-opt '-Lpath -lfoo'`. If you wish to pass an option to the linker, you must use gcc's `-Wl`, syntax, e.g., `-link-opt '-Wl,--export-dynamic'`.

- `-mlb-path-map file`

Use *file* as an [ML Basis path map](#) to define additional MLB path variables. Multiple uses of `-mlb-path-map` and `-mlb-path-var` are allowed, with variable definitions in later path maps taking precedence over earlier ones.

- `-mlb-path-var name value`

Define an additional MLB path variable. Multiple uses of `-mlb-path-map` and `-mlb-path-var` are allowed, with variable definitions in later path maps taking precedence over earlier ones.

- `-output file`

Specify the name of the final output file. The default name is the input file name with its suffix removed and an appropriate, possibly empty, suffix added.

- `-profile {no|alloc|count|time}`

Produce an executable that gathers [profiling](#) data. When such an executable is run, it produces an `mlmon.out` file.

- `-profile-branch {false|true}`

If true, the profiler will separately gather profiling data for each branch of a function definition, `case` expression, and `if` expression.

- `-profile-stack {false|true}`

If true, the executable will gather profiling data for all functions on the stack, not just the currently executing function. See [ProfilingTheStack](#).

- `-profile-val {false|true}`

If true, the profiler will separately gather profiling data for each (expansive) `val` declaration.

- `-runtime arg`

Pass argument to the runtime system via `@MLton`. See [RunTimeOptions](#). The argument will be processed before other `@MLton` command line switches. Multiple uses of `-runtime` are allowed, and will pass all the arguments in order. If the same runtime switch occurs more than once, then the last setting will take effect. There is no need to supply the leading `@MLton` or the trailing `--`; these will be supplied automatically.

An argument to `-runtime` may contain spaces, which will cause the argument to be treated as a sequence of words by the runtime. For example the command line:

```
mlton -runtime 'ram-slop 0.4' foo.sml
```

will cause `foo` to run as if it had been called like:

```
foo @MLton ram-slop 0.4 --
```

An executable created with `-runtime stop` doesn't process any @MLton arguments. This is useful to create an executable, e.g., `echo`, that must treat @MLton like any other command-line argument.

```
% mlton -runtime stop echo.sml
% echo @MLton --
@MLton --
```

- `-show-basis file`

Pretty print to *file* the basis defined by the input program. See [ShowBasis](#).

- `-show-def-use file`

Output def-use information to *file*. Each identifier that is defined appears on a line, followed on subsequent lines by the position of each use.

- `-stop {f|g|o|tc}`

Specify when to stop.

|    |                                                                                |
|----|--------------------------------------------------------------------------------|
| f  | list of files on stdout (only makes sense when input is <code>foo.mlb</code> ) |
| g  | generated <code>.c</code> and <code>.s</code> files                            |
| o  | object ( <code>.o</code> ) files                                               |
| tc | after type checking                                                            |

If you compile with `-stop g` or `-stop o`, you can resume compilation by running MLton on the generated `.c` and `.s` or `.o` files.

- `-target {self|...}`

Generate an executable that runs on the specified platform. The default is `self`, which means to compile for the machine that MLton is running on. To use any other target, you must first install a [cross compiler](#).

- `-target-as-opt target option`

Like `-as-opt`, this passes *option* to `gcc` when compiling assembler code, except it only passes *option* when the target architecture or operating system is *target*.

- `-target-cc-opt target option`

Like `-cc-opt`, this passes *option* to `gcc` when compiling C code, except it only passes *option* when the target architecture or operating system is *target*.

- `-target-link-opt target option`

Like `-link-opt`, this passes *option* to `gcc` when linking, except it only passes *option* when the target architecture or operating system is *target*.

- `-verbose {0|1|2|3}`

How verbose to be about what passes are running. The default is 0.

|   |                                          |
|---|------------------------------------------|
| 0 | silent                                   |
| 1 | calls to compiler, assembler, and linker |
| 2 | 1, plus intermediate compiler passes     |
| 3 | 2, plus some data structure sizes        |

## 35 CompilingWithSMLNJ

You can compile MLton with [SML/NJ](#), however the resulting compiler will run much more slowly than MLton compiled by itself. We don't recommend using SML/NJ as a means of [porting MLton](#) to a new platform or bootstrapping on a new platform.

If you do want to build MLton with SML/NJ, it is best to have a binary MLton package installed. If you don't, here are some issues you may encounter when you run `make nj-mlton`.

You will get (many copies of) the error message:

```
/bin/sh: mlton: not found
```

The Makefile calls `mlton` to determine dependencies, and can proceed in spite of this error.

If you don't have a `mlton` executable, you will get the error message:

```
Error: cannot upgrade basis because the compiler doesn't work
```

We use `upgrade-basis.sml` to work around basis library differences, allowing a version of MLton written for a newer basis library to be compiled by an older version of MLton. The file isn't necessary when building with SML/NJ, but is listed in `$(SOURCES)`, so the Makefile is attempting to build it. Building `upgrade-basis.sml` requires the old version of MLton to be around so that the right stubs can be built.

To work around this problem, do one of the following.

- Manually tweak sources to remove `$(UP)` until you're finished building MLton with SML/NJ and have a working MLton.
- Build `upgrade-basis.sml` on some other machine with a working MLton and copy it over.

If you don't have an `mllex` executable, you will get the error message:

```
mllex: Command not found
```

Building MLton requires `mllex` and `mlyacc` executables, which are distributed with a binary package of MLton. The easiest solution is to copy the front-end lexer/parser files from a different machine (`ml.grm.sml`, `ml.grm.sig`, `ml.lex.sml`, `mlb.grm.sig`, `mlb.grm.sml`).

## 36 ConcurrentML

**Concurrent ML** is an SML concurrency library based on synchronous message passing. MLton has an initial port of CML from SML/NJ, but is missing a thread-safe wrapper around the Basis Library and event-based equivalents to IO and OS functions.

All of the core CML functionality is present.

```
structure CML: CML
structure SyncVar: SYNC_VAR
structure Mailbox: MAILBOX
structure Multicast: MULTICAST
structure SimpleRPC: SIMPLE_RPC
structure RunCML: RUN_CML
```

The RUN\_CML signature is minimal.

```
signature RUN_CML =
 sig
 val isRunning: unit -> bool
 val doit: (unit -> unit) * Time.time option -> OS.Process.status
 val shutdown: OS.Process.status -> 'a
 end
```

MLton's RunCML structure does not include all of the cleanup and logging operations of SML/NJ's RunCML structure. However, the implementation does include the CML.timeOutEvt and CML.atTimeEvt functions, and a preemptive scheduler that knows to sleep when there are no ready threads and some threads blocked on time events.

Because MLton does not wrap the Basis Library for CML, the "right" way to call a Basis Library function that is stateful is to wrap the call with MLton.Thread.atomically.

### 36.1 Usage

- You can import the CML Library into an MLB file with:

| MLB file                | Description |
|-------------------------|-------------|
| \$(SML_LIB)/cml/cml.mlb |             |

- If you are porting a project from SML/NJ's [CompilationManager](#) to MLton's [ML Basis system](#) using cm2mlb, note that the following map is included by default:

```
CML Library
$cml $(SML_LIB)/cml
$cml/cml.cm $(SML_LIB)/cml/cml.mlb
```

This will automatically convert a \$cml/cml.cm import in an input .cm file into a \$(SML\_LIB)/cml/cml.mlb import in the output .mlb file.

### 36.2 Also see

- [ConcurrentMLImplementation](#)
- [eXene](#)

## 37 ConcurrentMLImplementation

Here are some notes on MLton's implementation of [ConcurrentML](#).

Concurrent ML was originally implemented for SML/NJ. It was ported to MLton in the summer of 2004. The main difference between the implementations is that SML/NJ uses continuations to implement CML threads, while MLton uses its underlying [thread](#) package. Presently, MLton's threads are a little more heavyweight than SML/NJ's continuations, but it's pretty clear that there is some fat there that could be trimmed.

The implementation of CML in SML/NJ is built upon the first-class continuations of the `SMLofNJ.Cont` module.

```
type 'a cont
val callcc: ('a cont -> 'a) -> 'a
val isolate: ('a -> unit) -> 'a cont
val throw: 'a cont -> 'a -> 'b
```

The implementation of CML in MLton is built upon the first-class threads of the [MLtonThread](#) module.

```
type 'a t
val new: ('a -> unit) -> 'a t
val prepare: 'a t * 'a -> Runnable.t
val switch: ('a t -> Runnable.t) -> 'a
```

The port is relatively straightforward, because CML always throws to a continuation at most once. Hence, an "abstract" implementation of CML could be built upon first-class one-shot continuations, which map equally well to SML/NJ's continuations and MLton's threads.

The "essence" of the port is to transform:

```
callcc (fn k => ... throw k' v')
```

to

```
switch (fn t => ... prepare (t', v'))
```

which suffices for the vast majority of the CML implementation.

There was only one complicated transformation: blocking multiple base events. In SML/NJ CML, the representation of base events is given by:

```
datatype 'a event_status
 = ENABLED of {prio: int, doFn: unit -> 'a}
 | BLOCKED of {
 transId: trans_id ref,
 cleanUp: unit -> unit,
 next: unit -> unit
 } -> 'a
type 'a base_evt = unit -> 'a event_status
```

When synchronizing on a set of base events, which are all blocked, we must invoke each BLOCKED function with the same `transId` and `cleanUp` (the `transId` is (checked and) set to CANCEL by the `cleanUp` function, which is invoked by the first enabled event; this "fizzles" every other event in the synchronization group that later becomes enabled). However, each BLOCKED function is implemented by a `callcc`, so that when the event is enabled, it throws back to the point of synchronization. Hence, the next function (which doesn't return) is invoked by the BLOCKED function to escape the `callcc` and continue in the thread performing the synchronization. In SML/NJ this is implemented as follows:

```
fun ext ([], blockFns) = callcc (fn k => let
 val throw = throw k
 val (transId, setFlg) = mkFlg()
 fun log [] = S.atomicDispatch ()
 | log (blockFn:: r) =
 throw (blockFn {
```

```

 transId = transId,
 cleanUp = setFlg,
 next = fn () => log r
 })
in
 log blockFns; error "[log]"
end)

```

(Note that `S.atomicDispatch` invokes the continuation of the next continuation on the ready queue.) This doesn't map well to the MLton thread model. Although it follows the

```
callcc (fn k => ... throw k v)
```

model, the fact that `blockFn` will also attempt to do

```
callcc (fn k' => ... next ())
```

means that the naive transformation will result in nested `switch-es`.

We need to think a little more about what this code is trying to do. Essentially, each `blockFn` wants to capture this continuation, hold on to it until the event is enabled, and continue with `next`; when the event is enabled, before invoking the continuation and returning to the synchronization point, the `cleanUp` and other event specific operations are performed.

To accomplish the same effect in the MLton thread implementation, we have the following:

```

datatype 'a status =
 ENABLED of {prio: int, doitFn: unit -> 'a}
| BLOCKED of {transId: trans_id,
 cleanUp: unit -> unit,
 next: unit -> rdy_thread} -> 'a

type 'a base = unit -> 'a status

fun ext ([], blockFns): 'a =
 S.atomicSwitch
 (fn (t: 'a S.thread) =>
 let
 val (transId, cleanUp) = TransID.mkFlg ()
 fun log blockFns: S.rdy_thread =
 case blockFns of
 [] => S.next ()
 | blockFn::blockFns =>
 (S.prep o S.new)
 (fn _ => fn () =>
 let
 val () = S.atomicBegin ()
 val x = blockFn {transId = transId,
 cleanUp = cleanUp,
 next = fn () => log blockFns}
 in S.switch(fn _ => S.prepVal (t, x))
 end)
 end)
 in
 log blockFns
 end)

```

To avoid the nested `switch-es`, I run the `blockFn` in it's own thread, whose only purpose is to return to the synchronization point. This corresponds to the `throw (blockFn {...})` in the SML/NJ implementation. I'm worried that this implementation might be a little expensive, starting a new thread for each blocked event (when there are only multiple blocked events in a synchronization group). But, I don't see another way of implementing this behavior in the MLton thread model.

Note that another way of thinking about what is going on is to consider each `blockFn` as prepending a different set of actions to the thread `t`. It might be possible to give a `MLton.Thread.unsafePrepend`.

```

fun unsafePrepend (T r: 'a t, f: 'b -> 'a): 'b t =
 let
 val t =
 case !r of
 Dead => raise Fail "prepend to a Dead thread"
 | New g => New (g o f)
 | Paused (g, t) => Paused (fn h => g (f o h), t)
 in (* r := Dead; *)
 T (ref t)
 end

```

I have commented out the `r := Dead`, which would allow multiple prepends to the same thread (i.e., not destroying the original thread in the process). Of course, only one of the threads could be run: if the original thread were in the `Paused` state, then multiple threads would share the underlying runtime/primitive thread. Now, this matches the "one-shot" nature of CML continuations/threads, but I'm not comfortable with extending `MLton.Thread` with such an unsafe operation.

Other than this complication with blocking multiple base events, the port was quite routine. (As a very pleasant surprise, the CML implementation in SML/NJ doesn't use any SML/NJ-isms.) There is a slight difference in the way in which critical sections are handled in SML/NJ and MLton; since `MLton.Thread.switch` *always* leaves a critical section, it is sometimes necessary to add additional `atomicBegin-s/atomicEnd-s` to ensure that we remain in a critical section after a thread switch.

While looking at virtually every file in the core CML implementation, I took the liberty of simplifying things where it seemed possible; in terms of style, the implementation is about half-way between Reppy's original and MLton's.

Some changes of note:

- `util/` contains all pertinent data-structures: (functional and imperative) queues, (functional) priority queues. Hence, it should be easier to switch in more efficient or real-time implementations.
- `core-cml/scheduler.sml`: in both implementations, this is where most of the interesting action takes place. I've made the connection between `MLton.Thread.t-s` and `ThreadId.thread_id-s` more abstract than it is in the SML/NJ implementation, and encapsulated all of the `MLton.Thread` operations in this module.
- eliminated all of the "by hand" inlining

### 37.1 Future Extensions

The CML documentation says the following:

```
CML.joinEvt: thread_id -> unit event
```

- `joinEvt tid`  
creates an event value for synchronizing on the termination of the thread with the ID `tid`. There are three ways that a thread may terminate: the function that was passed to `spawn` (or `spawnnc`) may return; it may call the `exit` function, or it may have an uncaught exception. Note that `joinEvt` does not distinguish between these cases; it also does not become enabled if the named thread deadlocks (even if it is garbage collected).

I believe that the `MLton.Finalizable` might be able to relax that last restriction. Upon the creation of a `'a Scheduler.thread`, we could attach a finalizer to the underlying `'a MLton.Thread.t` that enables the `joinEvt` (in the associated `ThreadId.thread_id`) when the `'a MLton.Thread.t` becomes unreachable.

I don't know why CML doesn't have

```
CML.kill: thread_id -> unit
```

which has a fairly simple implementation—setting a kill flag in the `thread_id` and adjusting the scheduler to discard any killed threads that it takes off the ready queue. The fairness of the scheduler ensures that a killed thread will eventually be discarded. The semantics are little murky for blocked threads that are killed, though. For example, consider a thread blocked on

`SyncVar.mTake mv` and a thread blocked on `SyncVar.mGet mv`. If the first thread is killed while blocked, and a third thread does `SyncVar.mPut (mv, x)`, then we might expect that we'll enable the second thread, and never the first. But, when only the ready queue is able to discard killed threads, then the `SyncVar.mPut` could enable the first thread (putting it on the ready queue, from which it will be discarded) and leave the second thread blocked. We could solve this by adjusting the `TransID.trans_id` types and the "cleaner" functions to look for both canceled transactions and transactions on killed threads.

John Reppy says that [MarlowEtAl101](#) and [FlattFidler04](#) explain why `CML.kill` would be a bad idea.

Between `CML.timeOutEvt` and `CML.kill`, one could give an efficient solution to the recent `comp.lang.ml` post about terminating a function that doesn't complete in a given time.

```
fun timeOut (f: unit -> 'a, t: Time.time): 'a option =
 let
 val iv = SyncVar.iVar ()
 val tid = CML.spawn (fn () => SyncVar.iPut (iv, f ()))
 in
 CML.select
 [CML.wrap (CML.timeOutEvt t, fn () => (CML.kill tid; NONE)),
 CML.wrap (SyncVar.iGetEvt iv, fn x => SOME x)]
 end
```

## 37.2 Space Safety

There are some CML related posts on the MLton mailing list:

- <http://www.mlton.org/pipermail/mlton/2004-May/>

that discuss concerns that SML/NJ's implementation is not space efficient, because multi-shot continuations can be held indefinitely on event queues. MLton is better off because of the one-shot nature — when an event enables a thread, all other copies of the thread waiting in other event queues get turned into dead threads (of zero size).

## 38 ConstantPropagation

[ConstantPropagation](#) is an optimization pass for the [SSA IntermediateLanguage](#), invoked from [SSASimplify](#).

### 38.1 Description

This is whole-program constant propagation, even through data structures. It also performs globalization of (small) values computed once.

Uses [Multi](#).

### 38.2 Implementation

- `constant-propagation.fun`

### 38.3 Details and Notes

---

## 39 Contact

### 39.1 Mailing lists

There are three mailing lists available.

- [MLton-user@mlton.org](mailto:MLton-user@mlton.org)  
MLton user community discussion
  - [subscribe](#)
  - [archive \(Gmane; current\)](#), [archive \(SourceForge; current\)](#), [archive \(PiperMail; through 201110\)](#)
- [MLton-devel@mlton.org](mailto:MLton-devel@mlton.org)  
MLton developer community discussion
  - [subscribe](#)
  - [archive \(Gmane; current\)](#), [archive \(SourceForge; current\)](#), [archive \(PiperMail; through 201110\)](#)
- [MLton-commit@mlton.org](mailto:MLton-commit@mlton.org)  
MLton code commits
  - [subscribe](#)
  - [archive \(SourceForge; current\)](#), [archive \(PiperMail; through 201110\)](#)

#### 39.1.1 Mailing list policies

- Both mailing lists are unmoderated. However, the mailing lists are configured them to discard all spam, to hold all non-subscriber posts for moderation, to accept all subscriber posts, and to admin approve subscription requests. Please contact [Matthew Fluet](#) if it appears that your messages are being discarded as spam.
- Large messages (over 256K) should not be sent. Rather, please send an email containing the discussion text and a link to any large files.
- Discussions started on the mailing lists should stay on the mailing lists. Private replies may be bounced to the mailing list for the benefit of those following the discussion.
- Discussions started on [MLton-user@mlton.org](mailto:MLton-user@mlton.org) may be migrated to [MLton-devel@mlton.org](mailto:MLton-devel@mlton.org), particularly when the discussion shifts from how to use MLton to how to modify MLton (e.g., to fix a bug identified by the initial discussion).

### 39.2 IRC

- Some MLton developers and users are in channel #sm1 on <http://freenode.net>.
-

## 40 Contify

[Contify](#) is an optimization pass for the [SSA IntermediateLanguage](#), invoked from [SSASimplify](#).

### 40.1 Description

Contification is a compiler optimization that turns a function that always returns to the same place into a continuation. This exposes control-flow information that is required by many optimizations, including traditional loop optimizations.

### 40.2 Implementation

- `contify.fun`

### 40.3 Details and Notes

See [Contification Using Dominators](#). The intermediate language described in that paper has since evolved to the [SSA IntermediateLanguage](#); hence, the complication described in Section 6.1 is no longer relevant.

---

## 41 CoreML

Core ML is an [IntermediateLanguage](#), translated from [AST](#) by [Elaborate](#), optimized by [CoreMLSimplify](#), and translated by [Defunctorize](#) to [XML](#).

### 41.1 Description

[CoreML](#) is polymorphic, higher-order, and has nested patterns.

### 41.2 Implementation

- [core-ml.sig](#)
- [core-ml.fun](#)

### 41.3 Type Checking

The [CoreML IntermediateLanguage](#) has no independent type checker.

### 41.4 Details and Notes

---

## 42 CoreMLSimplify

The single optimization pass for the [CoreML IntermediateLanguage](#) is controlled by the `Compile` functor (`compile.fun`).

The following optimization pass is implemented:

- [DeadCode](#)
-

## 43 Credits

MLton was designed and implemented by HenryCejtin, MatthewFluet, SureshJagannathan, and StephenWeeks.

- [HenryCejtin](#) wrote the `IntInf` implementation, the original profiler, the original man pages, the `.spec` files for the RPMs, and lots of little hacks to speed stuff up.
- [MatthewFluet](#) implemented the X86 and AMD64 native code generators, ported `mlprof` to work with the native code generator, did a lot of work on the SSA optimizer, both adding new optimizations and improving or porting existing optimizations, updated the [Basis Library](#) implementation, ported [ConcurrentML](#) and [ML-NLFFI](#) to MLton, implemented the [ML Basis system](#), ported MLton to 64-bit platforms, and currently leads the project.
- [SureshJagannathan](#) implemented some early inlining and uncurrying optimizations.
- [StephenWeeks](#) implemented most of the original version of MLton, and continues to keep his fingers in most every part.

Many people have helped us over the years. Here is an alphabetical list.

- [JesperLouisAndersen](#) sent several patches to improve the runtime on FreeBSD and ported MLton to run on NetBSD and OpenBSD.
- [JohnnyAndersen](#) implemented `BinIO`, modified MLton so it could cross compile to MinGW, and provided useful discussion about cross-compilation.
- Christopher Cramer contributed support for additional `Posix.ProcEnv.sysconf` variables and performance improvements for `String.concatWith`.
- Alain Deutsch and [PolySpace Technologies](#) provided many bug fixes and runtime system improvements, code to help the Sparc/Solaris port, and funded a number of improvements to MLton.
- Martin Elsman provided helpful discussions in the development of the [ML Basis system](#).
- Brent Fulgham ported MLton most of the way to MinGW.
- [AdamGoode](#) provided a script to build the PDF MLton Guide and maintains the [Fedora](#) packages.
- Simon Helsen provided bug reports, suggestions, and helpful discussions.
- Joe Hurd provided useful discussion and feedback on source-level profiling.
- [VesaKarvonen](#) contributed `esml-mode.el` and `esml-mlb-mode.el` (see [Emacs](#)), contributed patches for improving match warnings, contributed `esml-du-mlton.el` and extended def-use output to include types of variable definitions (see [EmacsDefUseMode](#)), and improved constant folding of floating-point operations.
- Richard Kelsey provided helpful discussions.
- Ville Laurikari ported MLton to IA64/HPUX, HPPA/HPUX, PowerPC/AIX, PowerPC64/AIX.
- Geoffrey Mainland helped with FreeBSD packaging.
- Eric McCorkle ported MLton to Intel Mac.
- [TomMurphy](#) wrote the original version of `MLton.Syslog` as part of his `mlftpd` project, and has sent many useful bug reports and suggestions.
- Michael Neumann helped to patch the runtime to compile under FreeBSD.
- Barak Pearlmuter built the original [Debian package](#) for MLton, and helped us to take over the process.
- Filip Pizlo ported MLton to (PowerPC) Darwin.
- John Reppy assisted in porting MLton to Intel Mac.
- Sam Rushing ported MLton to FreeBSD.

- Jeffrey Mark Siskind provided helpful discussions and inspiration with his Stalin Scheme compiler.
- [Wesley Terpstra](#) added support for `MLton.Process.create`, made a number of contributions to the [ForeignFunctionInterface](#), contributed a number of runtime system patches, added support for compiling to a [C library](#), ported MLton to [MinGW](#) and all [Debian](#) supported architectures with [cross-compiling](#) support, and maintains the [Debian](#) and [MinGW](#) packages.
- Luke Ziarek assisted in porting MLton to (PowerPC) Darwin.

We have also benefited from other software development tools and used code from other sources.

- MLton was developed using [Standard ML of New Jersey](#) and the [Compilation Manager \(CM\)](#)
- MLton's `lexer` (`mlton/frontend/ml.lex`), `parser` (`mlton/frontend/ml.grm`), and `precedence-parser` (`mlton/elaborate/precedence-parse.fun`) are modified versions of code from SML/NJ.
- The MLton [Basis Library](#) implementation of conversions between binary and decimal representations of reals uses David Gay's [gdtoa](#) library.
- The MLton [Basis Library](#) implementation uses modified versions of portions of the the SML/NJ Basis Library implementation modules `OS.IO`, `Posix.IO`, `Process`, and `Unix`.
- The MLton [Basis Library](#) implementation uses modified versions of portions of the [ML Kit](#) Version 4.1.4 Basis Library implementation modules `Path`, `Time`, and `Date`.
- Many of the benchmarks come from the SML/NJ benchmark suite.
- Many of the regression tests come from the ML Kit Version 4.1.4 distribution, which borrowed them from the [Moscow ML](#) distribution.
- MLton uses the [GNU multiprecision library](#) for its implementation of `IntInf`.
- MLton's implementation of `mlex`, `mlyacc`, the [ckit Library](#), the [ML-LPT Library](#), the [MLRISC Library](#), the [SML/NJ Library](#), [Concurrent ML](#), `mlnlffigen` and [ML-NLFFI](#) are modified versions of code from SML/NJ.

## 44 CrossCompiling

MLton's `-target` flag directs MLton to cross compile an application for another platform. By default, MLton is only able to compile for the machine it is running on. In order to use MLton as a cross compiler, you need to do two things.

1. Install the GCC cross-compiler tools on the host so that GCC can compile to the target.
2. Cross compile the MLton runtime system to build the runtime libraries for the target.

To make the terminology clear, we refer to the *host* as the machine MLton is running on and the *target* as the machine that MLton is compiling for.

To build a GCC cross-compiler toolset on the host, you can use the script `bin/build-cross-gcc`, available in the MLton sources, as a template. The value of the `target` variable in that script is important, since that is what you will pass to MLton's `-target` flag. Once you have the toolset built, you should be able to test it by cross compiling a simple hello world program on your host machine.

```
% gcc -b i386-pc-cygwin -o hello-world hello-world.c
```

You should now be able to run `hello-world` on the target machine, in this case, a Cygwin machine.

Next, you must cross compile the MLton runtime system and inform MLton of the availability of the new target. The script `bin/add-cross` from the MLton sources will help you do this. Please read the comments at the top of the script. Here is a sample run adding a Solaris cross compiler.

```
% add-cross sparc-sun-solaris sun blade
Making runtime.
Building print-constants executable.
Running print-constants on blade.
```

Running `add-cross` uses `ssh` to compile the runtime on the target machine and to create `print-constants`, which prints out all of the constants that MLton needs in order to implement the [Basis Library](#). The script runs `print-constants` on the target machine (`blade` in this case), and saves the output.

Once you have done all this, you should be able to cross compile SML applications. For example,

```
mlton -target i386-pc-cygwin hello-world.sml
```

will create `hello-world`, which you should be able to run from a Cygwin shell on your Windows machine.

### 44.1 Cross-compiling alternatives

Building and maintaining cross-compiling `gcc`'s is complex. You may find it simpler to use `mlton -keep g` to generate the files on the host, then copy the files to the target, and then use `gcc` or `mlton` on the target to compile the files.

## 45 CVS

**CVS** (Concurrent Versions System) is a version control system. The MLton project used CVS to maintain its [source code](#), but switched to [Subversion](#) on 20050730.

Here are some online CVS resources.

- [Open Source Development with CVS](#)
-

## 46 DeadCode

`DeadCode` is an optimization pass for the [CoreML IntermediateLanguage](#), invoked from [CoreMLSimplify](#).

### 46.1 Description

This pass eliminates declarations from the [Basis Library](#) not needed by the user program.

### 46.2 Implementation

- [dead-code.sig](#)
- [dead-code.fun](#)

### 46.3 Details and Notes

In order to compile small programs rapidly, a pass of dead code elimination is run in order to eliminate as much of the Basis Library as possible. The dead code elimination algorithm used is not safe in general, and only works because the Basis Library implementation has special properties:

- it terminates
- it performs no I/O

The dead code elimination includes the minimal set of declarations from the Basis Library so that there are no free variables in the user program (or remaining Basis Library implementation). It has a special hack to include all bindings of the form:

```
val _ = ...
```

There is an [ML Basis annotation](#), `deadCode true`, that governs which code is subject to this unsafe dead-code elimination.

## 47 DeepFlatten

`DeepFlatten` is an optimization pass for the [SSA2 IntermediateLanguage](#), invoked from [SSA2Simplify](#).

### 47.1 Description

This pass flattens into mutable fields of objects and into vectors.

For example, an `(int * int) ref` is represented by a 2 word object, and an `(int * int) array` contains pairs of `int`-s, rather than pointers to pairs of `int`-s.

### 47.2 Implementation

- `deep-flatten.fun`

### 47.3 Details and Notes

There are some performance issues with the deep flatten pass, where it consumes an excessive amount of memory.

- <http://www.mlton.org/pipermail/mlton/2005-April/026990.html>
- <http://www.mlton.org/pipermail/mlton-user/2010-June/001626.html>
- <http://www.mlton.org/pipermail/mlton/2010-December/030876.html>

A number of applications require compilation with `-drop-pass deepFlatten` to avoid exceeding available memory. It is often asked whether the deep flatten pass usually has a significant impact on performance. The standard benchmark suite was run with and without the deep flatten pass enabled when the pass was first introduced:

- <http://www.mlton.org/pipermail/mlton/2004-August/025760.html>

The conclusion is that it does not have a significant impact. However, these are micro benchmarks; other applications may derive greater benefit from the pass.

---

## 48 DefineTypeBeforeUse

[Standard ML](#) requires types to be defined before they are used. Because of type inference, the use of a type can be implicit; hence, this requirement is more subtle than it might appear. For example, the following program is not type correct, because the type of `r` is `t option ref`, but `t` is defined after `r`.

```
val r = ref NONE
datatype t = A | B
val () = r := SOME A
```

MLton reports the following error, indicating that the type defined on line 2 is used on line 1.

```
Error: z.sml 1.1.
 Type escapes the scope of its definition at z.sml 2.10.
 type: t
 in: val r = ref NONE
```

While the above example is benign, the following example shows how to cast an integer to a function by (implicitly) using a type before it is defined. In the example, the ref cell `r` is of type `t option ref`, where `t` is defined *after* `r`, as a parameter to functor `F`.

```
val r = ref NONE
functor F (type t
 val x: t) =
 struct
 val () = r := SOME x
 fun get () = valOf (!r)
 end
structure S1 = F (type t = unit -> unit
 val x = fn () => ())
structure S2 = F (type t = int
 val x = 13)
val () = S1.get () ()
```

MLton reports the following error.

```
Warning: z.sml 1.1.
 Unable to locally determine type of variable: r.
 type: ??? option ref
 in: val r = ref NONE
Error: z.sml 1.1.
 Type escapes the scope of its definition at z.sml 2.17.
 type: t
 in: val r = ref NONE
```

[PolyML 4.1.3](#) to seg faults, but [PolyML 5.4](#) reports the following error.

```
Warning- in 'z.sml', line 13.
The type of (r) contains a free type variable. Setting it to a unique
monotype.
Error- in 'z.sml', line 5.
Type error in function application.
 Function: := : _a option ref * _a option -> unit
 Argument: (r, SOME x) : _a option ref * t option
 Reason:
 Can't unify _a (*Constructed from a free type variable.*) with t
 (Different type constructors)
Found near r := SOME x
Error- in 'z.sml', line 12.
Type error in function application.
 Function: S1.get () : _a
```

```
Argument: () : unit
Reason: Value being applied does not have a function type
Found near S1.get () ()
```

---

## 49 DefinitionOfStandardML

The [Definition of Standard ML \(Revised\)](#) is a terse and formal specification of [Standard ML](#)'s syntax and semantics. The language specified by this book is often referred to as SML 97.

The [Definition of Standard ML](#) is an older version of the definition, published in 1990, which has an accompanying [Commentary on Standard ML](#) that introduces and explains the notation and approach. The same notation is used in the SML 97 definition, so it is worth purchasing the older definition and commentary if you intend a close study of the definition.

---

## 50 Defunctorize

[Defunctorize](#) is a translation pass from the [CoreML IntermediateLanguage](#) to the [XML IntermediateLanguage](#).

### 50.1 Description

This pass converts a [CoreML](#) program to an [XML](#) program by performing:

- linearization
- [MatchCompile](#)
- polymorphic `val` dec expansion
- datatype lifting (to the top-level)

### 50.2 Implementation

- [defunctorize.sig](#)
- [defunctorize.fun](#)

### 50.3 Details and Notes

This pass is grossly misnamed and does not perform defunctorization.

#### 50.3.1 Datatype Lifting

This pass moves all `datatype` declarations to the top level.

[Standard ML](#) `datatype` declarations can contain type variables that are not bound in the declaration itself. For example, the following program is valid.

```
fun 'a f (x: 'a) =
 let
 datatype 'b t = T of 'a * 'b
 val y: int t = T (x, 1)
 in
 13
 end
```

Unfortunately, the `datatype` declaration can not be immediately moved to the top level, because that would leave `'a` free.

```
datatype 'b t = T of 'a * 'b
fun 'a f (x: 'a) =
 let
 val y: int t = T (x, 1)
 in
 13
 end
```

In order to safely move `datatype`s`, this pass must close them, as well as add any free type variables as extra arguments to the type constructor. For example, the above program would be translated to the following.

```
datatype ('a, 'b) t = T of 'a * 'b
fun 'a f (x: 'a) =
 let
 val y: ('a * int) t = T (x, 1)
 in
 13
 end
```

## 50.4 Historical Notes

The [Defunctorize](#) pass originally eliminated [Standard ML](#) functors by duplicating their body at each application. These duties have been adopted by the [Elaborate](#) pass.

## 51 Developers

Here is a picture of the MLton team at a meeting in Chicago in August 2003. From left to right we have:

[StephenWeeks](#) — [MatthewFluet](#) — [HenryCejtin](#) — [SureshJagannathan](#)



Also see the [Credits](#) for a list of specific contributions.

### 51.1 Developers list

A number of people read the developers mailing list, [MLton-devel@mlton.org](mailto:MLton-devel@mlton.org), and make contributions there. Here's a list of those who have a page here.

- [AndreiFormiga](#)
- [JesperLouisAndersen](#)
- [JohnnyAndersen](#)
- [MichaelNorrish](#)
- [MikeThomas](#)
- [RayRacine](#)
- [WesleyTerpstra](#)
- [VesaKarvonen](#)

## 52 Development

This page is the central point for MLton development.

- Access the [Sources](#).
- Check the current [changelog](#) or recent [commits](#).
- Ideas for [Projects](#) to improve MLton.
- [Developers](#) that are or have been involved in the project.

### 52.1 Notes

- [CompilerOverview](#)
  - [CompilingWithSMLNJ](#)
  - [CrossCompiling](#)
  - [License](#)
  - [NeedsReview](#)
  - [PortingMLton](#)
  - [ReleaseChecklist](#)
  - [SelfCompiling](#)
-

## 53 Documentation

Documentation is available on the following topics.

- [Standard ML](#)
    - [Basis Library](#)
    - [Additional libraries](#)
  - [Installing MLton](#)
  - [Using MLton](#)
    - [Foreign function interface \(FFI\)](#)
    - [Manual page \(compile-time options run-time options\)](#)
    - [ML Basis system](#)
    - [MLton structure](#)
    - [Platform-specific notes](#)
    - [Profiling](#)
    - [Type checking](#)
    - [Help for porting from SML/NJ to MLton.](#)
  - [About MLton](#)
    - [Credits](#)
    - [Drawbacks](#)
    - [Features](#)
    - [History](#)
    - [License](#)
    - [Talk](#)
    - [WishList](#)
  - [Tools](#)
    - [MLLex \(mllex.pdf\)](#)
    - [MLYacc \(mlyacc.pdf\)](#)
    - [MLNLFFIGen \(mlyacc.pdf\)](#)
  - [References](#)
-

## 54 Drawbacks

MLton has several drawbacks due to its use of whole-program compilation.

- Large compile-time memory requirement.

Because MLton performs whole-program analysis and optimization, compilation requires a large amount of memory. For example, compiling MLton (over 140K lines) requires at least 512M RAM.

- Long compile times.

Whole-program compilation can take a long time. For example, compiling MLton (over 140K lines) on a 1.6GHz machine takes five to ten minutes.

- No interactive top level.

Because of whole-program compilation, MLton does not provide an interactive top level. In particular, it does not implement the optional [Basis Library](#) function `use`.

---

## 55 Eclipse

Eclipse is an open, extensible IDE.

ML-Dev is a plug-in for Eclipse, based on [SML/NJ](#).

There has been some talk on the MLton mailing list about adding support to Eclipse for MLton/SML, and in particular, using <http://eclipsefp.sourceforge.net/>. We are unaware of any progress along those lines.

---

## 56 Elaborate

[Elaborate](#) is a translation pass from the [AST IntermediateLanguage](#) to the [CoreML IntermediateLanguage](#).

### 56.1 Description

This pass performs type inference and type checking according to the [Definition](#). It also defunctorizes the program, eliminating all module-level constructs.

### 56.2 Implementation

- [elaborate.sig](#)
- [elaborate.fun](#)
- [elaborate-env.sig](#)
- [elaborate-env.fun](#)
- [elaborate-modules.sig](#)
- [elaborate-modules.fun](#)
- [elaborate-core.sig](#)
- [elaborate-core.fun](#)
- [elaborate](#)

### 56.3 Details and Notes

At the modules level, the [Elaborate](#) pass:

- elaborates signatures with interfaces (see [interface.sig](#) and [interface.fun](#))  
The main trick is to use disjoint sets to efficiently handle sharing of tycons and of structures and then to copy signatures as dags rather than as trees.
- checks functors at the point of definition, using functor summaries to speed up checking of functor applications.  
When a functor is first type checked, we keep track of the dummy argument structure and the dummy result structure, as well as all the tycons that were created while elaborating the body. Then, if we later need to type check an application of the functor (as opposed to defunctorize an application), we pair up tycons in the dummy argument structure with the actual argument structure and then replace the dummy tycons with the actual tycons in the dummy result structure, yielding the actual result structure. We also generate new tycons for all the tycons that we created while originally elaborating the body.
- handles opaque signature constraints.  
This is implemented by building a dummy structure realized from the signature, just as we would for a functor argument when type checking a functor. The dummy structure contains exactly the type information that is in the signature, which is what opacity requires. We then replace the variables (and constructors) in the dummy structure with the corresponding variables (and constructors) from the actual structure so that the translation to [CoreML](#) uses the right stuff. For each tycon in the dummy structure, we keep track of the corresponding type structure in the actual structure. This is used when producing the [CoreML](#) types (see `expandOpaque` in [type-env.sig](#) and [type-env.fun](#)).  
Then, within each `structure` or `functor` body, for each declaration (`<dec>` in the [Standard ML](#) grammar), the [Elaborate](#) pass does three steps:

1. [ScopeInference](#)
  2. – [PrecedenceParse](#)
-

- `_{ex, im}port` expansion
  - profiling insertion
  - unification
3. Overloaded {constant, function, record pattern} resolution

### 56.3.1 Defunctorization

The [Elaborate](#) pass performs a number of duties historically assigned to the [Defunctorize](#) pass.

As part of the [Elaborate](#) pass, all module level constructs (open, signature, structure, functor, long identifiers) are removed. This works because the [Elaborate](#) pass assigns a unique name to every type and variable in the program. This also allows the [Elaborate](#) pass to eliminate `local` declarations, which are purely for namespace management.

## 56.4 Examples

Here are a number of examples of elaboration.

- All variables bound in `val` declarations are renamed.

```
val x = 13
val y = x
```

```
val x_0 = 13
val y_0 = x_0
```

- All variables in `fun` declarations are renamed.

```
fun f x = g x
and g y = f y
```

```
fun f_0 x_0 = g_0 x_0
and g_0 y_0 = f_0 y_0
```

- Type abbreviations are removed, and the abbreviation is expanded wherever it is used.

```
type 'a u = int * 'a
type 'b t = 'b u * real
fun f (x : bool t) = x
```

```
fun f_0 (x_0 : (int * bool) * real) = x_0
```

- Exception declarations create a new constructor and rename the type.

```
type t = int
exception E of t * real
```

```
exception E_0 of int * real
```

- The type and value constructors in datatype declarations are renamed.

```
datatype t = A of int | B of real * t
```

```
datatype t_0 = A_0 of int | B_0 of real * t_0
```

- Local declarations are moved to the top-level. The environment keeps track of the variables in scope.

```

val x = 13
local val x = 14
in val y = x
end
val z = x

```

```

val x_0 = 13
val x_1 = 14
val y_0 = x_1
val z_0 = x_0

```

- Structure declarations are eliminated, with all declarations moved to the top level. Long identifiers are renamed.

```

structure S =
 struct
 type t = int
 val x : t = 13
 end
val y : S.t = S.x

```

```

val x_0 : int = 13
val y_0 : int = x_0

```

- Open declarations are eliminated.

```

val x = 13
val y = 14
structure S =
 struct
 val x = 15
 end
open S
val z = x + y

```

```

val x_0 = 13
val y_0 = 14
val x_1 = 15
val z_0 = x_1 + y_0

```

- Functor declarations are eliminated, and the body of a functor is duplicated wherever the functor is applied.

```

functor F(val x : int) =
 struct
 val y = x
 end
structure F1 = F(val x = 13)
structure F2 = F(val x = 14)
val z = F1.y + F2.y

```

```

val x_0 = 13
val y_0 = x_0
val x_1 = 14
val y_1 = x_1
val z_0 = y_0 + y_1

```

- Signature constraints are eliminated. Note that signatures do affect how subsequent variables are renamed.

```
val y = 13
structure S : sig
 val x : int
end =
 struct
 val x = 14
 val y = x
 end
open S
val z = x + y
```

```
val y_0 = 13
val x_0 = 14
val y_1 = x_0
val z_0 = x_0 + y_0
```

## 57 Emacs

### 57.1 SML modes

There are a few Emacs modes for SML.

- `sml-mode`
  - [http://www.xemacs.org/Documentation/packages/html/sml-mode\\_3.html](http://www.xemacs.org/Documentation/packages/html/sml-mode_3.html)
  - <http://www.smlnj.org/doc/Emacs/sml-mode.html>
  - <http://www.iro.umontreal.ca/%7Emonnier/elisp/>
- `mlton.el` contains the Emacs lisp that [Stephen Weeks](#) uses to interact with MLton (in addition to using `sml-mode`).
- <http://primate.net/%7Eitz/mindent.tar>, developed by Ian Zimmerman, who writes:

Unlike the widespread `sml-mode.el` it doesn't try to indent code based on ML syntax. I gradually got skeptical about this approach after writing the initial indentation support for `caml mode` and watching it bloat insanely as the language added new features. Also, any such attempts that I know of impose a particular coding style, or at best a choice among a limited set of styles, which I now oppose. Instead my mode is based on a generic package which provides manual bindable commands for common indentation operations (example: indent the current line under the n-th occurrence of a particular character in the previous non-blank line).

### 57.2 MLB modes

There is a mode for editing [ML Basis](#) files.

- `esml-mlb-mode.el` (plus other files)

### 57.3 Definitions and uses

There is a mode that supports the precise def-use information that MLton can output. It highlights definitions and uses and provides commands for navigation (e.g., `jump-to-def`, `jump-to-next`, `list-all-refs`). It can be handy, for example, for navigating in the MLton compiler source code. See [EmacsDefUseMode](#) for further information.

### 57.4 Building on the background

Tired of manually starting/stopping/restarting builds after editing files? Now you don't have to. See [EmacsBgBuildMode](#) for further information.

### 57.5 Error messages

MLton's error messages are not in the format that the Emacs `next-error` parser natively understands. There are a couple of ways to fix this. The easiest way is to add the following to your `.emacs` to cause Emacs to recognize MLton's error messages.

```
(require 'compile)
(add-to-list
 'compilation-error-regexp-alist
 ' ("^\\(Warning\\|Error\\): \\(.+\\) \\([0-9]+\\)\\.\\.\\([0-9]+\\)\\.\\.\\.$"
 2 3 4))
```

Alternatively, you could use a `sed` script to rewrite MLton's errors. Here is one such script:

```
sed -e 's/^\\([W|E].*\\): \\([^\]*\\) \\([0-9][0-9]*\\)\\.\\.\\([0-9][0-9]*\\)\\.\\.\/\2:\3:\1:\4/'
```

## 58 EmacsBgBuildMode

Do you really want to think about starting a build of your project? What if you had a personal slave that would restart a build of your project whenever you save any file belonging to that project? The bg-build mode does just that. Just save the file, a compile is started (silently!), you can continue working without even thinking about starting a build, and if there are errors, you are notified (with a message), and can then jump to errors.

This mode is not specific to MLton per se, but is particularly useful for working with MLton due to the longer compile times. By the time you start wondering about possible errors, the build is already on the way.

### 58.1 Functionality and Features

- Each time a file is saved, and after a user configurable delay period has been exhausted, a build is started silently in the background.
- When the build is finished, a status indicator (message) is displayed non-intrusively.
- At any time, you can switch to a build process buffer where all the messages from the build are shown.
- Optionally highlights (error/warning) message locations in (source code) buffers after a finished build.
- After a build has finished, you can jump to locations of warnings and errors from the build process buffer or by using the `first-error` and `next-error` commands.
- When a build fails, bg-build mode can optionally execute a user specified command. By default, bg-build mode executes `first-error`.
- When starting a build of a particular project, a possible previous live build of the same project is interrupted first.
- A project configuration file specifies the commands required to build a project.
- Multiple projects can be loaded into bg-build mode and bg-build mode can build a given maximum number of projects concurrently.
- Supports both [Gnu Emacs](#) and [XEmacs](#).

### 58.2 Download

There is no package for the mode at the moment. To install the mode you need to fetch the Emacs Lisp, `*.el`, files from the MLton repository: [emacs](#).

### 58.3 Setup

The easiest way to load the mode is to first tell Emacs where to find the files. For example, add

```
(add-to-list 'load-path (file-truename "path-to-the-el-files"))
```

to your `~/.emacs` or `~/.xemacs/init.el`. You'll probably also want to start the mode automatically by adding

```
(require 'bg-build-mode)
(bg-build-mode)
```

to your Emacs init file. Once the mode is activated, you should see the BGB indicator on the mode line.

### 58.3.1 MLton and Compilation-Mode

At the time of writing, neither Gnu Emacs nor XEmacs contain an error regexp that would match MLton's messages.

If you use Gnu Emacs, insert the following code into your `.emacs` file:

```
(require 'compile)
(add-to-list
 'compilation-error-regexp-alist
 ' ("^\(Warning\|Error\): \\.+\) \([0-9]+\)\. \([0-9]+\)\. $"
 2 3 4))
```

If you use XEmacs, insert the following code into your `init.el` file:

```
(require 'compile)
(add-to-list
 'compilation-error-regexp-alist-alist
 '(mlton
 ("^\(Warning\|Error\): \\.+\) \([0-9]+\)\. \([0-9]+\)\. $"
 2 3 4)))
(compilation-build-compilation-error-regexp-alist)
```

## 58.4 Usage

Typically projects are built (or compiled) using a tool like `make`, but the details vary. The `bg-build` mode needs a project configuration file to know how to build your project. A project configuration file basically contains an Emacs Lisp expression calling a function named `bg-build` that returns a project object. A simple example of a project configuration file would be the (`Build.bgb`) file used with `smlbot`:

```
(bg-build
 :name "SML-Bot"
 :shell "nice -n5 make all")
```

The `bg-build` function takes a number of keyword arguments:

- `:name` specifies the name of the project. This can be any expression that evaluates to a string or to a nullary function that returns a string.
- `:shell` specifies a shell command to execute. This can be any expression that evaluates to a string, a list of strings, or to a nullary function returning a list of strings.
- `:build?` specifies a predicate to determine whether the project should be built after some files have been modified. The predicate is given a list of filenames and should return a non-nil value when the project should be built and nil otherwise.

All of the keyword arguments, except `:shell`, are optional and can be left out.

Note the use of the `nice` command above. It means that background build process is given a lower priority by the system process scheduler. Assuming your machine has enough memory, using `nice` ensures that your computer remains responsive. (You probably won't even notice when a build is started.)

Once you have written a project file for `bg-build` mode. Use the `bg-build-add-project` command to load the project file for `bg-build` mode. The `bg-build` mode can also optionally load recent project files automatically at startup.

After the project file has been loaded and `bg-build` mode activated, each time you save a file in Emacs, the `bg-build` mode tries to build your project.

The `bg-build-status` command creates a buffer that displays some status information on builds and allows you to manage projects (start builds explicitly, remove a project from `bg-build`, ...) as well as visit buffers created by `bg-build`. Notice the count of started builds. At the end of the day it can be in the hundreds or thousands. Imagine the number of times you've been relieved of starting a build explicitly!

## 59 EmacsDefUseMode

MLton provides an [option](#), `-show-def-use file`, to output precise (giving exact source locations) and accurate (including all uses and no false data) whole-program def-use information to a file. Unlike typical tags facilities, the information includes local variables and distinguishes between different definitions even when they have the same name. The def-use Emacs mode uses the information to provide navigation support, which can be particularly useful while reading SML programs compiled with MLton (such as the MLton compiler itself).

### 59.1 Screen Capture

Note the highlighting and the type displayed in the minibuffer.

```

emacs@localhost.localdomain
File Edit Options Buffers Tools SML Help

(* First a plain old type rep for our data; *)
val t1 = iso (record (R' "id" int
 *` R' "name" string))
 (fn {id = a, name = b} => a & b,
 fn a & b => {id = a, name = b})

(* Then we assign version {1} to the type; *)
val t = versioned $ 1 t1

val pickleV1 = pickle t

(* Then a plain old type rep for our new data; *)
val t2 = iso (record (R' "id" int
 *` R' "extra" bool
 *` R' "name" string))
 (fn {id = a, extra = b, name = c} => a & b & c,
 fn a & b & c => {id = a, extra = b, name = c})

(* Then we assign version {2} to the new type, keeping the
 * version {1} for the old type; *)
val t = versioned (version 1 t1
 (fn {id, name} =>
 {id = id, extra = false, name = name}))
 $ 2 t2

(* Note that the original versioned {t} is no longer needed.
 * In an actual program, you would have just edited the
 * original definition instead of introducing a new one.
 * However, the old type rep is required if you wish to be
 * able to unpickle old versions. *)
in
 thatEq t {expect = {id = 1, extra = false, name = "whatever"},
 actual = unpickle t
 (pickleV1 {id = 1, name = "whatever"})}
; thatEq t {expect = {id = 3, extra = true, name = "whenever"},
 actual = unpickle t (pickle t {id = 3, extra = true,
 name = "whenever"})}

--:-- pickle.sml 28% (99,41) SVN:6391 (SML BGB DU)
{id: int, name: string} -> string

```

### 59.2 Features

- Highlights definitions and uses. Different colors for definitions, unused definitions, and uses.
- Shows types (with highlighting) of variable definitions in the minibuffer.

- Navigation: `jump-to-def`, `jump-to-next`, and `jump-to-prev`. These work precisely (no searching involved).
- Can list, visit and mark all references to a definition (within a program).
- Automatically reloads updated def-use files.
- Automatically loads previously used def-use files at startup.
- Supports both [Gnu Emacs](#) and [XEmacs](#).

### 59.3 Download

There is no separate package for the def-use mode although the mode has been relatively stable for some time already. To install the mode you need to get the Emacs Lisp, `*.el`, files from MLton's repository: [emacs](#). The easiest way to get the files is to use [Git](#) to access MLton's [sources](#).

### 59.4 Setup

The easiest way to load def-use mode is to first tell Emacs where to find the files. For example, add

```
(add-to-list 'load-path (file-truename "path-to-the-el-files"))
```

to your `~/ .emacs` or `~/ .xemacs/init.el`. You'll probably also want to start `def-use-mode` automatically by adding

```
(require 'esml-du-mlton)
(def-use-mode)
```

to your Emacs init file. Once the def-use mode is activated, you should see the DU indicator on the mode line.

### 59.5 Usage

To use def-use mode one typically first sets up the program's makefile or build script so that the def-use information is saved each time the program is compiled. In addition to the `-show-def-use file` option, the `-prefer-abs-paths true` expert option is required. Note that the time it takes to save the information is small (compared to type-checking), so it is recommended to simply add the options to the MLton invocation that compiles the program. However, it is only necessary to type check the program (or library), so one can specify the `-stop tc` option. For example, suppose you have a program defined by an MLB file named `my-prg.mlb`, you can save the def-use information to the file `my-prg.du` by invoking MLton as:

```
mlton -prefer-abs-paths true -show-def-use my-prg.du -stop tc my-prg.mlb
```

Finally, one needs to tell the mode where to find the def-use information. This is done with the `esml-du-mlton` command. For example, to load the `my-prg.du` file, one would type:

```
M-x esml-du-mlton my-prg.du
```

After doing all of the above, find an SML file covered by the previously saved and loaded def-use information, and place the cursor at some variable (definition or use, it doesn't matter). You should see the variable being highlighted. (Note that specifications in signatures do not define variables.)

You might also want to setup and use the [Bg-Build mode](#) to start builds automatically.

### 59.6 Types

`-show-def-use` output was extended to include types of variable definitions in revision [r6333](#). To get good type names, the types must be in scope at the end of the program. If you are using the [ML Basis](#) system, this means that the root MLB-file for your application should not wrap the libraries used in the application inside `local ... in ... end`, because that would remove them from the scope before the end of the program.

## 60 Enscript

**GNU Enscript** converts ASCII files to PostScript, HTML, and other output languages, applying language sensitive highlighting (similar to Emacs's font lock mode). Here are a few *states* files for highlighting [Standard ML](#).

- `sml_simple.st` — Provides highlighting of keywords, string and character constants, and (nested) comments.
- `sml_verbose.st` — Supersedes the above, adding highlighting of numeric constants. Due to the limited parsing available, numeric record labels are highlighted as numeric constants, in all contexts. Likewise, a binding precedence separated from `infix` or `infixr` by a newline is highlighted as a numeric constant and a numeric record label selector separated from `#` by a newline is highlighted as a numeric constant.
- `sml_fancy.st` — Supersedes the above, adding highlighting of type and constructor bindings, highlighting of explicit binding of type variables at `val` and `fun` declarations, and separate highlighting of core and modules level keywords. Due to the limited parsing available, it is assumed that the input is a syntactically correct, top-level declaration.
- `sml_gaudy.st` — Supersedes the above, adding highlighting of type annotations, in both expressions and signatures. Due to the limited parsing available, it is assumed that the input is a syntactically correct, top-level declaration.

### 60.1 Install and use

- Version 1.6.3 of **GNU Enscript**
  - Copy all files to `/usr/share/enscript/hl/` or `.enscript/` in your home directory.
  - Invoke `enscript` with `--highlight=sml_simple` (or `--highlight=sml_verbose` or `--highlight=sml_fancy` or `--highlight=sml_gaudy`).
- Version 1.6.1 of **GNU Enscript**
  - Append `sml_all.st` to `/usr/share/enscript/enscript.st`
  - Invoke `enscript` with `--pretty-print=sml_simple` (or `--pretty-print=sml_verbose` or `--pretty-print=sml_fancy` or `--pretty-print=sml_gaudy`).

### 60.2 Feedback

Comments and suggestions should be directed to [MatthewFluet](#).

## 61 EqualityType

An equality type is a type to which [PolymorphicEquality](#) can be applied. The [Definition](#) and the [Basis Library](#) precisely spell out which types are equality types.

- `bool`, `char`, `IntInf.int`, `Int<N>.int`, `string`, and `Word<N>.word` are equality types.
- for any `t`, both `t array` and `t ref` are equality types.
- if `t` is an equality type, then `t list`, and `t vector` are equality types.
- if `t1, ..., tn` are equality types, then `t1 * ... * tn` and `{l1: t1, ..., ln: tn}` are equality types.
- if `t1, ..., tn` are equality types and `t` [AdmitsEquality](#), then `(t1, ..., tn) t` is an equality type.

To check that a type `t` is an equality type, use the following idiom.

```
structure S: sig eqtype t end =
 struct
 type t = ...
 end
```

Notably, `exn` and `real` are not equality types. Neither is `t1 -> t2`, for any `t1` and `t2`.

Equality on arrays and ref cells is by identity, not structure. For example, `ref 13 = ref 13` is false. On the other hand, equality for lists, strings, and vectors is by structure, not identity. For example, the following equalities hold.

```
val _ = [1, 2, 3] = 1 :: [2, 3]
val _ = "foo" = concat ["f", "o", "o"]
val _ = Vector.fromList [1, 2, 3] = Vector.tabulate (3, fn i => i + 1)
```

## 62 EqualityTypeVariable

An equality type variable is a type variable that starts with two or more primes, as in `''a` or `''b`. The canonical use of equality type variables is in specifying the type of the [PolymorphicEquality](#) function, which is `''a * ''a -> bool`. Equality type variables ensure that polymorphic equality is only used on [equality types](#), by requiring that at every use of a polymorphic value, equality type variables are instantiated by equality types.

For example, the following program is type correct because polymorphic equality is applied to variables of type `''a`.

```
fun f (x: ''a, y: ''a): bool = x = y
```

On the other hand, the following program is not type correct, because polymorphic equality is applied to variables of type `'a`, which is not an equality type.

```
fun f (x: 'a, y: 'a): bool = x = y
```

MLton reports the following error, indicating that polymorphic equality expects equality types, but didn't get them.

```
Error: z.sml 1.32.
Function applied to incorrect argument.
 expects: [<equality>] * [<equality>]
 but got: [<non-equality>] * [<non-equality>]
 in: = (x, y)
```

As an example of using such a function that requires equality types, suppose that `f` has polymorphic type `''a -> unit`. Then, `f 13` is type correct because `int` is an equality type. On the other hand, `f 13.0` and `f (fn x => x)` are not type correct, because `real` and arrow types are not equality types. We can test these facts with the following short programs. First, we verify that such an `f` can be applied to integers.

```
functor Ok (val f: ''a -> unit): sig end =
 struct
 val () = f 13
 val () = f 14
 end
```

We can do better, and verify that such an `f` can be applied to any integer.

```
functor Ok (val f: ''a -> unit): sig end =
 struct
 fun g (x: int) = f x
 end
```

Even better, we don't need to introduce a dummy function name; we can use a type constraint.

```
functor Ok (val f: ''a -> unit): sig end =
 struct
 val _ = f: int -> unit
 end
```

Even better, we can use a signature constraint.

```
functor Ok (S: sig val f: ''a -> unit end):
 sig val f: int -> unit end = S
```

This functor concisely verifies that a function of polymorphic type `''a -> unit` can be safely used as a function of type `int -> unit`.

As above, we can verify that such an `f` can not be used at non equality types.

```
functor Bad (S: sig val f: ''a -> unit end):
 sig val f: real -> unit end = S
```

```
functor Bad (S: sig val f: 'a -> unit end):
 sig val f: ('a -> 'a) -> unit end = S
```

For each of these programs, MLton reports the following error.

```
Error: z.sml 2.4.
 Variable type in structure disagrees with signature.
 variable: f
 structure: [<equality>] -> _
 signature: [<non-equality>] -> _
```

## 62.1 Equality type variables in type and datatype declarations

Equality type variables can be used in type and datatype declarations; however they play no special role. For example,

```
type 'a t = 'a * int
```

is completely identical to

```
type ''a t = ''a * int
```

In particular, such a definition does *not* require that `t` only be applied to equality types.

Similarly,

```
datatype 'a t = A | B of 'a
```

is completely identical to

```
datatype ''a t = A | B of ''a
```

## 63 EtaExpansion

Eta expansion is a simple syntactic change used to work around the [ValueRestriction](#) in [Standard ML](#).

The eta expansion of an expression  $e$  is the expression  $\text{fn } z \Rightarrow e\ z$ , where  $z$  does not occur in  $e$ . This only makes sense if  $e$  denotes a function, i.e. is of arrow type. Eta expansion delays the evaluation of  $e$  until the function is applied, and will re-evaluate  $e$  each time the function is applied.

The name "eta expansion" comes from the eta-conversion rule of the [lambda calculus](#). Expansion refers to the directionality of the equivalence being used, namely taking  $e$  to  $\text{fn } z \Rightarrow e\ z$  rather than  $\text{fn } z \Rightarrow e\ z$  to  $e$  (eta contraction).

---

## 64 eXene

**eXene** is a multi-threaded X Window System toolkit written in [ConcurrentML](#).

There is a group at K-State working toward **eXene 2.0**.

---

## 65 Experimental

This page is for experimental releases of MLton. These versions are not as well tested as our [public releases](#), and may not be available for our all our usual platforms.

---

## 66 FAQ

Feel free to ask questions and to update answers by editing this page. Since we try to make as much information as possible available on the web site and we like to avoid duplication, many of the answers are simply links to a web page that answers the question.

### 66.1 How do you pronounce MLton?

[Pronounce](#)

### 66.2 What SML software has been ported to MLton?

[Libraries](#)

### 66.3 What graphical libraries are available for MLton?

[Libraries](#)

### 66.4 How does MLton's performance compare to other SML compilers and to other languages?

MLton has [excellent performance](#).

### 66.5 Does MLton treat monomorphic arrays and vectors specially?

MLton implements monomorphic arrays and vectors (e.g. `BoolArray`, `Word8Vector`) exactly as instantiations of their polymorphic counterpart (e.g. `bool array`, `Word8.word vector`). Thus, there is no need to use the monomorphic versions except when required to interface with the [Basis Library](#) or for portability with other SML implementations.

### 66.6 Why do I get a Segfault/Bus error in a program that uses `IntInf/LargeInt` to calculate numbers with several hundred thousand digits?

[GnuMP](#)

### 66.7 How can I decrease compile-time memory usage?

- Compile with `-verbose 3` to find out if the problem is due to an SSA optimization pass. If so, compile with `-drop-pass pass` to skip that pass.
- Compile with `@MLton hash-cons 0.5 --`, which will instruct the runtime to hash cons the heap every other GC.
- Compile with `-polyvariance false`, which is an undocumented option that causes less code duplication.

Also, please [Contact](#) us to let us know the problem to help us better understand MLton's limitations.

### 66.8 How portable is SML code across SML compilers?

[StandardMLPortability](#)

---

## 67 Features

MLton has the following features.

### 67.1 Portability

- Runs on a variety of platforms.
  - [ARM](#):
    - \* [Linux](#) (Debian)
  - [Alpha](#):
    - \* [Linux](#) (Debian)
  - [AMD64](#):
    - \* [Darwin](#) (Mac OS X)
    - \* [FreeBSD](#)
    - \* [Linux](#) (Debian, Fedora, ...)
    - \* [Solaris](#) (10 and above)
  - [HPPA](#):
    - \* [HPUX](#) (11.11 and above)
    - \* [Linux](#) (Debian)
  - [IA64](#):
    - \* [HPUX](#) (11.11 and above)
    - \* [Linux](#) (Debian)
  - [PowerPC](#):
    - \* [AIX](#) (5.2 and above)
    - \* [Darwin](#) (Mac OS X)
    - \* [Linux](#) (Debian, Fedora)
  - [PowerPC64](#):
    - \* [AIX](#) (5.2 and above)
  - [S390](#)
    - \* [Linux](#) (Debian)
  - [Sparc](#)
    - \* [Linux](#) (Debian)
    - \* [Solaris](#) (8 and above)
  - [X86](#):
    - \* [Cygwin/Windows](#)
    - \* [Darwin](#) (Mac OS X)
    - \* [FreeBSD](#)
    - \* [Linux](#) (Debian, Fedora, ...)
    - \* [MinGW/Windows](#)
    - \* [NetBSD](#)
    - \* [OpenBSD](#)
    - \* [Solaris](#) (10 and above)

## 67.2 Robustness

- Supports the full SML 97 language as given in [The Definition of Standard ML \(Revised\)](#).  
If there is a program that is valid according to the [Definition](#) that is rejected by MLton, or a program that is invalid according to the [Definition](#) that is accepted by MLton, it is a bug. For a list of known bugs, see [UnresolvedBugs](#).
- A complete implementation of the [Basis Library](#).  
MLton's implementation matches latest [Basis Library specification](#), and includes a complete implementation of all the required modules, as well as many of the optional modules.
- Generates standalone executables.  
No additional code or libraries are necessary in order to run an executable, except for the standard shared libraries. MLton can also generate statically linked executables.
- Compiles large programs.  
MLton is sufficiently efficient and robust that it can compile large programs, including itself (over 140K lines). The distributed version of MLton was compiled by MLton.
- Support for large amounts of memory (up to 4G on 32-bit systems; more on 64-bit systems).
- Support for large array lengths (up to  $2^{31}-1$  on 32-bit systems; up to  $2^{63}-1$  on 64-bit systems).
- Support for large files, using 64-bit file positions.

## 67.3 Performance

- Executables have [excellent running times](#).
- Generates small executables.  
MLton takes advantage of whole-program compilation to perform very aggressive dead-code elimination, which often leads to smaller executables than with other SML compilers.
- Untagged and unboxed native integers, reals, and words.  
In MLton, integers and words are 8 bits, 16 bits, 32 bits, and 64 bits and arithmetic does not have any overhead due to tagging or boxing. Also, reals (32-bit and 64-bit) are stored unboxed, avoiding any overhead due to boxing.
- Unboxed native arrays.  
In MLton, an array (or vector) of integers, reals, or words uses the natural C-like representation. This is fast and supports easy exchange of data with C. Monomorphic arrays (and vectors) use the same C-like representations as their polymorphic counterparts.
- Multiple [garbage collection](#) strategies.
- Fast arbitrary precision arithmetic (`IntInf`) based on the [GnuMP](#).  
For `IntInf` intensive programs, MLton can be an order of magnitude or more faster than Poly/ML or SML/NJ.

## 67.4 Tools

- Source-level [Profiling](#) of both time and allocation.
  - [MLLex](#) lexer generator
  - [MLYacc](#) parser generator
  - [MLNLFFIGen](#) foreign-function-interface generator
-

## 67.5 Extensions

- A simple and fast C [ForeignFunctionInterface](#) that supports calling from SML to C and from C to SML.
  - The [ML Basis system](#) for programming in the very large, separate delivery of library sources, and more.
  - A number of extension libraries that provide useful functionality that cannot be implemented with the [Basis Library](#). See below for an overview and [MLtonStructure](#) for details.
    - [continuations](#)  
MLton supports continuations via `callcc` and `throw`.
    - [finalization](#)  
MLton supports finalizable values of arbitrary type.
    - [interval timers](#)  
MLton supports the functionality of the C `setitimer` function.
    - [random numbers](#)  
MLton has functions similar to the C `rand` and `srand` functions, as well as support for access to `/dev/random` and `/dev/urandom`.
    - [resource limits](#)  
MLton has functions similar to the C `getrlimit` and `setrlimit` functions.
    - [resource usage](#)  
MLton supports a subset of the functionality of the C `getrusage` function.
    - [signal handlers](#)  
MLton supports signal handlers written in SML. Signal handlers run in a separate MLton thread, and have access to the thread that was interrupted by the signal. Signal handlers can be used in conjunction with threads to implement preemptive multitasking.
    - [size primitive](#)  
MLton includes a primitive that returns the size (in bytes) of any object. This can be useful in understanding the space behavior of a program.
    - [system logging](#)  
MLton has a complete interface to the C `syslog` function.
    - [threads](#)  
MLton has support for its own threads, upon which either preemptive or non-preemptive multitasking can be implemented. MLton also has support for [Concurrent ML \(CML\)](#).
    - [weak pointers](#)  
MLton supports weak pointers, which allow the garbage collector to reclaim objects that it would otherwise be forced to keep. Weak pointers are also used to provide finalization.
    - [world save and restore](#)  
MLton has a facility for saving the entire state of a computation to a file and restarting it later. This facility can be used for staging and for checkpointing computations. It can even be used from within signal handlers, allowing interrupt driven checkpointing.
-

## 68 FirstClassPolymorphism

First-class polymorphism is the ability to treat polymorphic functions just like other values: pass them as arguments, store them in data structures, etc. Although [Standard ML](#) does have polymorphic functions, it does not support first-class polymorphism.

For example, the following declares and uses the polymorphic function `id`.

```
val id = fn x => x
val _ = id 13
val _ = id "foo"
```

If SML supported first-class polymorphism, we could write the following.

```
fun useId id = (id 13; id "foo")
```

However, this does not type check. MLton reports the following error.

```
Error: z.sml 1.24.
 Function applied to incorrect argument.
 expects: [int]
 but got: [string]
 in: id "foo"
```

The error message arises because MLton infers from `id 13` that `id` accepts an integer argument, but that `id "foo"` is passing a string.

Using explicit types sheds some light on the problem.

```
fun useId (id: 'a -> 'a) = (id 13; id "foo")
```

On this, MLton reports the following errors.

```
Error: z.sml 1.29.
 Function applied to incorrect argument.
 expects: ['a]
 but got: [int]
 in: id 13
Error: z.sml 1.36.
 Function applied to incorrect argument.
 expects: ['a]
 but got: [string]
 in: id "foo"
```

The errors arise because the argument `id` is *not* polymorphic; rather, it is monomorphic, with type `'a -> 'a`. It is perfectly valid to apply `id` to a value of type `'a`, as in the following

```
fun useId (id: 'a -> 'a, x: 'a) = id x (* type correct *)
```

So, what is the difference between the type specification on `id` in the following two declarations?

```
val id: 'a -> 'a = fn x => x
fun useId (id: 'a -> 'a) = (id 13; id "foo")
```

While the type specifications on `id` look identical, they mean different things. The difference can be made clearer by explicitly [scoping the type variables](#).

```
val 'a id: 'a -> 'a = fn x => x
fun 'a useId (id: 'a -> 'a) = (id 13; id "foo") (* type error *)
```

In `val 'a id`, the type variable scoping means that for any `'a`, `id` has type `'a -> 'a`. Hence, `id` can be applied to arguments of type `int`, `real`, etc. Similarly, in `fun 'a useId`, the scoping means that `useId` is a polymorphic function that for any `'a` takes a function of type `'a -> 'a` and does something. Thus, `useId` could be applied to a function of type `int -> int`, `real -> real`, etc.

One could imagine an extension of SML that allowed scoping of type variables at places other than `fun` or `val` declarations, as in the following.

```
fun useId (id: ('a). 'a -> 'a) = (id 13; id "foo") (* not SML *)
```

Such an extension would need to be thought through very carefully, as it could cause significant complications with [TypeInference](#), possible even undecidability.

## 69 Fixpoints

This page discusses a framework that makes it possible to compute fixpoints over arbitrary products of abstract types. The code is from an Extended Basis library ([README](#)).

First the signature of the framework ([tie.sig](#)):

```
(**
 * A framework for computing fixpoints.
 *
 * In a strict language you sometimes want to provide a fixpoint
 * combinator for an abstract type {t} to make it possible to write
 * recursive definitions. Unfortunately, a single combinator {fix} of the
 * type {(t -> t) -> t} does not support mutual recursion. To support
 * mutual recursion, you would need to provide a family of fixpoint
 * combinators having types of the form {(u -> u) -> u} where {u} is a
 * type of the form {t * ... * t}. Unfortunately, even such a family of
 * fixpoint combinators does not support mutual recursion over different
 * abstract types.
 *)
signature TIE = sig
 include ETAEXP'
 type 'a t = 'a etaexp
 (** The type of fixpoint witnesses. *)

 val fix : 'a t -> 'a Fix.t
 (**
 * Produces a fixpoint combinator from the given witness. For example,
 * one can make a mutually recursive definition of functions:
 *
 *> val isEven & isOdd =
 *> let open Tie in fix (function *` function) end
 *> (fn isEven & isOdd =>
 *> (fn 0 => true
 *> | 1 => false
 *> | n => isOdd (n-1)) &
 *> (fn 0 => false
 *> | 1 => true
 *> | n => isEven (n-1)))
 *)

 (** == Making New Witnesses == *)

 val pure : ('a * 'a UnOp.t) Thunk.t -> 'a t
 (**
 * {pure} is a more general version of {tier}. It is mostly useful for
 * computing fixpoints in a non-imperative manner.
 *)

 val tier : ('a * 'a Effect.t) Thunk.t -> 'a t
 (**
 * {tier} is used to define fixpoint witnesses for new abstract types
 * by providing a thunk whose instantiation allocates a mutable proxy
 * and a procedure for updating it with the result.
 *)

 val id : 'a -> 'a t
 (** {id x} is equivalent to {pure (const (x, id))}. *)

 (** == Combining Existing Witnesses == *)

 val iso : 'b t -> ('a, 'b) Iso.t -> 'a t
```

```

(**
 * Given an isomorphism between {'a} and {'b} and a witness for {'b},
 * produces a witness for {'a}. This is useful when you have a new
 * type that is isomorphic to some old type for which you already have
 * a witness.
 *)

val product : 'a t * ('a -> 'b t) -> ('a, 'b) Product.t t
(**
 * Dependent product combinator. Given a witness for {'a} and a
 * constructor from a {'a} to witness for {'b}, produces a witness for
 * the product {'a, 'b} Product.t. The constructor for {'b} should
 * not access the (proxy) value {'a} before it has been fixed.
 *)

val `*` : 'a t * 'b t -> ('a, 'b) Product.t t
(** {a * ` b} is equivalent to {product (a, const b)}. *)

val tuple2 : 'a t * 'b t -> ('a * 'b) t
(**
 * Given witnesses for {'a} and {'b} produces a witness for the product
 * {'a * 'b}.
 *)

(** == Particular Witnesses == *)

val function : ('a -> 'b) t
(** Witness for functions. *)
end

```

`fix` is a [type-indexed](#) function. The type-index parameter to `fix` is called a "witness". To compute fixpoints over products, one uses the ``*` operator to combine witnesses. To provide a fixpoint combinator for an abstract type, one implements a witness providing a thunk whose instantiation allocates a fresh, mutable proxy and a procedure for updating the proxy with the solution. Naturally this means that not all possible ways of computing a fixpoint of a particular type are possible under the framework. The `pure` combinator is a generalization of `tier`. The `iso` combinator is provided for reusing existing witnesses.

Note that instead of using an infix operator, we could alternatively employ an interface using [Fold](#). Also, witnesses are eta-expanded to work around the [value restriction](#), while maintaining abstraction.

Here is the implementation ([tie.sml](#)):

```

structure Tie :> TIE = struct
 open Product
 infix &
 type 'a etaexp_dom = Unit.t
 type 'a etaexp_cod = ('a * 'a UnOp.t) Thunk.t
 type 'a etaexp = 'a etaexp_dom -> 'a etaexp_cod
 type 'a t = 'a etaexp
 fun fix aT f = let val (a, ta) = aT () () in ta (f a) end
 val pure = Thunk.mk
 fun iso bT (iso as (_, b2a)) () () = let
 val (b, fB) = bT () ()
 in
 (b2a b, Fn.map iso fB)
 end
 fun product (aT, a2bT) () () = let
 val (a, fA) = aT () ()
 val (b, fB) = a2bT a () ()
 in
 (a & b, Product.map (fA, fB))
 end
 (* The rest are not primitive operations. *)
end

```

```

fun op *` (aT, bT) = product (aT, Fn.const bT)
fun tuple2 ab = iso (op *` ab) Product.isoTuple2
fun tier th = pure ((fn (a, ua) => (a, Fn.const a o ua)) o th)
fun id x = pure (Fn.const (x, Fn.id))
fun function ? =
 pure (fn () => let
 val r = ref (Basic.raising Fix.Fix)
 in
 (fn x => !r x, fn f => (r := f ; f))
 end) ?
end

```

Let's then take a look at a couple of additional examples.

Here is a naive implementation of lazy promises:

```

structure Promise :> sig
 type 'a t
 val lazy : 'a Thunk.t -> 'a t
 val force : 'a t -> 'a
 val Y : 'a t Tie.t
end = struct
 datatype 'a t' =
 EXN of exn
 | THUNK of 'a Thunk.t
 | VALUE of 'a
 type 'a t = 'a t' Ref.t
 fun lazy f = ref (THUNK f)
 fun force t =
 case !t
 of EXN e => raise e
 | THUNK f => (t := VALUE (f ()) handle e => t := EXN e ; force t)
 | VALUE v => v
 fun Y ? = Tie.tier (fn () => let
 val r = lazy (raising Fix.Fix)
 in
 (r, r <\ op := o !)
 end) ?
end

```

An example use of our naive lazy promises is to implement equally naive lazy streams:

```

structure Stream :> sig
 type 'a t
 val cons : 'a * 'a t -> 'a t
 val get : 'a t -> ('a * 'a t) Option.t
 val Y : 'a t Tie.t
end = struct
 datatype 'a t = IN of ('a * 'a t) Option.t Promise.t
 fun cons (x, xs) = IN (Promise.lazy (fn () => SOME (x, xs)))
 fun get (IN p) = Promise.force p
 fun Y ? = Tie.iso Promise.Y (fn IN p => p, IN) ?
end

```

Note that above we make use of the `iso` combinator. Here is a finite representation of an infinite stream of ones:

```

val ones = let
 open Tie Stream
in
 fix Y (fn ones => cons (1, ones))
end

```

## 70 Flatten

[Flatten](#) is an optimization pass for the [SSA IntermediateLanguage](#), invoked from [SSASimplify](#).

### 70.1 Description

This pass flattens arguments to [SSA](#) constructors, blocks, and functions.

If a tuple is explicitly available at all uses of a function (resp. block), then:

- The formals and call sites are changed so that the components of the tuple are passed.
- The tuple is reconstructed at the beginning of the body of the function (resp. block).

Similarly, if a tuple is explicitly available at all uses of a constructor, then:

- The constructor argument datatype is changed to flatten the tuple type.
- The tuple is passed flat at each `ConApp`.
- The tuple is reconstructed at each `Case` transfer target.

### 70.2 Implementation

- `flatten.fun`

### 70.3 Details and Notes

---

## 71 Fold

This page describes a technique that enables convenient syntax for a number of language features that are not explicitly supported by [Standard ML](#), including: variable number of arguments, [optional arguments and labeled arguments](#), [array and vector literals](#), [functional record update](#), and (seemingly) dependently typed functions like [printf](#) and [scanf](#).

The key idea to *fold* is to define functions `fold`, `step0`, and `$` such that the following equation holds.

```
fold (a, f) (step0 h1) (step0 h2) ... (step0 hn) $
= f (hn (... (h2 (h1 a))))
```

The name `fold` comes because this is like a traditional list fold, where `a` is the *base element*, and each *step function*, `step0 hi`, corresponds to one element of the list and does one step of the fold. The name `$` is chosen to mean "end of arguments" from its common use in regular-expression syntax.

Unlike the usual list fold in which the same function is used to step over each element in the list, this fold allows the step functions to be different from each other, and even to be of different types. Also unlike the usual list fold, this fold includes a "finishing function", `f`, that is applied to the result of the fold. The presence of the finishing function may seem odd because there is no analogy in list fold. However, the finishing function is essential; without it, there would be no way for the folder to perform an arbitrary computation after processing all the arguments. The examples below will make this clear.

The functions `fold`, `step0`, and `$` are easy to define.

```
fun $ (a, f) = f a
fun id x = x
structure Fold =
 struct
 fun fold (a, f) g = g (a, f)
 fun step0 h (a, f) = fold (h a, f)
 end
```

We've placed `fold` and `step0` in the `Fold` structure but left `$` at the toplevel because it is convenient in code to always have `$` in scope. We've also defined the identity function, `id`, at the toplevel since we use it so frequently.

Plugging in the definitions, it is easy to verify the equation from above.

```
fold (a, f) (step0 h1) (step0 h2) ... (step0 hn) $
= step0 h1 (a, f) (step0 h2) ... (step0 hn) $
= fold (h1 a, f) (step0 h2) ... (step0 hn) $
= step0 h2 (h1 a, f) ... (step0 hn) $
= fold (h2 (h1 a), f) ... (step0 hn) $
...
= fold (hn (... (h2 (h1 a))), f) $
= $ (hn (... (h2 (h1 a))), f)
= f (hn (... (h2 (h1 a))))
```

### 71.1 Example: variable number of arguments

The simplest example of fold is accepting a variable number of (curried) arguments. We'll define a function `f` and argument `a` such that all of the following expressions are valid.

```
f $
f a $
f a a $
f a a a $
f a a a ... a a a $ (* as many a's as we want *)
```

Off-hand it may appear impossible that all of the above expressions are type correct SML—how can a function `f` accept a variable number of curried arguments? What could the type of `f` be? We'll have more to say later on how type checking works.

For now, once we have supplied the definitions below, you can check that the expressions are type correct by feeding them to your favorite SML implementation.

It is simple to define `f` and `a`. We define `f` as a folder whose base element is `()` and whose finish function does nothing. We define `a` as the step function that does nothing. The only trickiness is that we must [eta expand](#) the definition of `f` and `a` to work around the ValueRestriction; we frequently use eta expansion for this purpose without mention.

```
val base = ()
fun finish () = ()
fun step () = ()
val f = fn z => Fold.fold (base, finish) z
val a = fn z => Fold.step0 step z
```

One can easily apply the fold equation to verify by hand that `f` applied to any number of `a`'s evaluates to `()`.

```
f a ... a $
= finish (step (... (step base)))
= finish (step (... ()))
...
= finish ()
= ()
```

## 71.2 Example: variable-argument sum

Let's look at an example that computes something: a variable-argument function `sum` and a stepper `a` such that

```
sum (a i1) (a i2) ... (a im) $ = i1 + i2 + ... + im
```

The idea is simple—the folder starts with a base accumulator of 0 and the stepper adds each element to the accumulator, `s`, which the folder simply returns at the end.

```
val sum = fn z => Fold.fold (0, fn s => s) z
fun a i = Fold.step0 (fn s => i + s)
```

Using the fold equation, one can verify the following.

```
sum (a 1) (a 2) (a 3) $ = 6
```

## 71.3 Step1

It is sometimes syntactically convenient to omit the parentheses around the steps in a fold. This is easily done by defining a new function, `step1`, as follows.

```
structure Fold =
 struct
 open Fold
 fun step1 h (a, f) b = fold (h (b, a), f)
 end
```

From the definition of `step1`, we have the following equivalence.

```
fold (a, f) (step1 h) b
= step1 h (a, f) b
= fold (h (b, a), f)
```

Using the above equivalence, we can compute the following equation for `step1`.

```
fold (a, f) (step1 h1) b1 (step1 h2) b2 ... (step1 hn) bn $
= fold (h1 (b1, a), f) (step1 h2) b2 ... (step1 hn) bn $
= fold (h2 (b2, h1 (b1, a)), f) ... (step1 hn) bn $
= fold (hn (bn, ... (h2 (b2, h1 (b1, a))))), f) $
= f (hn (bn, ... (h2 (b2, h1 (b1, a)))))
```

Here is an example using `step1` to define a variable-argument product function, `prod`, with a convenient syntax.

```
val prod = fn z => Fold.fold (1, fn p => p) z
val ` = fn z => Fold.step1 (fn (i, p) => i * p) z
```

The functions `prod` and ``` satisfy the following equation.

```
prod `i1 `i2 ... `im $ = i1 * i2 * ... * im
```

Note that in SML, ``i1` is two different tokens, ``` and `i1`. We often use ``` for an instance of a `step1` function because of its syntactic unobtrusiveness and because no space is required to separate it from an alphanumeric token.

Also note that there are no parenthesis around the steps. That is, the following expression is not the same as the above one (in fact, it is not type correct).

```
prod (`i1) (`i2) ... (`im) $
```

## 71.4 Example: list literals

SML already has a syntax for list literals, e.g. `[w, x, y, z]`. However, using `fold`, we can define our own syntax.

```
val list = fn z => Fold.fold ([], rev) z
val ` = fn z => Fold.step1 (op ::) z
```

The idea is that the folder starts out with the empty list, the steps accumulate the elements into a list, and then the finishing function reverses the list at the end.

With these definitions one can write a list like:

```
list `w `x `y `z $
```

While the example is not practically useful, it does demonstrate the need for the finishing function to be incorporated in `fold`. Without a finishing function, every use of `list` would need to be wrapped in `rev`, as follows.

```
rev (list `w `x `y `z $)
```

The finishing function allows us to incorporate the reversal into the definition of `list`, and to treat `list` as a truly variable argument function, performing an arbitrary computation after receiving all of its arguments.

See [ArrayLiteral](#) for a similar use of `fold` that provides a syntax for array and vector literals, which are not built in to SML.

## 71.5 Fold right

Just as `fold` is analogous to a fold left, in which the functions are applied to the accumulator left-to-right, we can define a variant of `fold` that is analogous to a fold right, in which the functions are applied to the accumulator right-to-left. That is, we can define functions `foldr` and `step0` such that the following equation holds.

```
foldr (a, f) (step0 h1) (step0 h2) ... (step0 hn) $
= f (h1 (h2 (... (hn a))))
```

The implementation of fold right is easy, using `fold`. The idea is for the fold to start with `f` and for each step to precompose the next `hi`. Then, the finisher applies the composed function to the base value, `a`. Here is the code.

```
structure Foldr =
 struct
 fun foldr (a, f) = Fold.fold (f, fn g => g a)
 fun step0 h = Fold.step0 (fn g => g o h)
 end
```

Verifying the fold-right equation is straightforward, using the fold-left equation.

```
foldr (a, f) (Foldr.step0 h1) (Foldr.step0 h2) ... (Foldr.step0 hn) $
= fold (f, fn g => g a)
 (Fold.step0 (fn g => g o h1))
 (Fold.step0 (fn g => g o h2))
 ...
 (Fold.step0 (fn g => g o hn)) $
= (fn g => g a)
 ((fn g => g o hn) (... ((fn g => g o h2) ((fn g => g o h1) f))))
= (fn g => g a)
 ((fn g => g o hn) (... ((fn g => g o h2) (f o h1))))
= (fn g => g a) ((fn g => g o hn) (... (f o h1 o h2)))
= (fn g => g a) (f o h1 o h2 o ... o hn)
= (f o h1 o h2 o ... o hn) a
= f (h1 (h2 (... (hn a))))
```

One can also define the fold-right analogue of `step1`.

```
structure Foldr =
 struct
 open Foldr
 fun step1 h = Fold.step1 (fn (b, g) => g o (fn a => h (b, a)))
 end
```

## 71.6 Example: list literals via fold right

Revisiting the list literal example from earlier, we can use fold right to define a syntax for list literals that doesn't do a reversal.

```
val list = fn z => Foldr.foldr ([], fn l => l) z
val ` = fn z => Foldr.step1 (op ::) z
```

As before, with these definitions, one can write a list like:

```
list `w `x `y `z $
```

The difference between the fold-left and fold-right approaches is that the fold-right approach does not have to reverse the list at the end, since it accumulates the elements in the correct order. In practice, MLton will simplify away all of the intermediate function composition, so the the fold-right approach will be more efficient.

## 71.7 Mixing steppers

All of the examples so far have used the same step function throughout a fold. This need not be the case. For example, consider the following.

```
val n = fn z => Fold.fold (0, fn i => i) z
val I = fn z => Fold.step0 (fn i => i * 2) z
val O = fn z => Fold.step0 (fn i => i * 2 + 1) z
```

Here we have one folder, `n`, that can be used with two different steppers, `I` and `O`. By using the fold equation, one can verify the following equations.

```
n 0 $ = 0
n I $ = 1
n I O $ = 2
n I O I $ = 5
n I I I O $ = 14
```

That is, we've defined a syntax for writing binary integer constants.

Not only can one use different instances of `step0` in the same fold, one can also intermix uses of `step0` and `step1`. For example, consider the following.

```
val n = fn z => Fold.fold (0, fn i => i) z
val 0 = fn z => Fold.step0 (fn i => n * 8) z
val ` = fn z => Fold.step1 (fn (i, n) => n * 8 + i) z
```

Using the straightforward generalization of the fold equation to mixed steppers, one can verify the following equations.

```
n 0 $ = 0
n `3 0 $ = 24
n `1 0 `7 $ = 71
```

That is, we've defined a syntax for writing octal integer constants, with a special syntax, `O`, for the zero digit (admittedly contrived, since one could just write ``0` instead of `O`).

See [NumericLiteral](#) for a practical extension of this approach that supports numeric constants in any base and of any type.

## 71.8 (Seemingly) dependent types

A normal list fold always returns the same type no matter what elements are in the list or how long the list is. Variable-argument fold is more powerful, because the result type can vary based both on the arguments that are passed and on their number. This can provide the illusion of dependent types.

For example, consider the following.

```
val f = fn z => Fold.fold ((), id) z
val a = fn z => Fold.step0 (fn () => "hello") z
val b = fn z => Fold.step0 (fn () => 13) z
val c = fn z => Fold.step0 (fn () => (1, 2)) z
```

Using the fold equation, one can verify the following equations.

```
f a $ = "hello": string
f b $ = 13: int
f c $ = (1, 2): int * int
```

That is, `f` returns a value of a different type depending on whether it is applied to argument `a`, argument `b`, or argument `c`.

The following example shows how the type of a fold can depend on the number of arguments.

```
val grow = fn z => Fold.fold ([], fn l => l) z
val a = fn z => Fold.step0 (fn x => [x]) z
```

Using the fold equation, one can verify the following equations.

```
grow $ = []: 'a list
grow a $ = [[]]: 'a list list
grow a a $ = [[[]]]: 'a list list list
```

Clearly, the result type of a call to the variable argument `grow` function depends on the number of arguments that are passed.

As a reminder, this is well-typed SML. You can check it out in any implementation.

## 71.9 (Seemingly) dependently-typed functional results

Fold is especially useful when it returns a curried function whose arity depends on the number of arguments. For example, consider the following.

```
val makeSum = fn z => Fold.fold (id, fn f => f 0) z
val I = fn z => Fold.step0 (fn f => fn i => fn x => f (x + i)) z
```

The `makeSum` folder constructs a function whose arity depends on the number of `I` arguments and that adds together all of its arguments. For example, `makeSum I $` is of type `int -> int` and `makeSum I I $` is of type `int -> int -> int`.

One can use the fold equation to verify that the `makeSum` works correctly. For example, one can easily check by hand the following equations.

```
makeSum I $ 1 = 1
makeSum I I $ 1 2 = 3
makeSum I I I $ 1 2 3 = 6
```

Returning a function becomes especially interesting when there are steppers of different types. For example, the following `makeSum` folder constructs functions that sum integers and reals.

```
val makeSum = fn z => Foldr.foldr (id, fn f => f 0.0) z
val I = fn z => Foldr.step0 (fn f => fn x => fn i => f (x + real i)) z
val R = fn z => Foldr.step0 (fn f => fn x: real => fn r => f (x + r)) z
```

With these definitions, `makeSum I R $` is of type `int -> real -> real` and `makeSum R I I $` is of type `real -> int -> int -> real`. One can use the `foldr` equation to check the following equations.

```
makeSum I $ 1 = 1.0
makeSum I R $ 1 2.5 = 3.5
makeSum R I I $ 1.5 2 3 = 6.5
```

We used `foldr` instead of `fold` for this so that the order in which the specifiers `I` and `R` appear is the same as the order in which the arguments appear. Had we used `fold`, things would have been reversed.

An extension of this idea is sufficient to define `Printf`-like functions in SML.

## 71.10 An idiom for combining steps

It is sometimes useful to combine a number of steps together and name them as a single step. As a simple example, suppose that one often sees an integer follower by a real in the `makeSum` example above. One can define a new *compound step* `IR` as follows.

```
val IR = fn u => Fold.fold u I R
```

With this definition in place, one can verify the following.

```
makeSum IR IR $ 1 2.2 3 4.4 = 10.6
```

In general, one can combine steps `s1, s2, ... sn` as

```
fn u => Fold.fold u s1 s2 ... sn
```

The following calculation shows why a compound step behaves as the composition of its constituent steps.

```
fold u (fn u => fold u s1 s2 ... sn)
= (fn u => fold u s1 s2 ... sn) u
= fold u s1 s2 ... sn
```

## 71.11 Post composition

Suppose we already have a function defined via fold,  $w = \text{fold } (a, f)$ , and we would like to construct a new fold function that is like  $w$ , but applies  $g$  to the result produced by  $w$ . This is similar to function composition, but we can't just do  $g \circ w$ , because we don't want to use  $g$  until  $w$  has been applied to all of its arguments and received the end-of-arguments terminator  $\$$ .

More precisely, we want to define a post-composition function `post` that satisfies the following equation.

```
post (w, g) s1 ... sn $ = g (w s1 ... sn $)
```

Here is the definition of `post`.

```
structure Fold =
 struct
 open Fold
 fun post (w, g) s = w (fn (a, h) => s (a, g o h))
 end
```

The following calculations show that `post` satisfies the desired equation, where  $w = \text{fold } (a, f)$ .

```
post (w, g) s
= w (fn (a, h) => s (a, g o h))
= fold (a, f) (fn (a, h) => s (a, g o h))
= (fn (a, h) => s (a, g o h)) (a, f)
= s (a, g o f)
= fold (a, g o f) s
```

Now, suppose  $s_i = \text{step0 } h_i$  for  $i$  from 1 to  $n$ .

```
post (w, g) s1 s2 ... sn $
= fold (a, g o f) s1 s2 ... sn $
= (g o f) (hn (... (h1 a)))
= g (f (hn (... (h1 a))))
= g (fold (a, f) s1 ... sn $)
= g (w s1 ... sn $)
```

For a practical example of post composition, see [ArrayLiteral](#).

## 71.12 Lift

We now define a peculiar-looking function, `lift0`, that is, equationally speaking, equivalent to the identity function on a step function.

```
fun lift0 s (a, f) = fold (fold (a, id) s $, f)
```

Using the definitions, we can prove the following equation.

```
fold (a, f) (lift0 (step0 h)) = fold (a, f) (step0 h)
```

Here is the proof.

```
fold (a, f) (lift0 (step0 h))
= lift0 (step0 h) (a, f)
= fold (fold (a, id) (step0 h) $, f)
= fold (step0 h (a, id) $, f)
= fold (fold (h a, id) $, f)
= fold ($ (h a, id), f)
= fold (id (h a), f)
= fold (h a, f)
= step0 h (a, f)
= fold (a, f) (step0 h)
```

If `lift0` is the identity, then why even define it? The answer lies in the typing of fold expressions, which we have, until now, left unexplained.

## 71.13 Typing

Perhaps the most surprising aspect of fold is that it can be checked by the SML type system. The types involved in fold expressions are complex; fortunately type inference is able to deduce them. Nevertheless, it is instructive to study the types of fold functions and steppers. More importantly, it is essential to understand the typing aspects of fold in order to write down signatures of functions defined using fold and step.

Here is the FOLD signature, and a recapitulation of the entire Fold structure, with additional type annotations.

```
signature FOLD =
 sig
 type ('a, 'b, 'c, 'd) step = 'a * ('b -> 'c) -> 'd
 type ('a, 'b, 'c, 'd) t = ('a, 'b, 'c, 'd) step -> 'd
 type ('a1, 'a2, 'b, 'c, 'd) step0 =
 ('a1, 'b, 'c, ('a2, 'b, 'c, 'd) t) step
 type ('a11, 'a12, 'a2, 'b, 'c, 'd) step1 =
 ('a12, 'b, 'c, 'a11 -> ('a2, 'b, 'c, 'd) t) step

 val fold: 'a * ('b -> 'c) -> ('a, 'b, 'c, 'd) t
 val lift0: ('a1, 'a2, 'a2, 'a2, 'a2) step0
 -> ('a1, 'a2, 'b, 'c, 'd) step0
 val post: ('a, 'b, 'c1, 'd) t * ('c1 -> 'c2)
 -> ('a, 'b, 'c2, 'd) t
 val step0: ('a1 -> 'a2) -> ('a1, 'a2, 'b, 'c, 'd) step0
 val step1: ('a11 * 'a12 -> 'a2)
 -> ('a11, 'a12, 'a2, 'b, 'c, 'd) step1
 end

structure Fold:> FOLD =
 struct
 type ('a, 'b, 'c, 'd) step = 'a * ('b -> 'c) -> 'd

 type ('a, 'b, 'c, 'd) t = ('a, 'b, 'c, 'd) step -> 'd

 type ('a1, 'a2, 'b, 'c, 'd) step0 =
 ('a1, 'b, 'c, ('a2, 'b, 'c, 'd) t) step

 type ('a11, 'a12, 'a2, 'b, 'c, 'd) step1 =
 ('a12, 'b, 'c, 'a11 -> ('a2, 'b, 'c, 'd) t) step

 fun fold (a: 'a, f: 'b -> 'c)
 (g: ('a, 'b, 'c, 'd) step): 'd =
 g (a, f)

 fun step0 (h: 'a1 -> 'a2)
 (a1: 'a1, f: 'b -> 'c): ('a2, 'b, 'c, 'd) t =
 fold (h a1, f)

 fun step1 (h: 'a11 * 'a12 -> 'a2)
 (a12: 'a12, f: 'b -> 'c)
 (a11: 'a11): ('a2, 'b, 'c, 'd) t =
 fold (h (a11, a12), f)

 fun lift0 (s: ('a1, 'a2, 'a2, 'a2, 'a2) step0)
 (a: 'a1, f: 'b -> 'c): ('a2, 'b, 'c, 'd) t =
 fold (fold (a, id) s $, f)

 fun post (w: ('a, 'b, 'c1, 'd) t,
 g: 'c1 -> 'c2)
 (s: ('a, 'b, 'c2, 'd) step): 'd =
 w (fn (a, h) => s (a, g o h))
 end
```

That's a lot to swallow, so let's walk through it one step at a time. First, we have the definition of type `Fold.step`.

```
type ('a, 'b, 'c, 'd) step = 'a * ('b -> 'c) -> 'd
```

As a fold proceeds over its arguments, it maintains two things: the accumulator, of type `'a`, and the finishing function, of type `'b -> 'c`. Each step in the fold is a function that takes those two pieces (i.e. `'a * ('b -> 'c)`) and does something to them (i.e. produces `'d`). The result type of the step is completely left open to be filled in by type inference, as it is an arrow type that is capable of consuming the rest of the arguments to the fold.

A folder, of type `Fold.t`, is a function that consumes a single step.

```
type ('a, 'b, 'c, 'd) t = ('a, 'b, 'c, 'd) step -> 'd
```

Expanding out the type, we have:

```
type ('a, 'b, 'c, 'd) t = ('a * ('b -> 'c) -> 'd) -> 'd
```

This shows that the only thing a folder does is to hand its accumulator (`'a`) and finisher (`'b -> 'c`) to the next step (`'a * ('b -> 'c) -> 'd`). If SML had [first-class polymorphism](#), we would write the fold type as follows.

```
type ('a, 'b, 'c) t = Forall 'd . ('a, 'b, 'c, 'd) step -> 'd
```

This type definition shows that a folder had nothing to do with the rest of the fold, it only deals with the next step.

We now can understand the type of `fold`, which takes the initial value of the accumulator and the finishing function, and constructs a folder, i.e. a function awaiting the next step.

```
val fold: 'a * ('b -> 'c) -> ('a, 'b, 'c, 'd) t
fun fold (a: 'a, f: 'b -> 'c)
 (g: ('a, 'b, 'c, 'd) step): 'd =
 g (a, f)
```

Continuing on, we have the type of step functions.

```
type ('a1, 'a2, 'b, 'c, 'd) step0 =
 ('a1, 'b, 'c, ('a2, 'b, 'c, 'd) t) step
```

Expanding out the type a bit gives:

```
type ('a1, 'a2, 'b, 'c, 'd) step0 =
 'a1 * ('b -> 'c) -> ('a2, 'b, 'c, 'd) t
```

So, a step function takes the accumulator (`'a1`) and finishing function (`'b -> 'c`), which will be passed to it by the previous folder, and transforms them to a new folder. This new folder has a new accumulator (`'a2`) and the same finishing function.

Again, imagining that SML had [first-class polymorphism](#) makes the type clearer.

```
type ('a1, 'a2) step0 =
 Forall ('b, 'c) . ('a1, 'b, 'c, ('a2, 'b, 'c) t) step
```

Thus, in essence, a `step0` function is a wrapper around a function of type `'a1 -> 'a2`, which is exactly what the definition of `step0` does.

```
val step0: ('a1 -> 'a2) -> ('a1, 'a2, 'b, 'c, 'd) step0
fun step0 (h: 'a1 -> 'a2)
 (a1: 'a1, f: 'b -> 'c): ('a2, 'b, 'c, 'd) t =
 fold (h a1, f)
```

It is not much beyond `step0` to understand `step1`.

```
type ('a11, 'a12, 'a2, 'b, 'c, 'd) step1 =
 ('a12, 'b, 'c, 'a11 -> ('a2, 'b, 'c, 'd) t) step
```

A `step1` function takes the accumulator (`'a12`) and finisher (`'b -> 'c`) passed to it by the previous folder and transforms them into a function that consumes the next argument (`'a11`) and produces a folder that will continue the fold with a new accumulator (`'a2`) and the same finisher.

```
fun step1 (h: 'a11 * 'a12 -> 'a2)
 (a12: 'a12, f: 'b -> 'c)
 (a11: 'a11): ('a2, 'b, 'c, 'd) t =
 fold (h (a11, a12), f)
```

With [first-class polymorphism](#), a `step1` function is more clearly seen as a wrapper around a binary function of type `'a11 * 'a12 -> 'a2`.

```
type ('a11, 'a12, 'a2) step1 =
 Forall ('b, 'c) . ('a12, 'b, 'c, 'a11 -> ('a2, 'b, 'c) t) step
```

The type of `post` is clear: it takes a folder with a finishing function that produces type `'c1`, and a function of type `'c1 -> 'c2` to postcompose onto the folder. It returns a new folder with a finishing function that produces type `'c2`.

```
val post: ('a, 'b, 'c1, 'd) t * ('c1 -> 'c2)
 -> ('a, 'b, 'c2, 'd) t
fun post (w: ('a, 'b, 'c1, 'd) t,
 g: 'c1 -> 'c2)
 (s: ('a, 'b, 'c2, 'd) step): 'd =
 w (fn (a, h) => s (a, g o h))
```

We will return to `lift0` after an example.

## 71.14 An example typing

Let's type check our simplest example, a variable-argument fold. Recall that we have a folder `f` and a stepper `a` defined as follows.

```
val f = fn z => Fold.fold ((), fn () => ()) z
val a = fn z => Fold.step0 (fn () => ()) z
```

Since the accumulator and finisher are uninteresting, we'll use some abbreviations to simplify things.

```
type 'd step = (unit, unit, unit, 'd) Fold.step
type 'd fold = 'd step -> 'd
```

With these abbreviations, `f` and `a` have the following polymorphic types.

```
f: 'd fold
a: 'd step
```

Suppose we want to type check

```
f a a a $: unit
```

As a reminder, the fully parenthesized expression is

```
((((f a) a) a) a) $
```

The observation that we will use repeatedly is that for any type `z`, if `f: z fold` and `s: z step`, then `f s: z`. So, if we want

```
(f a a a) $: unit
```

then we must have

```
f a a a: unit fold
$: unit step
```

Applying the observation again, we must have

```
f a a: unit fold fold
a: unit fold step
```

Applying the observation two more times leads to the following type derivation.

```
f: unit fold fold fold fold a: unit fold fold fold step
f a: unit fold fold fold a: unit fold fold step
f a a: unit fold fold a: unit fold step
f a a a: unit fold $: unit step
f a a a $: unit
```

So, each application is a fold that consumes the next step, producing a fold of one smaller type.

One can expand some of the type definitions in `f` to see that it is indeed a function that takes four curried arguments, each one a step function.

```
f: unit fold fold fold step
 -> unit fold fold step
 -> unit fold step
 -> unit step
 -> unit
```

This example shows why we must eta expand uses of `fold` and `step0` to work around the value restriction and make folders and steppers polymorphic. The type of a fold function like `f` depends on the number of arguments, and so will vary from use to use. Similarly, each occurrence of an argument like `a` has a different type, depending on the number of remaining arguments.

This example also shows that the type of a folder, when fully expanded, is exponential in the number of arguments: there are as many nested occurrences of the `fold` type constructor as there are arguments, and each occurrence duplicates its type argument. One can observe this exponential behavior in a type checker that doesn't share enough of the representation of types (e.g. one that represents types as trees rather than directed acyclic graphs).

Generalizing this type derivation to uses of `fold` where the accumulator and finisher are more interesting is straightforward. One simply includes the type of the accumulator, which may change, for each step, and the type of the finisher, which doesn't change from step to step.

## 71.15 Typing lift

The lack of [first-class polymorphism](#) in SML causes problems if one wants to use a step in a first-class way. Consider the following `double` function, which takes a step, `s`, and produces a composite step that does `s` twice.

```
fun double s = fn u => Fold.fold u s s
```

The definition of `double` is not type correct. The problem is that the type of a step depends on the number of remaining arguments but that the parameter `s` is not polymorphic, and so can not be used in two different positions.

Fortunately, we can define a function, `lift0`, that takes a monotyped step function and *lifts* it into a polymorphic step function. This is apparent in the type of `lift0`.

```
val lift0: ('a1, 'a2, 'a2, 'a2, 'a2) step0
 -> ('a1, 'a2, 'b, 'c, 'd) step0
fun lift0 (s: ('a1, 'a2, 'a2, 'a2, 'a2) step0)
 (a: 'a1, f: 'b -> 'c): ('a2, 'b, 'c, 'd) t =
 fold (fold (a, id) s $, f)
```

The following definition of `double` uses `lift0`, appropriately eta wrapped, to fix the problem.

```

fun double s =
 let
 val s = fn z => Fold.lift0 s z
 in
 fn u => Fold.fold u s s
 end

```

With that definition of `double` in place, we can use it as in the following example.

```

val f = fn z => Fold.fold ((), fn () => ()) z
val a = fn z => Fold.step0 (fn () => ()) z
val a2 = fn z => double a z
val () = f a a2 a a2 $

```

Of course, we must eta wrap the call `double` in order to use its result, which is a step function, polymorphically.

## 71.16 Hiding the type of the accumulator

For clarity and to avoid mistakes, it can be useful to hide the type of the accumulator in a fold. Reworking the simple variable-argument example to do this leads to the following.

```

structure S:>
 sig
 type ac
 val f: (ac, ac, unit, 'd) Fold.t
 val s: (ac, ac, 'b, 'c, 'd) Fold.step0
 end =
 struct
 type ac = unit
 val f = fn z => Fold.fold ((), fn () => ()) z
 val s = fn z => Fold.step0 (fn () => ()) z
 end

```

The idea is to name the accumulator type and use opaque signature matching to make it abstract. This can prevent improper manipulation of the accumulator by client code and ensure invariants that the folder and stepper would like to maintain.

For a practical example of this technique, see [ArrayLiteral](#).

## 71.17 Also see

Fold has a number of practical applications. Here are some of them.

- [ArrayLiteral](#)
- [Fold01N](#)
- [FunctionalRecordUpdate](#)
- [NumericLiteral](#)
- [OptionalArguments](#)
- [Printf](#)
- [VariableArityPolymorphism](#)

There are a number of related techniques. Here are some of them.

- [StaticSum](#)
- [TypeIndexedValues](#)

## 72 Fold01N

A common use pattern of `Fold` is to define a variable-arity function that combines multiple arguments together using a binary function. It is slightly tricky to do this directly using `fold`, because of the special treatment required for the case of zero or one argument. Here is a structure, `Fold01N`, that solves the problem once and for all, and eases the definition of such functions.

```
structure Fold01N =
 struct
 fun fold {finish, start, zero} =
 Fold.fold ((id, finish, fn () => zero, start),
 fn (finish, _, p, _) => finish (p ()))

 fun step0 {combine, input} =
 Fold.step0 (fn (_, finish, _, f) =>
 (finish,
 finish,
 fn () => f input,
 fn x' => combine (f input, x')))

 fun step1 {combine} z input =
 step0 {combine = combine, input = input} z
 end
```

If one has a value `zero`, and functions `start`, `c`, and `finish`, then one can define a variable-arity function `f` and stepper ``` as follows.

```
val f = fn z => Fold01N.fold {finish = finish, start = start, zero = zero} z
val ` = fn z => Fold01N.step1 {combine = c} z
```

One can then use the fold equation to prove the following equations.

```
f $ = zero
f `a1 $ = finish (start a1)
f `a1 `a2 $ = finish (c (start a1, a2))
f `a1 `a2 `a3 $ = finish (c (c (start a1, a2), a3))
...
```

For an example of `Fold01N`, see [VariableArityPolymorphism](#).

### 72.1 Typing Fold01N

Here is the signature for `Fold01N`. We use a trick to avoid having to duplicate the definition of some rather complex types in both the signature and the structure. We first define the types in a structure. Then, we define them via type re-definitions in the signature, and via `open` in the full structure.

```
structure Fold01N =
 struct
 type ('input, 'accum1, 'accum2, 'answer, 'zero,
 'a, 'b, 'c, 'd, 'e) t =
 (('zero -> 'zero)
 * ('accum2 -> 'answer)
 * (unit -> 'zero)
 * ('input -> 'accum1),
 ('a -> 'b) * 'c * (unit -> 'a) * 'd,
 'b,
 'e) Fold.t

 type ('input1, 'accum1, 'input2, 'accum2,
 'a, 'b, 'c, 'd, 'e, 'f) step0 =
 ('a * 'b * 'c * ('input1 -> 'accum1),
```

```

 'b * 'b * (unit -> 'accum1) * ('input2 -> 'accum2),
 'd, 'e, 'f) Fold.step0

type ('accum1, 'input, 'accum2,
 'a, 'b, 'c, 'd, 'e, 'f, 'g) step1 =
('a,
 'b * 'c * 'd * ('a -> 'accum1),
 'c * 'c * (unit -> 'accum1) * ('input -> 'accum2),
 'e, 'f, 'g) Fold.step1
end

signature FOLD_01N =
sig
type ('a, 'b, 'c, 'd, 'e, 'f, 'g, 'h, 'i, 'j) t =
('a, 'b, 'c, 'd, 'e, 'f, 'g, 'h, 'i, 'j) Fold01N.t
type ('a, 'b, 'c, 'd, 'e, 'f, 'g, 'h, 'i, 'j) step0 =
('a, 'b, 'c, 'd, 'e, 'f, 'g, 'h, 'i, 'j) Fold01N.step0
type ('a, 'b, 'c, 'd, 'e, 'f, 'g, 'h, 'i, 'j) step1 =
('a, 'b, 'c, 'd, 'e, 'f, 'g, 'h, 'i, 'j) Fold01N.step1

val fold:
{finish: 'accum2 -> 'answer,
 start: 'input -> 'accum1,
 zero: 'zero}
-> ('input, 'accum1, 'accum2, 'answer, 'zero,
 'a, 'b, 'c, 'd, 'e) t

val step0:
{combine: 'accum1 * 'input2 -> 'accum2,
 input: 'input1}
-> ('input1, 'accum1, 'input2, 'accum2,
 'a, 'b, 'c, 'd, 'e, 'f) step0

val step1:
{combine: 'accum1 * 'input -> 'accum2}
-> ('accum1, 'input, 'accum2,
 'a, 'b, 'c, 'd, 'e, 'f, 'g) step1
end

structure Fold01N: FOLD_01N =
struct
open Fold01N

fun fold {finish, start, zero} =
Fold.fold ((id, finish, fn () => zero, start),
 fn (finish, _, p, _) => finish (p ()))

fun step0 {combine, input} =
Fold.step0 (fn (_, finish, _, f) =>
 (finish,
 finish,
 fn () => f input,
 fn x' => combine (f input, x')))

fun step1 {combine} z input =
step0 {combine = combine, input = input} z
end

```

## 73 ForeignFunctionInterface

MLton's foreign function interface (FFI) extends Standard ML and makes it easy to take the address of C global objects, access C global variables, call from SML to C, and call from C to SML. MLton also provides [ML-NLFFI](#), which is a higher-level FFI for calling C functions and manipulating C data from SML.

### 73.1 Overview

- [Foreign Function Interface Types](#)
- [Foreign Function Interface Syntax](#)

### 73.2 Importing Code into SML

- [Calling From SML To C](#)
- [Calling From SML To C Function Pointer](#)

### 73.3 Exporting Code from SML

- [Calling From C To SML](#)

### 73.4 Building System Libraries

- [Library Support](#)
-

## 74 ForeignFunctionInterfaceSyntax

MLton extends the syntax of SML with expressions that enable a [ForeignFunctionInterface](#) to C. The following description of the syntax uses some abbreviations.

| C base type     | <i>cBaseTy</i> | Foreign Function Interface types        |
|-----------------|----------------|-----------------------------------------|
| C argument type | <i>cArgTy</i>  | $cBaseTy_1 * \dots * cBaseTy_n$ or unit |
| C return type   | <i>cRetTy</i>  | <i>cBaseTy</i> or unit                  |
| C function type | <i>cFuncTy</i> | $cArgTy \rightarrow cRetTy$             |
| C pointer type  | <i>cPtrTy</i>  | <code>MLton.Pointer.t</code>            |

The type annotation and the semicolon are not optional in the syntax of [ForeignFunctionInterface](#) expressions. However, the type is lexed, parsed, and elaborated as an SML type, so any type (including type abbreviations) may be used, so long as it elaborates to a type of the correct form.

### 74.1 Address

```
_address "CFunctionOrVariableName" attr... : cPtrTy;
```

Denotes the address of the C function or variable.

`attr...` denotes a (possibly empty) sequence of attributes. The following attributes are recognized:

- `external` : import with external symbol scope (see [LibrarySupport](#)) (default).
- `private` : import with private symbol scope (see [LibrarySupport](#)).
- `public` : import with public symbol scope (see [LibrarySupport](#)).

See [MLtonPointer](#) for functions that manipulate C pointers.

### 74.2 Symbol

```
_symbol "CVariableName" attr... : (unit -> cBaseTy) * (cBaseTy -> unit);
```

Denotes the *getter* and *setter* for a C variable. The *cBaseTys* must be identical.

`attr...` denotes a (possibly empty) sequence of attributes. The following attributes are recognized:

- `alloc` : allocate storage (and export a symbol) for the C variable.
- `external` : import or export with external symbol scope (see [LibrarySupport](#)) (default if not `alloc`).
- `private` : import or export with private symbol scope (see [LibrarySupport](#)).
- `public` : import or export with public symbol scope (see [LibrarySupport](#)) (default if `alloc`).

```
_symbol * : cPtrTy -> (unit -> cBaseTy) * (cBaseTy -> unit);
```

Denotes the *getter* and *setter* for a C pointer to a variable. The *cBaseTys* must be identical.

### 74.3 Import

```
_import "CFunctionName" attr... : cFuncTy;
```

Denotes an SML function whose behavior is implemented by calling the C function. See [Calling from SML to C](#) for more details.

`attr...` denotes a (possibly empty) sequence of attributes. The following attributes are recognized:

- `cdecl` : call with the `cdecl` calling convention (default).
- `external` : import with external symbol scope (see [LibrarySupport](#)) (default).
- `private` : import with private symbol scope (see [LibrarySupport](#)).
- `public` : import with public symbol scope (see [LibrarySupport](#)).
- `stdcall` : call with the `stdcall` calling convention (ignored except on Cygwin and MinGW).

```
_import * attr... : cPtrTy -> cFuncTy;
```

Denotes an SML function whose behavior is implemented by calling a C function through a C function pointer.

`attr...` denotes a (possibly empty) sequence of attributes. The following attributes are recognized:

- `cdecl` : call with the `cdecl` calling convention (default).
- `stdcall` : call with the `stdcall` calling convention (ignored except on Cygwin and MinGW).

See [Calling from SML to C function pointer](#) for more details.

### 74.4 Export

```
_export "CFunctionName" attr... : cFuncTy -> unit;
```

Exports a C function with the name `CFunctionName` that can be used to call an SML function of the type `cFuncTy`. When the function denoted by the export expression is applied to an SML function `f`, subsequent C calls to `CFunctionName` will call `f`. It is an error to call `CFunctionName` before the export has been applied. The export may be applied more than once, with each application replacing any previous definition of `CFunctionName`.

`attr...` denotes a (possibly empty) sequence of attributes. The following attributes are recognized:

- `cdecl` : call with the `cdecl` calling convention (default).
- `private` : export with private symbol scope (see [LibrarySupport](#)).
- `public` : export with public symbol scope (see [LibrarySupport](#)) (default).
- `stdcall` : call with the `stdcall` calling convention (ignored except on Cygwin and MinGW).

See [Calling from C to SML](#) for more details.

---

## 75 ForeignFunctionInterfaceTypes

MLton's [ForeignFunctionInterface](#) only allows values of certain SML types to be passed between SML and C. The following types are allowed: `bool`, `char`, `int`, `real`, `word`. All of the different sizes of (fixed-sized) integers, reals, and words are supported as well: `Int8.int`, `Int16.int`, `Int32.int`, `Int64.int`, `Real32.real`, `Real64.real`, `Word8.word`, `Word16.word`, `Word32.word`, `Word64.word`. There is a special type, `MLton.Pointer.t`, for passing C pointers—see [MLtonPointer](#) for details.

Arrays, refs, and vectors of the above types are also allowed. Because in MLton monomorphic arrays and vectors are exactly the same as their polymorphic counterpart, these are also allowed. Hence, `string`, `char vector`, and `CharVector.vector` are also allowed. Strings are not null terminated, unless you manually do so from the SML side.

Unfortunately, passing tuples or datatypes is not allowed because that would interfere with representation optimizations.

The C header file that `-export-header` generates includes `typedefs` for the C types corresponding to the SML types. Here is the mapping between SML types and C types.

| SML type                     | C typedef            | C type                       | Note        |
|------------------------------|----------------------|------------------------------|-------------|
| <code>array</code>           | <code>Pointer</code> | <code>unsigned char *</code> |             |
| <code>bool</code>            | <code>Bool</code>    | <code>int32_t</code>         |             |
| <code>char</code>            | <code>Char8</code>   | <code>uint8_t</code>         |             |
| <code>Int8.int</code>        | <code>Int8</code>    | <code>int8_t</code>          |             |
| <code>Int16.int</code>       | <code>Int16</code>   | <code>int16_t</code>         |             |
| <code>Int32.int</code>       | <code>Int32</code>   | <code>int32_t</code>         |             |
| <code>Int64.int</code>       | <code>Int64</code>   | <code>int64_t</code>         |             |
| <code>int</code>             | <code>Int32</code>   | <code>int32_t</code>         | (default)   |
| <code>MLton.Pointer.t</code> | <code>Pointer</code> | <code>unsigned char *</code> |             |
| <code>Real32.real</code>     | <code>Real32</code>  | <code>float</code>           |             |
| <code>Real64.real</code>     | <code>Real64</code>  | <code>double</code>          |             |
| <code>real</code>            | <code>Real64</code>  | <code>double</code>          | (default)   |
| <code>ref</code>             | <code>Pointer</code> | <code>unsigned char *</code> |             |
| <code>string</code>          | <code>Pointer</code> | <code>unsigned char *</code> | (read only) |
| <code>vector</code>          | <code>Pointer</code> | <code>unsigned char *</code> | (read only) |
| <code>Word8.word</code>      | <code>Word8</code>   | <code>uint8_t</code>         |             |
| <code>Word16.word</code>     | <code>Word16</code>  | <code>uint16_t</code>        |             |
| <code>Word32.word</code>     | <code>Word32</code>  | <code>uint32_t</code>        |             |
| <code>Word64.word</code>     | <code>Word64</code>  | <code>uint64_t</code>        |             |
| <code>word</code>            | <code>Word32</code>  | <code>uint32_t</code>        | (default)   |

Note (default): The default `int`, `real`, and `word` types may be set by the `-default-type type` compiler option. The given C typedef and C types correspond to the default behavior.

Note (read only): Because MLton assumes that vectors and strings are read-only (and will perform optimizations that, for instance, cause them to share space), you must not modify the data pointed to by the `unsigned char *` in C code.

Although the C type of an array, ref, or vector is always `Pointer`, in reality, the object has the natural C representation. Your C code should cast to the appropriate C type if you want to keep the C compiler from complaining.

When calling an [imported C function from SML](#) that returns an array, ref, or vector result or when calling an [exported SML function from C](#) that takes an array, ref, or string argument, then the object must be an ML object allocated on the ML heap. (Although an array, ref, or vector object has the natural C representation, the object also has an additional header used by the SML runtime system.)

In addition, there is an [MLBasis](#) file, `$(SML_LIB)/basis/c-types.mlb`, which provides structure aliases for various C types:

| C type                   | Structure            | Signature            |
|--------------------------|----------------------|----------------------|
| <code>char</code>        | <code>C_Char</code>  | <code>INTEGER</code> |
| <code>signed char</code> | <code>C_SChar</code> | <code>INTEGER</code> |

---

|                    |             |         |
|--------------------|-------------|---------|
| unsigned char      | C_UChar     | WORD    |
| short              | C_Short     | INTEGER |
| signed short       | C_SShort    | INTEGER |
| unsigned short     | C_UShort    | WORD    |
| int                | C_Int       | INTEGER |
| signed int         | C_SInt      | INTEGER |
| unsigned int       | C_UInt      | WORD    |
| long               | C_Long      | INTEGER |
| signed long        | C_SLong     | INTEGER |
| unsigned long      | C_ULong     | WORD    |
| long long          | C_LongLong  | INTEGER |
| signed long long   | C_SLongLong | INTEGER |
| unsigned long long | C_ULongLong | WORD    |
| float              | C_Float     | REAL    |
| double             | C_Double    | REAL    |
| size_t             | C_Size      | WORD    |
| ptrdiff_t          | C_Ptrdiff   | INTEGER |
| intmax_t           | C_Intmax    | INTEGER |
| uintmax_t          | C_UIntmax   | WORD    |
| intptr_t           | C_Intptr    | INTEGER |
| uintptr_t          | C_Uintptr   | WORD    |
| void *             | C_Pointer   | WORD    |

These aliases depend on the configuration of the C compiler for the target architecture, and are independent of the configuration of MLton (including the `-default-type type` [compiler option](#)).

---

## 76 ForLoops

A `for`-loop is typically used to iterate over a range of consecutive integers that denote indices of some sort. For example, in [OCaml](#) a `for`-loop takes either the form

```
for <name> = <lower> to <upper> do <body> done
```

or the form

```
for <name> = <upper> downto <lower> do <body> done
```

Some languages provide considerably more flexible `for`-loop or `foreach`-constructs.

A bit surprisingly, [Standard ML](#) provides special syntax for `while`-loops, but not for `for`-loops. Indeed, in SML, many uses of `for`-loops are better expressed using `app`, `foldl/foldr`, `map` and many other higher-order functions provided by the [Basis Library](#) for manipulating lists, vectors and arrays. However, the Basis Library does not provide a function for iterating over a range of integer values. Fortunately, it is very easy to write one.

### 76.1 A fairly simple design

The following implementation imitates both the syntax and semantics of the OCaml `for`-loop.

```
datatype for = to of int * int
 | downto of int * int

infix to downto

val for =
 fn lo to up =>
 (fn f => let fun loop lo = if lo > up then ()
 else (f lo; loop (lo+1))
 in loop lo end)
 | up downto lo =>
 (fn f => let fun loop up = if up < lo then ()
 else (f up; loop (up-1))
 in loop up end)
```

For example,

```
for (1 to 9)
 (fn i => print (Int.toString i))
```

would print 123456789 and

```
for (9 downto 1)
 (fn i => print (Int.toString i))
```

would print 987654321.

Straightforward formatting of nested loops

```
for (a to b)
 (fn i =>
 for (c to d)
 (fn j =>
 ...))
```

is fairly readable, but tends to cause the body of the loop to be indented quite deeply.

## 76.2 Off-by-one

The above design has an annoying feature. In practice, the upper bound of the iterated range is almost always excluded and most loops would subtract one from the upper bound:

```
for (0 to n-1) ...
for (n-1 downto 0) ...
```

It is probably better to break convention and exclude the upper bound by default, because it leads to more concise code and becomes idiomatic with very little practice. The iterator combinators described below exclude the upper bound by default.

## 76.3 Iterator combinators

While the simple `for`-function described in the previous section is probably good enough for many uses, it is a bit cumbersome when one needs to iterate over a Cartesian product. One might also want to iterate over more than just consecutive integers. It turns out that one can provide a library of iterator combinators that allow one to implement iterators more flexibly.

Since the types of the combinators may be a bit difficult to infer from their implementations, let's first take a look at a signature of the iterator combinator library:

```
signature ITER =
sig
 type 'a t = ('a -> unit) -> unit

 val return : 'a -> 'a t
 val >>= : 'a t * ('a -> 'b t) -> 'b t

 val none : 'a t

 val to : int * int -> int t
 val downto : int * int -> int t

 val inList : 'a list -> 'a t
 val inVector : 'a vector -> 'a t
 val inArray : 'a array -> 'a t

 val using : ('a, 'b) StringCvt.reader -> 'b -> 'a t

 val when : 'a t * ('a -> bool) -> 'a t
 val by : 'a t * ('a -> 'b) -> 'b t
 val @@ : 'a t * 'a t -> 'a t
 val ** : 'a t * 'b t -> ('a, 'b) product t

 val for : 'a -> 'a
end
```

Several of the above combinators are meant to be used as infix operators. Here is a set of suitable infix declarations:

```
infix 2 to downto
infix 1 @@ when by
infix 0 >>= **
```

A few notes are in order:

- The `'a t` type constructor with the `return` and `>>=` operators forms a monad.
- The `to` and `downto` combinators will omit the upper bound of the range.
- `for` is the identity function. It is purely for syntactic sugar and is not strictly required.
- The `@@` combinator produces an iterator for the concatenation of the given iterators.

- The `**` combinator produces an iterator for the Cartesian product of the given iterators.
  - See [ProductType](#) for the type constructor `('a, 'b) product` used in the type of the iterator produced by `**`.
- The `using` combinator allows one to iterate over slices, streams and many other kinds of sequences.
- `when` is the filtering combinator. The name `when` is inspired by [OCaml](#)'s guard clauses.
- `by` is the mapping combinator.

The below implementation of the ITER-signature makes use of the following basic combinators:

```
fun const x _ = x
fun flip f x y = f y x
fun id x = x
fun opt fno fso = fn NONE => fno () | SOME ? => fso ?
fun pass x f = f x
```

Here is an implementation the ITER-signature:

```
structure Iter :> ITER =
 struct
 type 'a t = ('a -> unit) -> unit

 val return = pass
 fun (iA >>= a2iB) f = iA (flip a2iB f)

 val none = ignore

 fun (l to u) f = let fun `l = if l<u then (f l; `(l+1)) else () in `l end
 fun (u downto l) f = let fun `u = if u>l then (f (u-1); `(u-1)) else () in `u end

 fun inList ? = flip List.app ?
 fun inVector ? = flip Vector.app ?
 fun inArray ? = flip Array.app ?

 fun using get s f = let fun `s = opt (const ()) (fn (x, s) => (f x; `s)) (get s) in `s ←
 end

 fun (iA when p) f = iA (fn a => if p a then f a else ())
 fun (iA by g) f = iA (f o g)
 fun (iA @@ iB) f = (iA f : unit; iB f)
 fun (iA ** iB) f = iA (fn a => iB (fn b => f (a & b)))

 val for = id
 end
```

Note that some of the above combinators (e.g. `**`) could be expressed in terms of the other combinators, most notably `return` and `>>=`. Another implementation issue worth mentioning is that `downto` is written specifically to avoid computing `l-1`, which could cause an Overflow.

To use the above combinators the `Iter`-structure needs to be opened

```
open Iter
```

and one usually also wants to declare the infix status of the operators as shown earlier.

Here is an example that illustrates some of the features:

```
for (0 to 10 when (fn x => x mod 3 <> 0) ** inList ["a", "b"] ** 2 downto 1 by real)
 (fn x & y & z =>
 print ("(^Int.toString x^", "\"^y^\"", "^Real.toString z^")\n"))
```

Using the `Iter` combinators one can easily produce more complicated iterators. For example, here is an iterator over a "triangle":

```
fun triangle (l, u) = l to u >>= (fn i => i to u >>= (fn j => return (i, j)))
```

---

## 77 FrontEnd

[FrontEnd](#) is a translation pass from source to the [AST IntermediateLanguage](#).

### 77.1 Description

This pass performs lexing and parsing to produce an abstract syntax tree.

### 77.2 Implementation

- [front-end.sig](#)
- [front-end.fun](#)

### 77.3 Details and Notes

The lexer is produced by [MLLex](#) from [ml.lex](#).

The parser is produced by [MLYacc](#) from [ml.grm](#).

The specifications for the lexer and parser were originally taken from [SML/NJ](#) (version 109.32), but have been heavily modified since then.

---

## 78 FSharp

**F#** is a functional programming language developed at Microsoft Research. F# was partly inspired by the [OCaml](#) language and shares some common core constructs with it. F# is integrated with Visual Studio 2010 as a first-class language.

---

## 79 FunctionalRecordUpdate

Functional record update is the copying of a record while replacing the values of some of the fields. [Standard ML](#) does not have explicit syntax for functional record update. We will show below how to implement functional record update in SML, with a little boilerplate code.

As an example, the functional update of the record

```
{a = 13, b = 14, c = 15}
```

with `c = 16` yields a new record

```
{a = 13, b = 14, c = 16}
```

Functional record update also makes sense with multiple simultaneous updates. For example, the functional update of the record above with `a = 18`, `c = 19` yields a new record

```
{a = 18, b = 14, c = 19}
```

One could easily imagine an extension of the SML that supports functional record update. For example

```
e with {a = 16, b = 17}
```

would create a copy of the record denoted by `e` with field `a` replaced with `16` and `b` replaced with `17`.

Since there is no such syntax in SML, we now show how to implement functional record update directly. We first give a simple implementation that has a number of problems. We then give an advanced implementation, that, while complex underneath, is a reusable library that admits simple use.

### 79.1 Simple implementation

To support functional record update on the record type

```
{a: 'a, b: 'b, c: 'c}
```

first, define an update function for each component.

```
fun withA ({a = _, b, c}, a) = {a = a, b = b, c = c}
fun withB ({a, b = _, c}, b) = {a = a, b = b, c = c}
fun withC ({a, b, c = _}, c) = {a = a, b = b, c = c}
```

Then, one can express `e with {a = 16, b = 17}` as

```
withB (withA (e, 16), 17)
```

With infix notation

```
infix withA withB withC
```

the syntax is almost as concise as a language extension.

```
e withA 16 withB 17
```

This approach suffers from the fact that the amount of boilerplate code is quadratic in the number of record fields. Furthermore, changing, adding, or deleting a field requires time proportional to the number of fields (because each `with<L>` function must be changed). It is also annoying to have to define a `with<L>` function, possibly with a fixity declaration, for each field.

Fortunately, there is a solution to these problems.

## 79.2 Advanced implementation

Using `Fold` one can define a family of `makeUpdate<N>` functions and single *update* operator `U` so that one can define a functional record update function for any record type simply by specifying a (trivial) isomorphism between that type and function argument list. For example, suppose that we would like to do functional record update on records with fields `a` and `b`. Then one defines a function `updateAB` as follows.

```
val updateAB =
 fn z =>
 let
 fun from v1 v2 = {a = v1, b = v2}
 fun to f {a = v1, b = v2} = f v1 v2
 in
 makeUpdate2 (from, from, to)
 end
 z
```

The functions `from` (think *from function arguments*) and `to` (think *to function arguments*) specify an isomorphism between `a,b` records and function arguments. There is a second use of `from` to work around the lack of [first-class polymorphism](#) in SML.

With the definition of `updateAB` in place, the following expressions are valid.

```
updateAB {a = 13, b = "hello"} (set#b "goodbye") $
updateAB {a = 13.5, b = true} (set#b false) (set#a 12.5) $
```

As another example, suppose that we would like to do functional record update on records with fields `b`, `c`, and `d`. Then one defines a function `updateBCD` as follows.

```
val updateBCD =
 fn z =>
 let
 fun from v1 v2 v3 = {b = v1, c = v2, d = v3}
 fun to f {b = v1, c = v2, d = v3} = f v1 v2 v3
 in
 makeUpdate3 (from, from, to)
 end
 z
```

With the definition of `updateBCD` in place, the following expression is valid.

```
updateBCD {b = 1, c = 2, d = 3} (set#c 4) (set#c 5) $
```

Note that not all fields need be updated and that the same field may be updated multiple times. Further note that the same `set` operator is used for all update functions (in the above, for both `updateAB` and `updateBCD`).

In general, to define a functional-record-update function on records with fields `f1`, `f2`, ..., `fN`, use the following template.

```
val update =
 fn z =>
 let
 fun from v1 v2 ... vn = {f1 = v1, f2 = v2, ..., fn = vn}
 fun to f {f1 = v1, f2 = v2, ..., fn = vn} = v1 v2 ... vn
 in
 makeUpdateN (from, from, to)
 end
 z
```

With this, one can update a record as follows.

```
update {f1 = v1, ..., fn = vn} (set#f1 v1) ... (set#fN vN) $
```

### 79.3 The `FunctionalRecordUpdate` structure

Here is the implementation of functional record update.

```

structure FunctionalRecordUpdate =
 struct
 local
 fun next g (f, z) x = g (f x, z)
 fun f1 (f, z) x = f (z x)
 fun f2 z = next f1 z
 fun f3 z = next f2 z

 fun c0 from = from
 fun c1 from = c0 from f1
 fun c2 from = c1 from f2
 fun c3 from = c2 from f3

 fun makeUpdate cX (from, from', to) record =
 let
 fun ops () = cX from'
 fun vars f = to f record
 in
 Fold.fold ((vars, ops), fn (vars, _) => vars from)
 end
 in
 fun makeUpdate0 z = makeUpdate c0 z
 fun makeUpdate1 z = makeUpdate c1 z
 fun makeUpdate2 z = makeUpdate c2 z
 fun makeUpdate3 z = makeUpdate c3 z

 fun upd z = Fold.step2 (fn (s, f, (vars, ops)) => (fn out => vars (s (ops ())) (out ←
 , f)), ops)) z
 fun set z = Fold.step2 (fn (s, v, (vars, ops)) => (fn out => vars (s (ops ())) (out ←
 , fn _ => v)), ops)) z
 end
 end
end

```

The idea of `makeUpdate` is to build a record of functions which can replace the contents of one argument out of a list of arguments. The functions `f<X>` replace the 0th, 1st, ... argument with their argument `z`. The `c<X>` functions pass the first `X` `f` functions to the record constructor.

The `#field` notation of Standard ML allows us to select the map function which replaces the corresponding argument. By converting the record to an argument list, feeding that list through the selected map function and piping the list into the record constructor, functional record update is achieved.

### 79.4 Efficiency

With MLton, the efficiency of this approach is as good as one would expect with the special syntax. Namely a sequence of updates will be optimized into a single record construction that copies the unchanged fields and fills in the changed fields with their new values.

Before Sep 14, 2009, this page advocated an alternative implementation of `FunctionalRecordUpdate`. However, the old structure caused exponentially increasing compile times. We advise you to switch to the newer version.

### 79.5 Applications

Functional record update can be used to implement labelled [optional arguments](#).

## 80 fxp

`fxp` is an XML parser written in Standard ML.

It has a `patch` to compile with MLton.

---

## 81 GarbageCollection

For a good introduction and overview to garbage collection, see [Jones99](#).

MLton's garbage collector uses copying, mark-compact, and generational collection, automatically switching between them at run time based on the amount of live data relative to the amount of RAM. The runtime system tries to keep the heap within RAM if at all possible.

MLton's copying collector is a simple, two-space, breadth-first, Cheney-style collector. The design for the generational and mark-compact GC is based on [Sansom91](#).

### 81.1 Design notes

- <http://www.mlton.org/pipermail/mlton/2002-May/012420.html>  
object layout and header word design

### 81.2 Also see

- [Regions](#)
-

## 82 GenerativeDatatype

In [Standard ML](#), datatype declarations are said to be *generative*, because each time a datatype declaration is evaluated, it yields a new type. Thus, any attempt to mix the types will lead to a type error at compile-time. The following program, which does not type check, demonstrates this.

```
functor F () =
 struct
 datatype t = T
 end
structure S1 = F ()
structure S2 = F ()
val _: S1.t -> S2.t = fn x => x
```

Generativity also means that two different datatype declarations define different types, even if they define identical constructors. The following program does not type check due to this.

```
datatype t = A | B
val a1 = A
datatype t = A | B
val a2 = A
val _ = if true then a1 else a2
```

### 82.1 Also see

- [GenerativeException](#)

## 83 GenerativeException

In [Standard ML](#), exception declarations are said to be *generative*, because each time an exception declaration is evaluated, it yields a new exception.

The following program demonstrates the generativity of exceptions.

```
exception E
val e1 = E
fun isE1 (e: exn): bool =
 case e of
 E => true
 | _ => false
exception E
val e2 = E
fun isE2 (e: exn): bool =
 case e of
 E => true
 | _ => false
fun pb (b: bool): unit =
 print (concat [Bool.toString b, "\n"])
val () = (pb (isE1 e1)
 ; pb (isE1 e2)
 ; pb (isE2 e1)
 ; pb (isE2 e2))
```

In the above program, two different exception declarations declare an exception `E` and a corresponding function that returns `true` only on that exception. Although declared by syntactically identical exception declarations, `e1` and `e2` are different exceptions. The program, when run, prints `true, false, false, true`.

A slight modification of the above program shows that even a single exception declaration yields a new exception each time it is evaluated.

```
fun f (): exn * (exn -> bool) =
 let
 exception E
 in
 (E, fn E => true | _ => false)
 end
val (e1, isE1) = f ()
val (e2, isE2) = f ()
fun pb (b: bool): unit =
 print (concat [Bool.toString b, "\n"])
val () = (pb (isE1 e1)
 ; pb (isE1 e2)
 ; pb (isE2 e1)
 ; pb (isE2 e2))
```

Each call to `f` yields a new exception and a function that returns `true` only on that exception. The program, when run, prints `true, false, false, true`.

### 83.1 Type Safety

Exception generativity is required for type safety. Consider the following valid SML program.

```
fun f (): ('a -> exn) * (exn -> 'a) =
 let
 exception E of 'a
 in
 (E, fn E x => x | _ => raise Fail "f")
 end
```

```
end
fun cast (a: 'a): 'b =
 let
 val (make: 'a -> exn, _) = f ()
 val (_, get: exn -> 'b) = f ()
 in
 get (make a)
 end
val _ = ((cast 13): int -> int) 14
```

If exceptions weren't generative, then each call `f ()` would yield the same exception constructor `E`. Then, our `cast` function could use `make: 'a -> exn` to convert any value into an exception and then `get: exn -> 'b` to convert that exception to a value of arbitrary type. If `cast` worked, then we could cast an integer as a function and apply. Of course, because of generative exceptions, this program raises `Fail "f"`.

## 83.2 Applications

The `exn` type is effectively a [universal type](#).

## 83.3 Also see

- [GenerativeDatatype](#)

## 84 Git

Git is a distributed version control system. The MLton project currently uses Git to maintain its [source code](#).

Here are some online Git resources.

- [Reference Manual](#)
  - [ProGit, by Scott Chacon](#)
-

## 85 Glade

Glade is a tool for generating Gtk user interfaces.

WesleyTerpstra is working on a Glade→mGTK converter.

- <http://www.mlton.org/pipermail/mlton/2004-December/016865.html>
-

## 86 Globalize

[Globalize](#) is an analysis pass for the [SXML IntermediateLanguage](#), invoked from [ClosureConvert](#).

### 86.1 Description

This pass marks values that are constant, allowing [ClosureConvert](#) to move them out to the top level so they are only evaluated once and do not appear in closures.

### 86.2 Implementation

- [globalize.sig](#)
- [globalize.fun](#)

### 86.3 Details and Notes

---

## 87 GnuMP

The **GnuMP** library (GNU multiprecision library) is a library for arbitrary precision integer arithmetic. MLton uses the GnuMP library to implement the [Basis Library](#) `IntInf` module.

### 87.1 Known issues

- There is a known problem with the GnuMP library (prior to version 4.2.x), where it requires a lot of stack space for some computations, e.g. `IntInf.toString` of a million digit number. If you run with stack size limited, you may see a segfault in such programs. This problem is mentioned in the [GnuMP FAQ](#), where they describe two solutions.
  - Increase (or unlimit) your stack space. From your program, use `setrlimit`, or from the shell, use `ulimit`.
  - Configure and rebuild `libgmp` with `--disable-alloca`, which will cause it to allocate temporaries using `malloc` instead of on the stack.
- On some platforms, the GnuMP library may be configured to use one of multiple ABIs (Application Binary Interfaces). For example, on some 32-bit architectures, GnuMP may be configured to represent a limb as either a 32-bit `long` or as a 64-bit `long long`. Similarly, GnuMP may be configured to use specific CPU features.

In order to efficiently use the GnuMP library, MLton represents an `IntInf.int` value in a manner compatible with the GnuMP library's representation of a limb. Hence, it is important that MLton and the GnuMP library agree upon the representation of a limb.

- When using a source package of MLton, building will detect the GnuMP library's representation of a limb.
- When using a binary package of MLton that is dynamically linked against the GnuMP library, the build machine and the install machine must have the GnuMP library configured with the same representation of a limb. (On the other hand, the build machine need not have the GnuMP library configured with CPU features compatible with the install machine.)
- When using a binary package of MLton that is statically linked against the GnuMP library, the build machine and the install machine need not have the GnuMP library configured with the same representation of a limb. (On the other hand, the build machine must have the GnuMP library configured with CPU features compatible with the install machine.)

However, MLton will be configured with the representation of a limb from the GnuMP library of the build machine. Executables produced by MLton will be incompatible with the GnuMP library of the install machine. To *reconfigure* MLton with the representation of a limb from the GnuMP library of the install machine, one must edit:

```
/usr/lib/mlton/self/sizes
```

changing the

```
mplimb = ??
```

entry so that ?? corresponds to the bytes in a limb; and, one must edit:

```
/usr/lib/mlton/sml/basis/config/c/arch-os/c-types.sml
```

changing the

```
(* from "gmp.h" *)
structure C_MPLimb = struct open Word?? type t = word end
functor C_MPLimb_ChooseWordN (A: CHOOSE_WORDN_ARG) = ChooseWordN_Word?? (A)
```

entries so that ?? corresponds to the bits in a limb.

## 88 Google Summer of Code (2013)

### 88.1 Mentors

The following developers have agreed to serve as mentors for the 2013 Google Summer of Code:

- [Matthew Fluet](#)
- [Lukasz \(Luke\) Ziarek](#)
- [Suresh Jagannathan](#)

### 88.2 Ideas List

#### 88.2.1 Implement a Partial Redundancy Elimination (PRE) Optimization

Partial redundancy elimination (PRE) is a program transformation that removes operations that are redundant on some, but not necessarily all paths, through the program. PRE can subsume both common subexpression elimination and loop-invariant code motion, and is therefore a potentially powerful optimization. However, a naïve implementation of PRE on a program in static single assignment (SSA) form is unlikely to be effective. This project aims to adapt and implement the SSAPRE algorithm(s) of Thomas VanDrunen in MLton's SSA intermediate language.

Background:

- [Anticipation-based partial redundancy elimination for static single assignment form](#); Thomas VanDrunen and Antony L. Hosking
- [Partial Redundancy Elimination for Global Value Numbering](#); Thomas VanDrunen
- [Value-Based Partial Redundancy Elimination](#); Thomas VanDrunen and Antony L. Hosking
- [Partial redundancy elimination in SSA form](#); Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred Chow

Recommended Skills: SML programming experience; some middle-end compiler experience

#### 88.2.2 Design and Implement a Heap Profiler

A heap profile is a description of the space usage of a program. A heap profile is concerned with the allocation, retention, and deallocation (via garbage collection) of heap data during the execution of a program. A heap profile can be used to diagnose performance problems in a functional program that arise from space leaks. This project aims to design and implement a heap profiler for MLton compiled programs.

Background:

- [GCspy: an adaptable heap visualisation framework](#); Tony Printezis and Richard Jones
- [New dimensions in heap profiling](#); Colin Runciman and Niklas Røjemo
- [Heap profiling for space efficiency](#); Colin Runciman and Niklas Røjemo
- [Heap profiling of lazy functional programs](#); Colin Runciman and David Wakeling

Recommended Skills: C and SML programming experience; some experience with UI and visualization

---

### 88.2.3 Garbage Collector Improvements

The garbage collector plays a significant role in the performance of functional languages. Garbage collect too often, and program performance suffers due to the excessive time spent in the garbage collector. Garbage collect not often enough, and program performance suffers due to the excessive space used by the uncollected garbage. One particular issue is ensuring that a program utilizing a garbage collector "plays nice" with other processes on the system, by not using too much or too little physical memory. While there are some reasonable theoretical results about garbage collections with heaps of fixed size, there seems to be insufficient work that really looks carefully at the question of dynamically resizing the heap in response to the live data demands of the application and, similarly, in response to the behavior of the operating system and other processes. This project aims to investigate improvements to the memory behavior of MLton compiled programs through better tuning of the garbage collector.

Background:

- [Automated Heap Sizing in the Poly/ML Runtime \(Position Paper\)](#); David White, Jeremy Singer, Jonathan Aitken, and David Matthews
- [Isla Vista Heap Sizing: Using Feedback to Avoid Paging](#); Chris Grzegorzcyk, Sunil Soman, Chandra Krintz, and Rich Wolski
- [Controlling garbage collection and heap growth to reduce the execution time of Java applications](#); Tim Brecht, Eshrat Arjomandi, Chang Li, and Hang Pham
- [Garbage collection without paging](#); Matthew Hertz, Yi Feng, and Emery D. Berger
- [Automatic heap sizing: taking real memory into account](#); Ting Yang, Matthew Hertz, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss

Recommended Skills: C programming experience; some operating systems and/or systems programming experience; some compiler and garbage collector experience

### 88.2.4 Implement Successor ML Language Features

Any programming language, including Standard ML, can be improved. The community has identified a number of modest extensions and revisions to the Standard ML programming language that would likely prove useful in practice. This project aims to implement these language features in the MLton compiler.

Background:

- [Successor ML](#)
- [HaMLet \(Successor ML\)](#)
- [A critique of Standard ML](#); Andrew W. Appel

Recommended Skills: SML programming experience; some front-end compiler experience (i.e., scanners and parsers)

### 88.2.5 Implement Source-level Debugging

Debugging is a fact of programming life. Unfortunately, most SML implementations (including MLton) provide little to no source-level debugging support. This project aims to add basic to intermediate source-level debugging support to the MLton compiler. MLton already supports source-level profiling, which can be used to attribute bytes allocated or time spent in source functions. It should be relatively straightforward to leverage this source-level information into basic source-level debugging support, with the ability to set/unset breakpoints and step through declarations and functions. It may be possible to also provide intermediate source-level debugging support, with the ability to inspect in-scope variables of basic types (e.g., types compatible with MLton's foreign function interface).

Background:

- [MLton — How Profiling Works](#)

- [MLton — Foreign Function Interface Types](#)
- [DWARF Debugging Standard](#)
- [STABS Debugging Format](#)

Recommended Skills: SML programming experience; some compiler experience

### 88.2.6 SIMD Primitives

Most modern processors offer some direct support for SIMD (Single Instruction, Multiple Data) operations, such as Intel's MMX/SSE instructions, AMD's 3DNow! instructions, and IBM's AltiVec. Such instructions are particularly useful for multimedia, scientific, and cryptographic applications. This project aims to add preliminary support for vector data and vector operations to the MLton compiler. Ideally, after surveying SIMD instruction sets and SIMD support in other compilers, a core set of SIMD primitives with broad architecture and compiler support can be identified. After adding SIMD primitives to the core compiler and carrying them through to the various backends, there will be opportunities to design and implement an SML library that exposes the primitives to the SML programmer as well as opportunities to design and implement auto-vectorization optimizations.

Background:

- [SIMD](#)
- [Auto-vectorization in GCC](#)
- [Auto-vectorization in LLVM](#)

Recommended Skills: SML programming experience; some compiler experience; some computer architecture experience

### 88.2.7 RTOS Support

This project entails porting the MLton compiler to RTOSs such as: RTEMS, RT Linux, and FreeRTOS. The project will include modifications to the MLton build and configuration process. Students will need to extend the MLton configuration process for each of the RTOSs. The MLton compilation process will need to be extended to invoke the C cross compilers the RTOSs provide for embedded support. Test scripts for validation will be necessary and these will need to be run in emulators for supported architectures.

Recommended Skills: C programming experience; some scripting experience

### 88.2.8 Region Based Memory Management

Region based memory management is an alternative automatic memory management scheme to garbage collection. Regions can be inferred by the compiler (e.g., Cyclone and MLKit) or provided to the programmer through a library. Since many students do not have extensive experience with compilers we plan on adopting the later approach. Creating a viable region based memory solution requires the removal of the GC and changes to the allocator. Additionally, write barriers will be necessary to ensure references between two ML objects is never established if the left hand side of the assignment has a longer lifetime than the right hand side. Students will need to come up with an appropriate interface for creating, entering, and exiting regions (examples include RTSJ scoped memory and SCJ scoped memory).

Background:

- Cyclone
- MLKit
- RTSJ + SCJ scopes

Recommended Skills: SML programming experience; C programming experience; some compiler and garbage collector experience

---

### 88.2.9 Integration of Multi-MLton

**MultiMLton** is a compiler and runtime environment that targets scalable multicore platforms. It is an extension of MLton. It combines new language abstractions and associated compiler analyses for expressing and implementing various kinds of fine-grained parallelism (safe futures, speculation, transactions, etc.), along with a sophisticated runtime system tuned to efficiently handle large numbers of lightweight threads. The core stable features of MultiMLton will need to be integrated with the latest MLton public release. Certain experimental features, such as support for the Intel SCC and distributed runtime will be omitted. This project requires students to understand the delta between the MultiMLton code base and the MLton code base. Students will need to create build and configuration scripts for MLton to enable MultiMLton features.

Background

- [MultiMLton — Publications](#)

Recommended Skills: SML programming experience; C programming experience; some compiler experience

---

## 89 HaMLet

HaMLet is a [Standard ML implementation](#). It is intended as reference implementation of [The Definition of Standard ML \(Revised\)](#) and not for serious practical work.

---

## 90 HenryCejtin

I was one of the original developers of Mathematica (actually employee #1). My background is a combination of mathematics and computer science. Currently I am doing various things in Chicago.

---

## 91 History

In April 1997, Stephen Weeks wrote a defunctorizer for Standard ML and integrated it with SML/NJ. The defunctorizer used SML/NJ's visible compiler and operated on the `AST` intermediate representation produced by the SML/NJ front end. Experiments showed that defunctorization gave a speedup of up to six times over separate compilation and up to two times over batch compilation without functor expansion.

In August 1997, we began development of an independent compiler for SML. At the time the compiler was called `smlc`. By October, we had a working monomorphiser. By November, we added a polyvariant higher-order control-flow analysis. At that point, MLton was about 10,000 lines of code.

Over the next year and half, `smlc` morphed into a full-fledged compiler for SML. It was renamed MLton, and first released in March 1999.

From the start, MLton has been driven by whole-program optimization and an emphasis on performance. Also from the start, MLton has had a fast C FFI and `IntInf` based on the GNU multiprecision library. At its first release, MLton was 48,006 lines.

Between the March 1999 and January 2002, MLton grew to 102,541 lines, as we added a native code generator, `mlex`, `mlyacc`, a profiler, many optimizations, and many libraries including threads and signal handling.

During 2002, MLton grew to 112,204 lines and we had releases in April and September. We added support for cross compilation and used this to enable MLton to run on Cygwin/Windows and FreeBSD. We also made improvements to the garbage collector, so that it now works with large arrays and up to 4G of memory and so that it automatically uses copying, mark-compact, or generational collection depending on heap usage and RAM size. We also continued improvements to the optimizer and libraries.

During 2003, MLton grew to 122,299 lines and we had releases in March and July. We extended the profiler to support source-level profiling of time and allocation and to display call graphs. We completed the Basis Library implementation, and added new MLton-specific libraries for weak pointers and finalization. We extended the FFI to allow callbacks from C to SML. We added support for the Sparc/Solaris platform, and made many improvements to the C code generator.

## 92 HowProfilingWorks

Here's how [Profiling](#) works. If profiling is on, the front end (elaborator) inserts `Enter` and `Leave` statements into the source program for function entry and exit. For example,

```
fun f n = if n = 0 then 0 else 1 + f (n - 1)
```

becomes

```
fun f n =
 let
 val () = Enter "f"
 val res = (if n = 0 then 0 else 1 + f (n - 1))
 handle e => (Leave "f"; raise e)
 val () = Leave "f"
 in
 res
 end
```

Actually there is a bit more information than just the source function name; there is also lexical nesting and file position.

Most of the middle of the compiler ignores, but preserves, `Enter` and `Leave`. However, so that profiling preserves tail calls, the [SSA shrinker](#) has an optimization that notices when the only operations that cause a call to be a nontail call are profiling operations, and if so, moves them before the call, turning it into a tail call. If you observe a program that has a tail call that appears to be turned into a nontail when compiled with profiling, please [report a bug](#).

There is the `checkProf` function in `type-check.fun`, which checks that the `Enter/Leave` statements match up.

In the backend, just before translating to the [Machine IL](#), the profiler uses the `Enter/Leave` statements to infer the "local" portion of the control stack at each program point. The profiler then removes the `Enters/Leaves` and inserts different information depending on which kind of profiling is happening. For time profiling (with the native codegen), the profiler inserts labels that cover the code (i.e. each statement has a unique label in its basic block that prefixes it) and associates each label with the local control stack. For time profiling (with the C and bytecode codegens), the profiler inserts code that sets a global field that records the local control stack. For allocation profiling, the profiler inserts calls to a C function that will maintain byte counts. With stack profiling, the profiler also inserts a call to a C function at each nontail call in order to maintain information at runtime about what SML functions are on the stack.

At run time, the profiler associates counters (either clock ticks or byte counts) with source functions. When the program finishes, the profiler writes the counts out to the `mlmon.out` file. Then, `mlprof` uses source information stored in the executable to associate the counts in the `mlmon.out` file with source functions.

For time profiling, the profiler catches the `SIGPROF` signal 100 times per second and increments the appropriate counter, determined by looking at the label prefixing the current program counter and mapping that to the current source function.

### 92.1 Caveats

There may be a few missed clock ticks or bytes allocated at the very end of the program after the data is written.

Profiling has not been tested with signals or threads. In particular, stack profiling may behave strangely.

## 93 Identifier

In [Standard ML](#), there are syntactically two kinds of identifiers.

- **Alphanumeric:** starts with a letter or prime ( `'` ) and is followed by letters, digits, primes and underbars ( `_` ).

Examples: `abc`, `ABC123`, `Abc_123`, `' a`.

- **Symbolic:** a sequence of the following

```
! % & $ # + - / : < = > ? @ | ~ ` ^ | *
```

Examples: `+=`, `<=`, `>>`, `$`.

With the exception of `=`, reserved words can not be identifiers.

There are a number of different classes of identifiers, some of which have additional syntactic rules.

- Identifiers not starting with a prime.
  - value identifier (includes variables and constructors)
  - type constructor
  - structure identifier
  - signature identifier
  - functor identifier
- Identifiers starting with a prime.
  - type variable
- Identifiers not starting with a prime and numeric labels ( `1`, `2`, ... ).
  - record label

## 94 Immutable

Immutable means not [mutable](#) and is an adjective meaning "can not be modified". Most values in [Standard ML](#) are immutable. For example, constants, tuples, records, lists, and vectors are all immutable.

---

## 95 ImperativeTypeVariable

In [Standard ML](#), an imperative type variable is a type variable whose second character is a digit, as in '1a or '2b. Imperative type variables were used as an alternative to the [ValueRestriction](#) in an earlier version of SML, but no longer play a role. They are treated exactly as other type variables.

---

## 96 ImplementExceptions

[ImplementExceptions](#) is a pass for the [SXML IntermediateLanguage](#), invoked from [SXMLSimplify](#).

### 96.1 Description

This pass implements exceptions.

### 96.2 Implementation

- `implement-exceptions.fun`

### 96.3 Details and Notes

---

## 97 ImplementHandlers

[ImplementHandlers](#) is a pass for the [RSSA IntermediateLanguage](#), invoked from [RSSASimplify](#).

### 97.1 Description

This pass implements the (threaded) exception handler stack.

### 97.2 Implementation

- `implement-handlers.fun`

### 97.3 Details and Notes

---

## 98 ImplementProfiling

[ImplementProfiling](#) is a pass for the [RSSA IntermediateLanguage](#), invoked from [RSSASimplify](#).

### 98.1 Description

This pass implements profiling.

### 98.2 Implementation

- `implement-profiling.fun`

### 98.3 Details and Notes

See [HowProfilingWorks](#).

---

## 99 ImplementSuffix

`ImplementSuffix` is a pass for the [SXML IntermediateLanguage](#), invoked from [SXMLSimplify](#).

### 99.1 Description

This pass implements the `TopLevel_setSuffix` primitive, which installs a function to exit the program.

### 99.2 Implementation

- `implement-suffix.fun`

### 99.3 Details and Notes

`ImplementSuffix` works by introducing a new `ref` cell to contain the function of type `unit -> unit` that should be called on program exit.

- The following code (appropriately alpha-converted) is appended to the beginning of the [SXML](#) program:

```
val z_0 =
 fn a_0 =>
 let
 val x_0 =
 "oplevel suffix not installed"
 val x_1 =
 MLton_bug (x_0)
 in
 x_1
 end
val topLevelSuffixCell =
 Ref_ref (z_0)
```

- Any occurrence of

```
val x_0 =
 TopLevel_setSuffix (f_0)
```

is rewritten to

```
val x_0 =
 Ref_assign (topLevelSuffixCell, f_0)
```

- The following code (appropriately alpha-converted) is appended to the end of the [SXML](#) program:

```
val f_0 =
 Ref_deref (topLevelSuffixCell)
val z_0 =
 ()
val x_0 =
 f_0 z_0
```

## 100 Infixing Operators

Fixity specifications are not part of signatures in [Standard ML](#). When one wants to use a module that provides functions designed to be used as infix operators there are several obvious alternatives:

- Use only prefix applications. Unfortunately there are situations where infix applications lead to considerably more readable code.
- Make the fixity declarations at the top-level. This may lead to collisions and may be unsustainable in a large project. Pollution of the top-level should be avoided.
- Make the fixity declarations at each scope where you want to use infix applications. The duplication becomes inconvenient if the operators are widely used. Duplication of code should be avoided.
- Use non-standard extensions, such as the [ML Basis system](#) to control the scope of fixity declarations. This has the obvious drawback of reduced portability.
- Reuse existing infix operator symbols ( $\wedge$ ,  $+$ ,  $-$ , ...). This can be convenient when the standard operators aren't needed in the same scope with the new operators. On the other hand, one is limited to the standard operator symbols and the code may appear confusing.

None of the obvious alternatives is best in every case. The following describes a slightly less obvious alternative that can sometimes be useful. The idea is to approximate Haskell's special syntax for treating any identifier enclosed in grave accents (backquotes) as an infix operator. In Haskell, instead of writing the prefix application  $f\ x\ y$  one can write the infix application  $x\ `f`\ y$ .

### 100.1 Infixing operators

Let's first take a look at the definitions of the operators:

```
infix 3 <\ fun x <\ f = fn y => f (x, y) (* Left section *)
infix 3 \> fun f \> y = f y (* Left application *)
infixr 3 /> fun f /> y = fn x => f (x, y) (* Right section *)
infixr 3 </ fun x </ f = f x (* Right application *)

infix 2 o (* See motivation below *)
infix 0 :=
```

The left and right sectioning operators,  $<\$  and  $/>$ , are useful in SML for partial application of infix operators. [ML For the Working Programmer](#) describes curried functions `secl` and `secr` for the same purpose on pages 179-181. For example,

```
List.map (op- /> y)
```

is a function for subtracting  $y$  from a list of integers and

```
List.exists (x <\ op=)
```

is a function for testing whether a list contains an  $x$ .

Together with the left and right application operators,  $\backslash>$  and  $</$ , the sectioning operators provide a way to treat any binary function (i.e. a function whose domain is a pair) as an infix operator. In general,

```
x0 <\f1\> x1 <\f2\> x2 ... <\fN\> xN = fN (... f2 (f1 (x0, x1), x2) ... , xN)
```

and

```
xN </fN/> ... x2 </f2/> x1 </f1/> x0 = fN (xN, ... f2 (x2, f1 (x1, x0)) ...)
```

### 100.1.1 Examples

As a fairly realistic example, consider providing a function for sequencing comparisons:

```
structure Order (* ... *) =
 struct
 (* ... *)
 val orWhenEq = fn (EQUAL, th) => th ()
 | (other, _) => other
 (* ... *)
 end
```

Using `orWhenEq` and the infixing operators, one can write a `compare` function for triples as

```
fun compare (fad, fbe, fcf) ((a, b, c), (d, e, f)) =
 fad (a, d) <\Order.orWhenEq\> `fbe (b, e) <\Order.orWhenEq\> `fcf (c, f)
```

where ``` is defined as

```
fun `f x = fn () => f x
```

Although `orWhenEq` can be convenient (try rewriting the above without it), it is probably not useful enough to be defined at the top level as an infix operator. Fortunately we can use the infixing operators and don't have to.

Another fairly realistic example would be to use the infixing operators with the technique described on the [Printf](#) page. Assuming that you would have a `Printf` module binding `printf`, ```, and formatting combinators named `int` and `string`, you could write

```
let open Printf in
 printf ("Here's an int "<\int\>" and a string "<\string\>".") 13 "foo" end
```

without having to duplicate the fixity declarations. Alternatively, you could write

```
P.printf (P."Here's an int "<\P.int\>" and a string "<\P.string\>".") 13 "foo"
```

assuming you have the made the binding

```
structure P = Printf
```

## 100.2 Application and piping operators

The left and right application operators may also provide some notational convenience on their own. In general,

```
f \> x1 \> ... \> xN = f x1 ... xN
```

and

```
xN </ ... </ x1 </ f = f x1 ... xN
```

If nothing else, both of them can eliminate parentheses. For example,

```
foo (1 + 2) = foo \> 1 + 2
```

The left and right application operators are related to operators that could be described as the right and left piping operators:

```
infix 1 >| val op>| = op</ (* Left pipe *)
infixr 1 |< val op|< = op\> (* Right pipe *)
```

As you can see, the left and right piping operators, `>|` and `|<`, are the same as the right and left application operators, respectively, except the associativities are reversed and the binding strength is lower. They are useful for piping data through a sequence of operations. In general,

```
x >| f1 >| ... >| fN = fN (... (f1 x) ...) = (fN o ... o f1) x
```

and

```
fN |< ... |< f1 |< x = fN (... (f1 x) ...) = (fN o ... o f1) x
```

The right piping operator, `|<`, is provided by the Haskell prelude as `$`. It can be convenient in CPS or continuation passing style.

A use for the left piping operator is with parsing combinators. In a strict language, like SML, eta-reduction is generally unsafe. Using the left piping operator, parsing functions can be formatted conveniently as

```
fun parsingFunc input =
 input >| (* ... *)
 || (* ... *)
 || (* ... *)
```

where `||` is supposed to be a combinator provided by the parsing combinator library.

### 100.3 About precedences

You probably noticed that we redefined the [precedences](#) of the function composition operator `o` and the assignment operator `:=`. Doing so is not strictly necessary, but can be convenient and should be relatively safe. Consider the following motivating examples from [Wesley W. Terpstra](#) relying on the redefined precedences:

```
Word8.fromInt o Char.ord o s <\String.sub
(* Combining sectioning and composition *)

x := s <\String.sub\> i
(* Assigning the result of an infix application *)
```

In imperative languages, assignment usually has the lowest precedence (ignoring statement separators). The precedence of `:=` in the [Basis Library](#) is perhaps unnecessarily high, because an expression of the form `r := x` always returns a unit, which makes little sense to combine with anything. Dropping `:=` to the lowest precedence level makes it behave more like in other imperative languages.

The case for `o` is different. With the exception of `before` and `:=`, it doesn't seem to make much sense to use `o` with any of the operators defined by the [Basis Library](#) in an unparenthesized expression. This is simply because none of the other operators deal with functions. It would seem that the precedence of `o` could be chosen completely arbitrarily from the set  $\{1, \dots, 9\}$  without having any adverse effects with respect to other infix operators defined by the [Basis Library](#).

### 100.4 Design of the symbols

The closest approximation of Haskell's `x `f` y` syntax achievable in Standard ML would probably be something like `x `f` ^ y`, but `^` is already used for string concatenation by the [Basis Library](#). Other combinations of the characters ``` and `^` would be possible, but none seems clearly the best visually. The symbols `<\`, `\>`, `</`, and `/>` are reasonably concise and have a certain self-documenting appearance and symmetry, which can help to remember them. As the names suggest, the symbols of the piping operators `>|` and `|<` are inspired by Unix shell pipelines.

### 100.5 Also see

- [Utilities](#)

## 101 Inline

[Inline](#) is an optimization pass for the [SSA IntermediateLanguage](#), invoked from [SSASimplify](#).

### 101.1 Description

This pass inlines [SSA](#) functions using a size-based metric.

### 101.2 Implementation

- `inline.sig`
- `inline.fun`

### 101.3 Details and Notes

The [Inline](#) pass can be invoked to use one of three metrics:

- `NonRecursive(product, small)` — inline any function satisfying  $(\text{numCalls} - 1) * (\text{size} - \text{small}) \leq \text{product}$ , where `numCalls` is the static number of calls to the function and `size` is the size of the function.
  - `Leaf(size)` — inline any leaf function smaller than `size`
  - `LeafNoLoop(size)` — inline any leaf function without loops smaller than `size`
-

## 102 InsertLimitChecks

[InsertLimitChecks](#) is a pass for the [RSSA IntermediateLanguage](#), invoked from [RSSASimplify](#).

### 102.1 Description

This pass inserts limit checks.

### 102.2 Implementation

- `limit-check.fun`

### 102.3 Details and Notes

---

## 103 InsertSignalChecks

[InsertSignalChecks](#) is a pass for the [RSSA IntermediateLanguage](#), invoked from [RSSASimplify](#).

### 103.1 Description

This pass inserts signal checks.

### 103.2 Implementation

- [limit-check.fun](#)

### 103.3 Details and Notes

---

## 104 Installation

MLton runs on a variety of platforms and is distributed in both source and binary form. The format for the binary package depends on the platform. The binary package will install under `/usr` or `/usr/local`, depending on the platform. A `.tgz` or `.tbz` binary package should be extracted in the root directory. If you install MLton somewhere else, you must set the `lib` variable in the `/usr/bin/mlton` script to the directory that contains the libraries (`/usr/lib/mlton` by default).

MLton requires that you have the [GNU multiprecision](#) library installed on your machine. MLton must be able to find both the `gmp.h` include file and the `libgmp.a` (or `libgmp.so` or `libgmp.dylib`) library. If you see the error message `gmp.h: No such file or directory`, you should copy `gmp.h` to `/usr/lib/mlton/self/include`. If you see the error message `/usr/bin/ld: cannot find -lgmp`, you should add a `-link-opt -L` argument in the `/usr/bin/mlton` script so that the linker can find `libgmp`. If, for example, `libgmp.a` is in `/tmp`, then add `-link-opt -L/tmp`.

Installation of MLton creates the following files and directories.

- `/usr/bin/mllex`  
The [MLLex](#) lexer generator.
- `/usr/bin/mlnlffigen`  
The [ML-NLFFI](#) tool.
- `/usr/bin/mlprof`  
A [Profiling](#) tool.
- `/usr/bin/mlton`  
A script to call the compiler. This script may be moved anywhere, however, it makes use of files in `/usr/lib/mlton`.
- `/usr/bin/mlyacc`  
The [MLYacc](#) parser generator.
- `/usr/lib/mlton`  
Directory containing libraries and include files needed during compilation.
- `/usr/share/man/man1/mllex.1,mlnlffigen.1,mlprof.1,mlton.1,mlyacc.1`  
Man pages.
- `/usr/share/doc/mlton`  
Directory containing the user guide for MLton, `mllex`, and `mlyacc`, as well as example SML programs (in the `examples` dir), and license information.

### 104.1 Hello, World!

Once you have installed MLton, create a file called `hello-world.sml` with the following contents.

```
print "Hello, world!\n";
```

Now create an executable, `hello-world`, with the following command.

```
mlton hello-world.sml
```

You can now run `hello-world` to verify that it works. There are more small examples in `/usr/share/doc/mlton/examples`.

### 104.2 Installation on Cygwin

When installing the Cygwin `tgz`, you should use Cygwin's `bash` and `tar`. The use of an archiving tool that is not aware of Cygwin's mounts will put the files in the wrong place.

## 105 IntermediateLanguage

MLton uses a number of intermediate languages in translating from the input source program to low-level code. Here is a list in the order which they are translated to.

- [AST](#). Pretty close to the source.
  - [CoreML](#). Explicitly typed, no module constructs.
  - [XML](#). Polymorphic, [HigherOrder](#).
  - [SXML](#). SimplyTyped, [HigherOrder](#).
  - [SSA](#). SimplyTyped, [FirstOrder](#).
  - [SSA2](#). SimplyTyped, [FirstOrder](#).
  - [RSSA](#). Explicit data representations.
  - [Machine](#). Untyped register transfer language.
-

## 106 IntroduceLoops

[IntroduceLoops](#) is an optimization pass for the [SSA IntermediateLanguage](#), invoked from [SSASimplify](#).

### 106.1 Description

This pass rewrites any [SSA](#) function that calls itself in tail position into one with a local loop and no self tail calls.

A [SSA](#) function like

```
fun F (arg_0, arg_1) = L_0 ()
 ...
 L_16 (x_0)
 ...
 F (z_0, z_1) Tail
 ...
```

becomes

```
fun F (arg_0', arg_1') = loopS_0 ()
 loopS_0 ()
 loop_0 (arg_0', arg_1')
 loop_0 (arg_0, arg_1)
 L_0 ()
 ...
 L_16 (x_0)
 ...
 loop_0 (z_0, z_1)
 ...
```

### 106.2 Implementation

- [introduce-loops.fun](#)

### 106.3 Details and Notes

## 107 JesperLouisAndersen

Jesper Louis Andersen is an undergraduate student at DIKU, the department of computer science, Copenhagen university. His contributions to MLton are few, though he has made the port of MLton to the NetBSD and OpenBSD platforms.

His general interests in computer science are compiler theory, language theory, algorithms and datastructures and programming. His assets are his general knowledge of UNIX systems, knowledge of system administration, knowledge of operating system kernels; NetBSD in particular.

He was employed by the university as a system administrator for 2 years, which has set him back somewhat in his studies. Currently he is trying to learn mathematics (real analysis, general topology, complex functional analysis and algebra).

### 107.1 Projects using MLton

#### 107.1.1 A register allocator

For internal use at a compiler course at DIKU. It is written in the literate programming style and implements the *Iterated Register Coalescing* algorithm by Lal George and Andrew Appel <http://citeseer.ist.psu.edu/george96iterated.html>. The status of the project is that it is unfinished. Most of the basic parts of the algorithm is done, but the interface to the students (simple) datatype takes some conversion.

#### 107.1.2 A configuration management system in SML

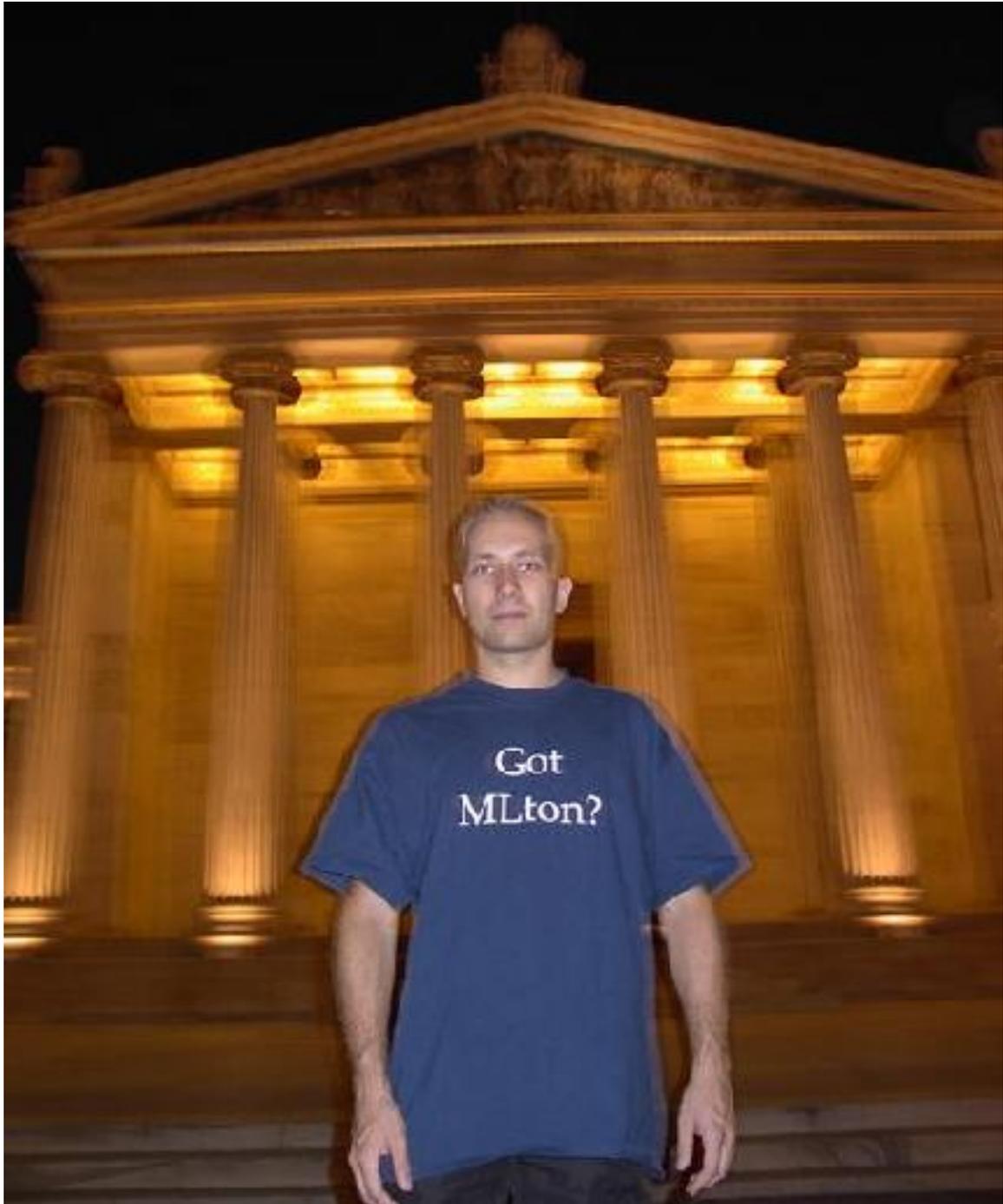
At this time, only loose plans exists for this. The plan is to build a Configuration Management system on the principles of the OpenCM system, see <http://www.opencm.org/docs.html>. The basic idea is to unify "naming" and "identity" into one by uniquely identifying all objects managed in the repository by the use of cryptographic checksums. This mantra guides the rest of the system, providing integrity, accessibility and confidentiality.

---

## 108 JohnnyAndersen

Johnny Andersen (aka Anoq of the Sun)

Here is a picture in front of the academy building at the University of Athens, Greece, taken in September 2003.



## 109 KnownCase

[KnownCase](#) is an optimization pass for the [SSA IntermediateLanguage](#), invoked from [SSASimplify](#).

### 109.1 Description

This pass duplicates and simplifies `Case` transfers when the constructor of the scrutinee is known.

Uses [Restore](#).

For example, the program

```
val rec last =
 fn [] => 0
 | [x] => x
 | _ :: l => last l

val _ = 1 + last [2, 3, 4, 5, 6, 7]
```

gives rise to the [SSA](#) function

```
fun last_0 (x_142) = loopS_1 ()
 loopS_1 ()
 loop_11 (x_142)
 loop_11 (x_143)
 case x_143 of
 nil_1 => L_73 | ::_0 => L_74
 L_73 ()
 return global_5
 L_74 (x_145, x_144)
 case x_145 of
 nil_1 => L_75 | _ => L_76
 L_75 ()
 return x_144
 L_76 ()
 loop_11 (x_145)
```

which is simplified to

```
fun last_0 (x_142) = loopS_1 ()
 loopS_1 ()
 case x_142 of
 nil_1 => L_73 | ::_0 => L_118
 L_73 ()
 return global_5
 L_118 (x_230, x_229)
 L_74 (x_230, x_229, x_142)
 L_74 (x_145, x_144, x_232)
 case x_145 of
 nil_1 => L_75 | ::_0 => L_114
 L_75 ()
 return x_144
 L_114 (x_227, x_226)
 L_74 (x_227, x_226, x_145)
```

### 109.2 Implementation

- `known-case.fun`

### 109.3 Details and Notes

One interesting aspect of [KnownCase](#), is that it often has the effect of unrolling list traversals by one iteration, moving the `nil/: : check` to the end of the loop, rather than the beginning.

---

## 110 LambdaCalculus

The **lambda calculus** is the formal system underlying [Standard ML](#).

---

## 111 LambdaFree

[LambdaFree](#) is an analysis pass for the [SXML IntermediateLanguage](#), invoked from [ClosureConvert](#).

### 111.1 Description

This pass descends the entire [SXML](#) program and attaches a property to each `Lambda PrimExp.t` in the program. Then, you can use `lambdaFree` and `lambdaRec` to get free variables of that `Lambda`.

### 111.2 Implementation

- [lambda-free.sig](#)
- [lambda-free.fun](#)

### 111.3 Details and Notes

For `Lambda`-s bound in a `Fun dec`, `lambdaFree` gives the union of the frees of the entire group of mutually recursive functions. Hence, `lambdaFree` for every `Lambda` in a single `Fun dec` is the same. Furthermore, for a `Lambda` bound in a `Fun dec`, `lambdaRec` gives the list of other functions bound in the same `dec` defining that `Lambda`.

For example:

```
val rec f = fn x => ... y ... g ... f ...
and g = fn z => ... f ... w ...
```

```
lambdaFree(fn x =>) = [y, w]
lambdaFree(fn z =>) = [y, w]
lambdaRec(fn x =>) = [g, f]
lambdaRec(fn z =>) = [f]
```

## 112 LanguageChanges

We are sometimes asked to modify MLton to change the language it compiles. In short, we are very conservative about making such changes. There are a number of reasons for this.

- [The Definition of Standard ML](#) is an extremely high standard of specification. The value of the Definition would be significantly diluted by changes that are not specified at an equally high level, and the dilution increases with the complexity of the language change and its interaction with other language features.
- The SML community is small and there are a number of [SML implementations](#). Without an agreed-upon standard, it becomes very difficult to port programs between compilers, and the community would be balkanized.
- Our main goal is to enable programmers to be as effective as possible with MLton/SML. There are a number of improvements other than language changes that we could spend our time on that would provide more benefit to programmers.
- The more the language that MLton compiles changes over time, the more difficult it is to use MLton as a stable platform for serious program development.

Despite these drawbacks, we have extended SML in a couple of cases.

- [Foreign function interface](#)
- [ML Basis system](#)

We allow these language extensions because they provide functionality that is impossible to achieve without them. The Definition does not define a foreign function interface. So, we must either extend the language or greatly restrict the class of programs that can be written. Similarly, the Definition does not provide a mechanism for namespace control at the module level, making it impossible to deliver packaged libraries and have a hope of users using them without name clashes. The ML Basis system addresses this problem. We have also provided a formal specification of the ML Basis system at the level of the Definition.

### 112.1 Also see

- <http://www.mlton.org/pipermail/mlton/2004-August/016165.html>
- <http://www.mlton.org/pipermail/mlton-user/2004-December/000320.html>

## 113 Lazy

In a lazy (or non-strict) language, the arguments to a function are not evaluated before calling the function. Instead, the arguments are suspended and only evaluated by the function if needed.

**Standard ML** is an eager (or strict) language, not a lazy language. However, it is easy to delay evaluation of an expression in SML by creating a *thunk*, which is a nullary function. In SML, a thunk is written `fn () => e`. Another essential feature of laziness is *memoization*, meaning that once a suspended argument is evaluated, subsequent references look up the value. We can express this in SML with a function that maps a thunk to a memoized thunk.

```
signature LAZY =
 sig
 val lazy: (unit -> 'a) -> unit -> 'a
 end
```

This is easy to implement in SML.

```
structure Lazy: LAZY =
 struct
 fun lazy (th: unit -> 'a): unit -> 'a =
 let
 val r: 'a option ref = ref NONE
 in
 fn () =>
 case !r of
 NONE =>
 let
 val a = th ()
 val () = r := SOME a
 in
 a
 end
 | SOME a => a
 end
 end
```

## 114 Libraries

In theory every strictly conforming Standard ML program should run on MLton. However, often large SML projects use implementation specific features so some "porting" is required. Here is a partial list of software that is known to run on MLton.

- Utility libraries:
  - [SMLNJLibrary](#) - distributed with MLton
  - [MLtonLibraryProject](#) - various libraries located on the MLton subversion repository
  - [mlton](#) - the internal MLton utility library, which we hope to cleanup and make more accessible someday
  - [sml-ext](#), a grab bag of libraries for MLton and other SML implementations (by Sean McLaughlin)
  - [sml-lib](#), a grab bag of libraries for MLton and other SML implementations (by [TomMurphy](#))
- Scanner generators:
  - [MLLPTLibrary](#) - distributed with MLton
  - [MLLex](#) - distributed with MLton
  - [MLULex](#) -
- Parser generators:
  - [MLAntlr](#) -
  - [MLLPTLibrary](#) - distributed with MLton
  - [MLYacc](#) - distributed with MLton
- Concurrency: [ConcurrentML](#) - distributed with MLton
- Graphics
  - [SML3d](#)
  - [mGTK](#)
- Misc. libraries:
  - [CKitLibrary](#) - distributed with MLton
  - [MLRISCLibrary](#) - distributed with MLton
  - [ML-NLFFI](#) - distributed with MLton
  - [Swerve](#), an HTTP server
  - [fxp](#), an XML parser

### 114.1 Ports in progress

[Contact](#) us for details on any of these.

- [MLDoc](http://people.cs.uchicago.edu/%7Ejhr/tools/ml-doc.html) <http://people.cs.uchicago.edu/%7Ejhr/tools/ml-doc.html>
- [Unicode](#)

### 114.2 More

More projects using MLton can be seen on the [Users](#) page.

---

### 114.3 Software for SML implementations other than MLton

- PostgreSQL
  - Moscow ML: <http://www.dina.kvl.dk/%7Esestoft/mosmlib/Postgres.html>
  - SML/NJ NLFFI: <http://smlweb.sourceforge.net/smlsql/>
- Web:
  - ML Kit: [SMLserver](#) (a plugin for AOLserver)
  - Moscow ML: [ML Server Pages](#) (support for PHP-style CGI scripting)
  - SML/NJ: [smlweb](#)

## 115 LibrarySupport

MLton supports both linking to and creating system-level libraries. While Standard ML libraries should be designed with the [MLBasis](#) system to work with other Standard ML programs, system-level library support allows MLton to create libraries for use by other programming languages. Even more importantly, system-level library support allows MLton to access libraries from other languages. This article will explain how to use libraries portably with MLton.

### 115.1 The Basics

A Dynamic Shared Object (DSO) is a piece of executable code written in a format understood by the operating system. Executable programs and dynamic libraries are the two most common examples of a DSO. They are called shared because if they are used more than once, they are only loaded once into main memory. For example, if you start two instances of your web browser (an executable), there may be two processes running, but the program code of the executable is only loaded once. A dynamic library, for example a graphical toolkit, might be used by several different executable programs, each possibly running multiple times. Nevertheless, the dynamic library is only loaded once and its program code is shared between all of the processes.

In addition to program code, DSOs contain a table of textual strings called symbols. These are used in order to make the DSO do something useful, like execute. For example, on linux the symbol `_start` refers to the point in the program code where the operating system should start executing the program. Dynamic libraries generally provide many symbols, corresponding to functions which can be called and variables which can be read or written. Symbols can be used by the DSO itself, or by other DSOs which require services.

When a DSO creates a symbol, this is called *exporting*. If a DSO needs to use a symbol, this is called *importing*. A DSO might need to use symbols defined within itself or perhaps from another DSO. In both cases, it is importing that symbol, but the scope of the import differs. Similarly, a DSO might export a symbol for use only within itself, or it might export a symbol for use by other DSOs. Some symbols are resolved at compile time by the linker (those used within the DSO) and some are resolved at runtime by the dynamic link loader (symbols accessed between DSOs).

### 115.2 Symbols in MLton

Symbols in MLton are both imported and exported via the [ForeignFunctionInterface](#). The notation `_import "symbolname"` imports functions, `_symbol "symbolname"` imports variables, and `_address "symbolname"` imports an address. To create and export a symbol, `_export "symbolname"` creates a function symbol and `_symbol "symbolname" 'alloc'` creates and exports a variable. For details of the syntax and restrictions on the supported FFI types, read the [ForeignFunctionInterface](#) page. In this discussion it only matters that every FFI use is either an import or an export.

When exporting a symbol, MLton supports controlling the export scope. If the symbol should only be used within the same DSO, that symbol has *private* scope. Conversely, if the symbol should also be available to other DSOs the symbol has *public* scope. Generally, one should have as few public exports as possible. Since they are public, other DSOs will come to depend on them, limiting your ability to change them. You specify the export scope in MLton by putting `private` or `public` after the symbol's name in an FFI directive. eg: `_export "foo" private: int->int;` or `_export "bar" public: int->int;`

For technical reasons, the linker and loader on various platforms need to know the scope of a symbol being imported. If the symbol is exported by the same DSO, use `public` or `private` as appropriate. If the symbol is exported by a different DSO, then the scope *external* should be used to import it. Within a DSO, all references to a symbol must use the same scope. MLton will check this at compile time, reporting: `symbol "foo" redeclared as public (previously external)`. This may cause linker errors. However, MLton can only check usage within Standard ML. All objects being linked into a resulting DSO must agree, and it is the programmer's responsibility to ensure this.

Summary of symbol scopes:

- `private`: used for symbols exported within a DSO only for use within that DSO
- `public`: used for symbols exported within a DSO that may also be used outside that DSO
- `external`: used for importing symbols from another DSO
- All uses of a symbol within a DSO (both imports and exports) must agree on the symbol scope

### 115.3 Output Formats

MLton can create executables (`-format executable`) and dynamic shared libraries (`-format library`). To link a shared library, use `-link-opt -l<dso_name>`. The default output format is executable.

MLton can also create archives. An archive is not a DSO, but it does have a collection of symbols. When an archive is linked into a DSO, it is completely absorbed. Other objects being compiled into the DSO should refer to the public symbols in the archive as public, since they are still in the same DSO. However, in the interest of modular programming, private symbols in an archive cannot be used outside of that archive, even within the same DSO.

Although both executables and libraries are DSOs, some implementation details differ on some platforms. For this reason, MLton can create two types of archives. A normal archive (`-format archive`) is appropriate for linking into an executable. Conversely, a libarchive (`-format libarchive`) should be used if it will be linked into a dynamic library.

When MLton does not create an executable, it creates two special symbols. The symbol `libname_open` is a function which must be called before any other symbols are accessed. The `libname` is controlled by the `-libname` compile option and defaults to the name of the output, with any prefixing `lib` stripped (eg: `foo` → `foo`, `libfoo` → `foo`). The symbol `libname_close` is a function which should be called to clean up memory once done.

Summary of `-format` options:

- `executable`: create an executable (a DSO)
- `library`: create a dynamic shared library (a DSO)
- `archive`: create an archive of symbols (not a DSO) that can be linked into an executable
- `libarchive`: create an archive of symbols (not a DSO) that can be linked into a library

Related options:

- `-libname x`: controls the name of the special `_open` and `_close` functions.

### 115.4 Interfacing with C

MLton can generate a C header file. When the output format is not an executable, it creates one by default named `libname.h`. This can be overridden with `-export-header foo.h`. This header file should be included by any C files using the exported Standard ML symbols.

If C is being linked with Standard ML into the same output archive or DSO, then the C code should `#define PART_OF_LIBNAME` before it includes the header file. This ensures that the C code is using the symbols with correct scope. Any symbols exported from C should also be marked using the `PRIVATE/PUBLIC/EXTERNAL` macros defined in the Standard ML export header. The declared C scope on exported C symbols should match the import scope used in Standard ML.

An example:

```
#define PART_OF_FOO
#include "foo.h"

PUBLIC int cFoo() {
 return smlFoo();
}
```

```
val () = _export "smlFoo" private: unit -> int; (fn () => 5)
val cFoo = _import "cFoo" public: unit -> int;
```

## 115.5 Operating-system specific details

On Windows, `libarchive` and `archive` are the same. However, depending on this will lead to portability problems. Windows is also especially sensitive to mixups of `public` and `external`. If an archive is linked, make sure it's symbols are imported as `public`. If a DLL is linked, make sure it's symbols are imported as `external`. Using `external` instead of `public` will result in link errors that `__imp__foo` is undefined. Using `public` instead of `external` will result in inconsistent function pointer addresses and failure to update the imported variables.

On Linux, `libarchive` and `archive` are different. Libarchives are quite rare, but necessary if creating a library from an archive. It is common for a library to provide both an archive and a dynamic library on this platform. The linker will pick one or the other, usually preferring the dynamic library. While a quirk of the operating system allows `external import` to work for both archives and libraries, portable projects should not depend on this behaviour. On other systems it can matter how the library is linked (static or dynamic).

---

## 116 License

### 116.1 Web Site

In order to allow the maximum freedom for the future use of the content in this web site, we require that contributions to the web site be dedicated to the public domain. That means that you can only add works that are already in the public domain, or that you must hold the copyright on the work that you agree to dedicate the work to the public domain.

By contributing to this web site, you agree to dedicate your contribution to the public domain.

### 116.2 Software

As of 20050812, MLton software is licensed under the BSD-style license below. By contributing code to the project, you agree to release the code under this license. Contributors can retain copyright to their contributions by asserting copyright in their code. Contributors may also add to the list of copyright holders in `doc/license/MLton-LICENSE`, which appears below.

This is the license for MLton, a whole-program optimizing compiler for the Standard ML programming language. Send comments and questions to `MLton@mlton.org`.

MLton COPYRIGHT NOTICE, LICENSE AND DISCLAIMER.

Copyright (C) 1999-2013 Henry Cejtin, Matthew Fluet, Suresh Jagannathan, and Stephen Weeks.  
Copyright (C) 1997-2000 by the NEC Research Institute

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both the copyright notice and this permission notice and warranty disclaimer appear in supporting documentation, and that the name of the above copyright holders, or their entities, not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

The above copyright holders disclaim all warranties with regard to this software, including all implied warranties of merchantability and fitness. In no event shall the above copyright holders be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of this software.

## 117 LineDirective

To aid in the debugging of code produced by program generators such as [Noweb](#), MLton supports comments with line directives of the form

```
(*#line l.c "f"*)
```

Here,  $l$  and  $c$  are sequences of decimal digits and  $f$  is the source file. The first character of a source file has the position 1.1. A line directive causes the front end to believe that the character following the right parenthesis is at the line and column of the specified file. A line directive only affects the reporting of error messages and does not affect program semantics (except for functions like `MLton.Exn.history` that report source file positions). Syntactically invalid line directives are ignored. To prevent incompatibilities with SML, the file name may not contain the character sequence `*`).

## 118 LLVM

**LLVM** (Low Level Virtual Machine) is an abstract machine, optimizer, and code generator. It might make a nice backend for MLton, and there has been some discussion about this on the MLton list.

- <http://www.mlton.org/pipermail/mlton/2005-November/028263.html>

The latest is that LLVM's `gcc` variant has been used in place of `gcc`, and so there has been no work toward changing MLton to target LLVM's IL directly.

- <http://www.mlton.org/pipermail/mlton/2006-August/029021.html>

### 118.1 Also see

- [CMinusMinus](#)
-

## 119 LocalFlatten

[LocalFlatten](#) is an optimization pass for the [SSA IntermediateLanguage](#), invoked from [SSASimplify](#).

### 119.1 Description

This pass flattens arguments to [SSA](#) blocks.

A block argument is flattened as long as it only flows to selects and there is some tuple constructed in this function that flows to it.

### 119.2 Implementation

- [local-flatten.fun](#)

### 119.3 Details and Notes

---

## 120 LocalRef

[LocalRef](#) is an optimization pass for the [SSA IntermediateLanguage](#), invoked from [SSASimplify](#).

### 120.1 Description

This pass optimizes `ref` cells local to a [SSA](#) function:

- global `ref`-s only used in one function are moved to the function
- `ref`-s only created, read from, and written to (i.e., don't escape) are converted into function local variables

Uses [Multi](#) and [Restore](#).

### 120.2 Implementation

- `local-ref.fun`

### 120.3 Details and Notes

Moving a global `ref` requires the [Multi](#) analysis, because a global `ref` can only be moved into a function that is executed at most once.

Conversion of non-escaping `ref`-s is structured in three phases:

- analysis—a variable `r = Ref_ref x` escapes if
  - `r` is used in any context besides `Ref_assign (r, _)` or `Ref_deref r`
  - all uses `r` reachable from a (direct or indirect) call to `Thread_copyCurrent` are of the same flavor (either `Ref_assign` or `Ref_deref`); this also requires the [Multi](#) analysis.
- transformation
  - rewrites `r = Ref_ref x` to `r = x`
  - rewrites `_ = Ref_assign (r, y)` to `r = y`
  - rewrites `z = Ref_deref r` to `z = r`

Note that the resulting program violates the SSA condition.

- [Restore](#)—restore the SSA condition.
-

## 121 Logo



### 121.1 Files

- [mlton.svg](#)
  - [mlton-1024.png](#)
  - [mlton-512.png](#)
  - [mlton-256.png](#)
  - [mlton-128.png](#)
  - [mlton-64.png](#)
  - [mlton-32.png](#)
-

## 122 LoopInvariant

[LoopInvariant](#) is an optimization pass for the [SSA IntermediateLanguage](#), invoked from [SSASimplify](#).

### 122.1 Description

This pass removes loop invariant arguments to local loops.

```
loop (x, y)
 ...
 ...
 loop (x, z)
 ...
```

becomes

```
loop' (x, y)
 loop (y)
loop (y)
 ...
 ...
 loop (z)
 ...
```

### 122.2 Implementation

- `loop-invariant.fun`

### 122.3 Details and Notes

## 123 Machine

[Machine](#) is an [IntermediateLanguage](#), translated from [RSSA](#) by [ToMachine](#) and used as input by the [Codegen](#).

### 123.1 Description

[Machine](#) is an [Untyped IntermediateLanguage](#), corresponding to a abstract register machine.

### 123.2 Implementation

- [machine.sig](#)
- [machine.fun](#)

### 123.3 Type Checking

The [Machine IntermediateLanguage](#) has a primitive type checker ([machine.sig](#), [machine.fun](#)), which only checks some liveness properties.

### 123.4 Details and Notes

The runtime structure sets some constants according to the configuration files on the target architecture and OS.

---

## 124 ManualPage

MLton is run from the command line with a collection of options followed by a file name and a list of files to compile, assemble, and link with.

```
mlton [option ...] file.{c|mlb|o|sml} [file.{c|o|s|S} ...]
```

The simplest case is to run `mlton foo.sml`, where `foo.sml` contains a valid SML program, in which case MLton compiles the program to produce an executable `foo`. Since MLton does not support separate compilation, the program must be the entire program you wish to compile. However, the program may refer to signatures and structures defined in the [Basis Library](#).

Larger programs, spanning many files, can be compiled with the [ML Basis system](#). In this case, `mlton foo.mlb` will compile the complete SML program described by the basis `foo.mlb`, which may specify both SML files and additional bases.

### 124.1 Next Steps

- [CompileTimeOptions](#)
- [RunTimeOptions](#)

## 125 MatchCompilation

Match compilation is the process of translating an SML match into a nested tree (or dag) of simple case expressions and tests. MLton's match compiler is described [here](#).

### 125.1 Match compilation in other compilers

- [BaudinetMacQueen85](#)
  - [Leroy90](#), pages 60-69.
  - [Sestoft96](#)
  - [ScottRamsey00](#)
-

## 126 MatchCompile

[MatchCompile](#) is a translation pass, agnostic in the [IntermediateLanguages](#) between which it translates.

### 126.1 Description

[Match compilation](#) converts a case expression with nested patterns into a case expression with flat patterns.

### 126.2 Implementation

- [match-compile.sig](#)
- [match-compile.fun](#)

### 126.3 Details and Notes

```
val matchCompile:
 {caseType: Type.t, (* type of entire expression *)
 cases: (NestedPat.t * ((Var.t -> Var.t) -> Exp.t)) vector,
 conTycon: Con.t -> Tycon.t,
 region: Region.t,
 test: Var.t,
 testType: Type.t,
 tyconCons: Tycon.t -> {con: Con.t, hasArg: bool} vector}
-> Exp.t * (unit -> ((Layout.t * {isOnlyExns: bool}) vector) vector)
```

`matchCompile` is complicated by the desire for modularity between the match compiler and its caller. Its caller is responsible for building the right hand side of a rule  $p \Rightarrow e$ . On the other hand, the match compiler is responsible for destructing the test and binding new variables to the components. In order to connect the new variables created by the match compiler with the variables in the pattern  $p$ , the match compiler passes an environment back to its caller that maps each variable in  $p$  to the corresponding variable introduced by the match compiler.

The match compiler builds a tree of n-way case expressions by working from outside to inside and left to right in the patterns. For example,

```
case x of
 (_, C1 a) => e1
| (C2 b, C3 c) => e2
```

is translated to

```
let
 fun f1 a = e1
 fun f2 (b, c) = e2
in
 case x of
 (x1, x2) =>
 (case x1 of
 C2 b' => (case x2 of
 C1 a' => f1 a'
 | C3 c' => f2 (b', c')
 | _ => raise Match)
 | _ => (case x2 of
 C1 a_ => f1 a_
 | _ => raise Match))
end
```

Here you can see the necessity of abstracting out the right hand sides of the cases in order to avoid code duplication. Right hand sides are always abstracted. The simplifier cleans things up. You can also see the new (primed) variables introduced by the match compiler and how the renaming works. Finally, you can see how the match compiler introduces the necessary default clauses in order to make a match exhaustive, i.e. cover all the cases.

The match compiler uses `numCons` and `tyconCons` to determine the exhaustivity of matches against constructors.

---

## 127 MatthewFluet

Matthew Fluet ( [matthew.fluet@gmail.com](mailto:matthew.fluet@gmail.com) , <http://www.cs.rit.edu/~7Etmf> ) is an Assistant Professor at the [Rochester Institute of Technology](#).

---

Current MLton projects:

- general maintenance
  - release new version
- 

Misc. and underspecified TODOs:

- understand [RefFlatten](#) and [DeepFlatten](#)
    - <http://www.mlton.org/pipermail/mlton/2005-April/026990.html>
    - <http://www.mlton.org/pipermail/mlton/2007-November/030056.html>
    - <http://www.mlton.org/pipermail/mlton/2008-April/030250.html>
    - <http://www.mlton.org/pipermail/mlton/2008-July/030279.html>
    - <http://www.mlton.org/pipermail/mlton/2008-August/030312.html>
    - <http://www.mlton.org/pipermail/mlton/2008-September/030360.html>
    - <http://www.mlton.org/pipermail/mlton-user/2009-June/001542.html>
  - MSG\_DONTWAIT isn't Posix
  - coordinate w/ Dan Spoonhower and Lukasz Ziarek and Armand Navabi on multi-threaded
    - <http://www.mlton.org/pipermail/mlton/2008-March/030214.html>
  - Intel Research bug: no `tyconRep` property (company won't release sample code)
    - <http://www.mlton.org/pipermail/mlton-user/2008-March/001358.html>
  - treatment of real constants
    - <http://www.mlton.org/pipermail/mlton/2008-May/030262.html>
    - <http://www.mlton.org/pipermail/mlton/2008-June/030271.html>
  - representation of `bool` and `_bool` in [ForeignFunctionInterface](#)
    - <http://www.mlton.org/pipermail/mlton/2008-May/030264.html>
  - <http://www.icfpcontest.org>
    - John Reppy claims that "It looks like the card-marking overhead that one incurs when using generational collection swamps the benefits of generational collection."
  - page to disk policy / single heap
    - <http://www.mlton.org/pipermail/mlton/2008-June/030278.html>
    - <http://www.mlton.org/pipermail/mlton/2008-August/030318.html>
  - `MLton.GC.pack` doesn't keep a small heap if a garbage collection occurs before `MLton.GC.unpack`.
-

- 
- It might be preferable for `MLton.GC.pack` to be implemented as a (new) `MLton.GC.Ratios.setLive 1.1` followed by `MLton.GC.collect ()` and for `MLton.GC.unpack` to be implemented as `MLton.GC.Ratios.setLive 8.0` followed by `MLton.GC.collect ()`.
  - The static struct `GC_objectType objectTypes[] = array` includes many duplicates. Objects of distinct source type, but equivalent representations (in terms of size, bytes non-pointers, number pointers) can share the `objectType` index.
  - PolySpace bug: [Redundant](http://www.mlton.org/pipermail/mlton/2008-September/030355.html) optimization (company won't release sample code)
    - <http://www.mlton.org/pipermail/mlton/2008-September/030355.html>
  - treatment of exception raised during [BasisLibrary](#) evaluation
    - <http://www.mlton.org/pipermail/mlton/2008-December/030501.html>
    - <http://www.mlton.org/pipermail/mlton/2008-December/030502.html>
    - <http://www.mlton.org/pipermail/mlton/2008-December/030503.html>
  - Use `memcpy`
    - <http://www.mlton.org/pipermail/mlton-user/2009-January/001506.html>
    - <http://www.mlton.org/pipermail/mlton/2009-January/030506.html>
  - Implement more 64bit primops in x86 codegen
    - <http://www.mlton.org/pipermail/mlton/2009-January/030507.html>
  - Enrich path-map file syntax:
    - <http://www.mlton.org/pipermail/mlton/2008-September/030348.html>
    - <http://www.mlton.org/pipermail/mlton-user/2009-January/001507.html>
  - PolySpace bug: crash during Cheney-copy collection
    - <http://www.mlton.org/pipermail/mlton/2009-February/030513.html>
  - eliminate `-build-constants`
    - all `_const-s` are known by `runtime/gen/basis-ffi.def`
    - generate `gen-constants.c` from `basis-ffi.def`
    - generate constants from `gen-constants.c` and `libmlton.a`
    - similar to `gen-sizes.c` and `sizes`
  - eliminate "Windows hacks" for Cygwin from `Path` module
    - <http://www.mlton.org/pipermail/mlton/2009-July/030606.html>
  - extend IL type checkers to check for empty property lists
  - make (unsafe) `IntInf` conversions into primitives
    - <http://www.mlton.org/pipermail/mlton/2009-July/030622.html>
-

## 128 mGTK

**mGTK** is a wrapper for **GTK+**, a GUI toolkit.

We recommend using mGTK 0.93, which is not listed on their home page, but is available at the [file release page](#). To test it, after unpacking, do `cd examples; make mlton`, after which you should be able to run the many examples (`signup-mlton`, `listview-mlton`, ...).

### 128.1 Also see

- [Glade](#)
-

## 129 MichaelNorrish

I am a researcher at [NICTA](#), with a web-page [here](#).

I'm interested in MLton because of the chance that it might be a good vehicle for future implementations of the [HOL](#) theorem-proving system. It's beginning to look as if one route forward will be to embed an SML interpreter into a MLton-compiled executable. I don't know if an extensible interpreter of the kind we're looking for already exists.

---

## 130 MikeThomas

Here is a picture at home in Brisbane, Queensland, Australia, taken in January 2004.



## 131 ML

ML stands for *meta language*. ML was originally designed in the 1970s as a programming language to assist theorem proving in the logic LCF. In the 1980s, ML split into two variants, [Standard ML](#) and [OCaml](#), both of which are still used today.

---

## 132 MLAntlr

**MLAntlr** is a parser generator for [Standard ML](#).

### 132.1 Also see

- [MLULex](#)
  - [MLLPTLibrary](#)
-

## 133 MLBasis

The ML Basis system extends [Standard ML](#) to support programming-in-the-very-large, namespace management at the module level, separate delivery of library sources, and more. While Standard ML modules are a sophisticated language for programming-in-the-large, it is difficult, if not impossible, to accomplish a number of routine namespace management operations when a program draws upon multiple libraries provided by different vendors.

The ML Basis system is a simple, yet powerful, approach that builds upon the programmer's intuitive notion (and [The Definition of Standard ML \(Revised\)](#)'s formal notion) of the top-level environment (a *basis*). The system is designed as a natural extension of [Standard ML](#); the formal specification of the ML Basis system ([mlb-formal.pdf](#)) is given in the style of the Definition.

Here are some of the key features of the ML Basis system:

1. **Explicit file order:** The order of files (and, hence, the order of evaluation) in the program is explicit. The ML Basis system's semantics are structured in such a way that for any well-formed project, there will be exactly one possible interpretation of the project's syntax, static semantics, and dynamic semantics.
2. **Implicit dependencies:** A source file (corresponding to an SML top-level declaration) is elaborated in the environment described by preceding declarations. It is not necessary to explicitly list the dependencies of a file.
3. **Scoping and renaming:** The ML Basis system provides mechanisms for limiting the scope of (i.e, hiding) and renaming identifiers.
4. **No naming convention for finding the file that defines a module.** To import a module, its defining file must appear in some ML Basis file.

### 133.1 Next steps

- [MLBasisSyntaxAndSemantics](#)
  - [MLBasisExamples](#)
  - [MLBasisPathMap](#)
  - [MLBasisAnnotations](#)
  - [MLBasisAvailableLibraries](#)
-

## 134 MLBasisAnnotationExamples

Here are some example uses of [MLBasisAnnotations](#).

### 134.1 Eliminate spurious warnings in automatically generated code

Programs that automatically generate source code can often produce nonexhaustive matches, relying on invariants of the generated code to ensure that the matches never fail. A programmer may wish to elide the nonexhaustive match warnings from this code, in order that legitimate warnings are not missed in a flurry of false positives. To do so, the programmer simply annotates the generated code with the `nonexhaustiveMatch ignore` annotation:

```
local
 $(GEN_ROOT)/gen-lib.mlb

 ann "nonexhaustiveMatch ignore" in
 foo.gen.sml
 end
in
 signature FOO
 structure Foo
end
```

### 134.2 Deliver a library

Standard ML libraries can be delivered via `.mlb` files. Authors of such libraries should strive to be mindful of the ways in which programmers may choose to compile their programs. For example, although the defaults for `sequenceNonUnit` and `warnUnused` are `ignore` and `false`, periodically compiling with these annotations defaulted to `warn` and `true` can help uncover likely bugs. However, a programmer is unlikely to be interested in unused modules from an imported library, and the behavior of `sequenceNonUnit error` may be incompatible with some libraries. Hence, a library author may choose to deliver a library as follows:

```
ann
 "nonexhaustiveMatch warn" "redundantMatch warn"
 "sequenceNonUnit warn"
 "warnUnused true" "forceUsed"
in
 local
 file1.sml
 ...
 filen.sml
 in
 functor F1
 ...
 signature S1
 ...
 structure SN
 ...
 end
 end
```

The annotations `nonexhaustiveMatch warn`, `redundantMatch warn`, and `sequenceNonUnit warn` have the obvious effect on elaboration. The annotations `warnUnused true` and `forceUsed` work in conjunction — warning on any identifiers that do not contribute to the exported modules, and preventing warnings on exported modules that are not used in the remainder of the program. Many of the [available libraries](#) are delivered with these annotations.

## 135 MLBasisAnnotations

[ML Basis](#) annotations control options that affect the elaboration of SML source files. Conceptually, a basis file is elaborated in a default annotation environment (just as it is elaborated in an empty basis). The declaration `ann "ann" in basdec end` merges the annotation `ann` with the "current" annotation environment for the elaboration of `basdec`. To allow for future expansion, `"ann"` is lexed as a single SML string constant. To conveniently specify multiple annotations, the following derived form is provided:

```
ann "ann" ("ann")+ in basdec end ⇒ ann "ann" in ann ("ann")+ in basdec end end
```

Here are the available annotations. In the explanation below, for annotations that take an argument, the first value listed is the default.

- `allowFFI {false|true}`  
If `true`, allow `_address`, `_export`, `_import`, and `_symbol` expressions to appear in source files. See [ForeignFunctionInterface](#).
- `forceUsed`  
Force all identifiers in the basis denoted by the body of the `ann` to be considered used; use in conjunction with `warnUnused true`.
- `nonexhaustiveExnMatch {default|ignore}`  
If `ignore`, suppress errors and warnings about nonexhaustive matches that arise solely from unmatched exceptions. If `default`, follow the behavior of `nonexhaustiveMatch`.
- `nonexhaustiveMatch {warn|error|ignore}`  
If `error` or `warn`, report nonexhaustive matches. An error will abort a compile, while a warning will not.
- `redundantMatch {warn|error|ignore}`  
If `error` or `warn`, report redundant matches. An error will abort a compile, while a warning will not.
- `resolveScope {strdec|dec|topdec|program}`  
Used to control the scope at which overload constraints are resolved to default types (if not otherwise resolved by type inference) and the scope at which unresolved flexible record constraints are reported.  
The syntactic-class argument means to perform resolution checks at the smallest enclosing syntactic form of the given class. The default behavior is to resolve at the smallest enclosing `strdec` (which is equivalent to the largest enclosing `dec`). Other useful behaviors are to resolve at the smallest enclosing `topdec` (which is equivalent to the largest enclosing `strdec`) and at the smallest enclosing `program` (which corresponds to a single `.sml` file and does not correspond to the whole `.mlb` program).
- `sequenceNonUnit {ignore|error|warn}`  
If `error` or `warn`, report when `e1` is not of type `unit` in the sequence expression `(e1; e2)`. This can be helpful in detecting carried applications that are mistakenly not fully applied. To silence spurious messages, you can use `ignore e1`.
- `warnUnused {false|true}`  
Report unused identifiers.

### 135.1 Next Steps

- [MLBasisAnnotationExamples](#)
- [WarnUnusedAnomalies](#)

## 136 MLBasisAvailableLibraries

MLton comes with the following [ML Basis](#) files available.

- `$(SML_LIB)/basis/basis.mlb`  
The [Basis Library](#).
- `$(SML_LIB)/basis/basis-1997.mlb`  
The (deprecated) 1997 version of the [Basis Library](#).
- `$(SML_LIB)/basis/mlton.mlb`  
The [MLton](#) structure and signatures.
- `$(SML_LIB)/basis/c-types.mlb`  
Various structure aliases useful as [ForeignFunctionInterfaceTypes](#).
- `$(SML_LIB)/basis/unsafe.mlb`  
The [Unsafe](#) structure and signature.
- `$(SML_LIB)/basis/sml-nj.mlb`  
The [SMLofNJ](#) structure and signature.
- `$(SML_LIB)/mlyacc-lib/mlyacc-lib.mlb`  
Modules used by parsers built with [MLYacc](#).
- `$(SML_LIB)/cml/cml.mlb`  
[ConcurrentML](#), a library for message-passing concurrency.
- `$(SML_LIB)/mlnlffi-lib/mlnlffi-lib.mlb`  
[ML-NLFFI](#), a library for foreign function interfaces.
- `$(SML_LIB)/mlrisc-lib/...`  
[MLRISCLibrary](#), a library for retargetable and optimizing compiler back ends.
- `$(SML_LIB)/smlnj-lib/...`  
[SMLNJLibrary](#), a collection of libraries distributed with SML/NJ.
- `$(SML_LIB)/ckit-lib/ckit-lib.mlb`  
[CKitLibrary](#), a library for C source code.
- `$(SML_LIB)/mllpt-lib/mllpt-lib.mlb`  
[MLLPTLibrary](#), a support library for the [MLULex](#) scanner generator and the [MLAntlr](#) parser generator.

### 136.1 Basis fragments

There are a number of specialized ML Basis files for importing fragments of the [Basis Library](#) that can not be expressed within SML.

- `$(SML_LIB)/basis/pervasive-types.mlb`  
The top-level types and constructors of the Basis Library.
- `$(SML_LIB)/basis/pervasive-exns.mlb`  
The top-level exception constructors of the Basis Library.

- `$(SML_LIB)/basis/pervasive-vals.mlb`  
The top-level values of the Basis Library, without infix status.
  - `$(SML_LIB)/basis/overloads.mlb`  
The top-level overloaded values of the Basis Library, without infix status.
  - `$(SML_LIB)/basis/equal.mlb`  
The polymorphic equality = and inequality <> values, without infix status.
  - `$(SML_LIB)/basis/infixes.mlb`  
The infix declarations of the Basis Library.
  - `$(SML_LIB)/basis/pervasive.mlb`  
The entire top-level value and type environment of the Basis Library, with infix status. This is the same as importing the above six MLB files.
-

## 137 MLBasisExamples

Here are some example uses of [ML Basis](#) files.

### 137.1 Complete program

Suppose your complete program consists of the files `file1.sml`, ..., `filen.sml`, which depend upon libraries `lib1.mlb`, ..., `libm.mlb`.

```
(* import libraries *)
lib1.mlb
...
libm.mlb

(* program files *)
file1.sml
...
filen.sml
```

The bases denoted by `lib1.mlb`, ..., `libm.mlb` are merged (bindings of names in later bases take precedence over bindings of the same name in earlier bases), producing a basis in which `file1.sml`, ..., `filen.sml` are elaborated, adding additional bindings to the basis.

### 137.2 Export filter

Suppose you only want to export certain structures, signatures, and functors from a collection of files.

```
local
 file1.sml
 ...
 filen.sml
in
 (* export filter here *)
 functor F
 structure S
end
```

While `file1.sml`, ..., `filen.sml` may declare top-level identifiers in addition to `F` and `S`, such names are not accessible to programs and libraries that import this `.mlb`.

### 137.3 Export filter with renaming

Suppose you want an export filter, but want to rename one of the modules.

```
local
 file1.sml
 ...
 filen.sml
in
 (* export filter, with renaming, here *)
 functor F
 structure S' = S
end
```

Note that functor `F` is an abbreviation for functor `F = F`, which simply exports an identifier under the same name.

### 137.4 Import filter

Suppose you only want to import a functor  $F$  from one library and a structure  $S$  from another library.

```
local
 lib1.mlb
in
 (* import filter here *)
 functor F
end
local
 lib2.mlb
in
 (* import filter here *)
 structure S
end
file1.sml
...
filen.sml
```

### 137.5 Import filter with renaming

Suppose you want to import a structure  $S$  from one library and another structure  $S$  from another library.

```
local
 lib1.mlb
in
 (* import filter, with renaming, here *)
 structure S1 = S
end
local
 lib2.mlb
in
 (* import filter, with renaming, here *)
 structure S2 = S
end
file1.sml
...
filen.sml
```

### 137.6 Full Basis

Since the Modules level of SML is the natural means for organizing program and library components, MLB files provide convenient syntax for renaming Modules level identifiers (in fact, renaming of functor identifiers provides a mechanism that is not available in SML). However, please note that .mlb files elaborate to full bases including top-level types and values (including infix status), in addition to structures, signatures, and functors. For example, suppose you wished to extend the [Basis Library](#) with an  $(\text{'a}, \text{'b})$  `either` datatype corresponding to a disjoint sum; the type and some operations should be available at the top-level; additionally, a signature and structure provide the complete interface.

We could use the following files.

`either-sigs.sml`

```
signature EITHER_GLOBAL =
sig
 datatype ('a, 'b) either = Left of 'a | Right of 'b
 val & : ('a -> 'c) * ('b -> 'c) -> ('a, 'b) either -> 'c
 val && : ('a -> 'c) * ('b -> 'd) -> ('a, 'b) either -> ('c, 'd) either
end
```

```
signature EITHER =
 sig
 include EITHER_GLOBAL
 val isLeft : ('a, 'b) either -> bool
 val isRight : ('a, 'b) either -> bool
 ...
 end
```

either-strs.sml

```
structure Either : EITHER =
 struct
 datatype ('a, 'b) either = Left of 'a | Right of 'b
 fun f & g = fn x =>
 case x of Left z => f z | Right z => g z
 fun f && g = (Left o f) & (Right o g)
 fun isLeft x = ((fn _ => true) & (fn _ => false)) x
 fun isRight x = (not o isLeft) x
 ...
 end
structure EitherGlobal : EITHER_GLOBAL = Either
```

either-infixes.sml

```
infixr 3 & &&
```

either-open.sml

```
open EitherGlobal
```

either.mlb

```
either-infixes.sml
local
 (* import Basis Library *)
 $(SML_LIB)/basis/basis.mlb
 either-sigs.sml
 either-strs.sml
in
 signature EITHER
 structure Either
 either-open.sml
end
```

A client that imports `either.mlb` will have access to neither `EITHER_GLOBAL` nor `EitherGlobal`, but will have access to the type `either` and the values `&` and `&&` (with infix status) in the top-level environment. Note that `either-infixes.sml` is outside the scope of the local, because we want the infixes available in the implementation of the library and to clients of the library.

## 138 MLBasisPathMap

An [ML Basis path map](#) describes a map from ML Basis path variables (of the form  $\$(VAR)$ ) to file system paths. ML Basis path variables provide a flexible way to refer to libraries while allowing them to be moved without changing their clients.

The format of an `mlb-path-map` file is a sequence of lines; each line consists of two, white-space delimited tokens. The first token is a path variable `VAR` and the second token is the path to which the variable is mapped. The path may include path variables, which are recursively expanded.

The mapping from path variables to paths is initialized by reading a system-wide configuration file: `/usr/lib/mlton/mlb-path-map`. Additional path maps can be specified with `-mlb-path-map` and individual path variable mappings can be specified with `-mlb-path-var` (see [CompileTimeOptions](#)). Configuration files are processed from first to last and from top to bottom, later mappings take precedence over earlier mappings.

The compiler and system-wide configuration file makes the following path variables available.

| MLB path variable         | Description                                                            |
|---------------------------|------------------------------------------------------------------------|
| <code>SML_LIB</code>      | path to system-wide libraries, usually <code>/usr/lib/mlton/sml</code> |
| <code>TARGET_ARCH</code>  | string representation of target architecture                           |
| <code>TARGET_OS</code>    | string representation of target operating system                       |
| <code>DEFAULT_INT</code>  | binding for default int, usually <code>int32</code>                    |
| <code>DEFAULT_WORD</code> | binding for default word, usually <code>word32</code>                  |
| <code>DEFAULT_REAL</code> | binding for default real, usually <code>real64</code>                  |

## 139 MLBasisSyntaxAndSemantics

An [ML Basis](#) (MLB) file should have the `.mlb` suffix and should contain a basis declaration.

### 139.1 Syntax

A basis declaration (*basdec*) must be one of the following forms.

- `basis basid = basexp` (and `basid = basexp`)\*
- `open basid1 ... basidn`
- `local basdec in basdec end`
- `basdec [;] basdec`
- `structure strid [= strid]` (and `strid [= strid]`)\*
- `signature sigid [= sigid]` (and `sigid [= sigid]`)\*
- `functor funid [= funid]` (and `funid [= funid]`)\*
- `path.sml`, `path.sig`, or `path.fun`
- `path.mlb`
- `ann "ann" in basdec end`

A basis expression (*basexp*) must be of one the following forms.

- `bas basdec end`
- `basid`
- `let basdec in basexp end`

Nested SML-style comments (enclosed with `(* and *)`) are ignored (but [LineDirectives](#) are recognized).

Paths can be relative or absolute. Relative paths are relative to the directory containing the MLB file. Paths may include path variables and are expanded according to a [path map](#). Unquoted paths may include alpha-numeric characters and the symbols `"-`" and `"_"`, along with the arc separator `"/"` and extension separator `"."`. More complicated paths, including paths with spaces, may be included by quoting the path with `"`. A quoted path is lexed as an SML string constant.

[Annotations](#) allow a library author to control options that affect the elaboration of SML source files.

### 139.2 Semantics

There is a [formal semantics](#) for ML Basis files in the style of the [Definition](#). Here, we give an informal explanation.

An SML structure is a collection of types, values, and other structures. Similarly, a basis is a collection, but of more kinds of objects: types, values, structures, fixities, signatures, functors, and other bases.

A basis declaration denotes a basis. A structure, signature, or functor declaration denotes a basis containing the corresponding module. Sequencing of basis declarations merges bases, with later definitions taking precedence over earlier ones, just like sequencing of SML declarations. Local declarations provide name hiding, just like SML local declarations. A reference to an SML source file causes the file to be elaborated in the basis extant at the point of reference. A reference to an MLB file causes the basis denoted by that MLB file to be imported — the basis at the point of reference does *not* affect the imported basis.

Basis expressions and basis identifiers allow binding a basis to a name.

An MLB file is elaborated starting in an empty basis. Each MLB file is elaborated and evaluated only once, with the result being cached. Subsequent references use the cached value. Thus, any observable effects due to evaluation are not duplicated if the MLB file is referred to multiple times.

## 140 MLj

MLj is a [Standard ML implementation](#) that targets Java bytecode. It is no longer maintained. It has morphed into [SML.NET](#).

### 140.1 Also see

- [BentonEtAl98](#)
  - [BentonKennedy99](#)
-

## 141 MLKit

The **ML Kit** is a **Standard ML implementation**.

MLKit supports:

- **SML'97**
  - including most of the latest **Basis Library specification**,
- **ML Basis files**
  - and separate compilation,
- **Region-Based Memory Management**
  - and **garbage collection**,
- Multiple backends, including
  - native x86,
  - bytecode, and
  - JavaScript (see **SMLtoJs**).

At the time of writing, MLKit does not support:

- concurrent programming / threads,
  - calling from C to SML.
-

## 142 MLLex

[MLLex](#) is a lexical analyzer generator for [Standard ML](#) modeled after the Lex lexical analyzer generator.

A version of MLLex, ported from the [SML/NJ](#) sources, is distributed with MLton.

### 142.1 Description

MLLex takes as input the lex language as defined in the ML-Lex manual, and outputs a lexical analyzer in SML.

### 142.2 Implementation

- [lexgen.sml](#)
- [main.sml](#)
- [call-main.sml](#)

### 142.3 Details and Notes

There are 3 main passes in the MLLex tool:

- Source parsing. In this pass, lex source program are parsed into internal representations. The core part of this pass is a hand-written lexer and an LL(1) parser. The output of this pass is a record of user code, rules (along with start states) and actions. (MLLex definitions are wiped off.)
- DFA construction. In this pass, a DFA is constructed by the algorithm of H. Yamada et. al.
- Output. In this pass, the generated DFA is written out as a transition table, along with a table-driven algorithm, to an SML file.

### 142.4 Also see

- [mllex.pdf](#)
  - [MLYacc](#)
  - [AppelEtAl94](#)
  - [Price09](#)
-

## 143 MLLPTLibrary

The **ML-LPT Library** is a support library for the **MLULex** scanner generator and the **MLAntlr** parser generator. The ML-LPT Library is distributed with SML/NJ.

As of 20130706, MLton includes the ML-LPT Library synchronized with SML/NJ version 110.76.

### 143.1 Usage

- You can import the ML-LPT Library into an MLB file with:

| MLB file                                         | Description |
|--------------------------------------------------|-------------|
| <code>\$(SML_LIB)/mllpt-lib/mllpt-lib.mlb</code> |             |

- If you are porting a project from SML/NJ's **CompilationManager** to MLton's **ML Basis system** using `cm2mlb`, note that the following map is included by default:

```
MLLPT Library
$m1-lpt-lib.cm $(SML_LIB)/mllpt-lib
$m1-lpt-lib.cm/ml-lpt-lib.cm $(SML_LIB)/mllpt-lib/mllpt-lib.mlb
```

This will automatically convert a `$/mllpt-lib.cm` import in an input `.cm` file into a `$(SML_LIB)/mllpt-lib/mllpt-lib.mlb` import in the output `.mlb` file.

### 143.2 Details

### 143.3 Patch

- `m1-lpt.patch`

## 144 MLmon

An `mlmon.out` file records dynamic [profiling](#) counts.

### 144.1 File format

An `mlmon.out` file is a text file with a sequence of lines.

- The string "MLton prof".
  - The string "alloc", "count", or "time", depending on the kind of profiling information, corresponding to the command-line argument supplied to `mlton -profile`.
  - The string "current" or "stack" depending on whether profiling data was gathered for only the current function (the top of the stack) or for all functions on the stack. This corresponds to whether the executable was compiled with `-profile-stack false` or `-profile-stack true`.
  - The magic number of the executable.
  - The number of non-gc ticks, followed by a space, then the number of GC ticks.
  - The number of (split) functions for which data is recorded.
  - A line for each (split) function with counts. Each line contains an integer count of the number of ticks while the function was current. In addition, if stack data was gathered (`-profile-stack true`), then the line contains two additional tick counts:
    - the number of ticks while the function was on the stack.
    - the number of ticks while the function was on the stack and a GC was performed.
  - The number of (master) functions for which data is recorded.
  - A line for each (master) function with counts. The lines have the same format and meaning as with split-function counts.
-

## 145 MLNLFFI

[ML-NLFFI](#) is the no-longer-foreign-function interface library for SML.

As of 20050212, MLton has an initial port of ML-NLFFI from SML/NJ to MLton. All of the ML-NLFFI functionality is present.

Additionally, MLton has an initial port of the [mlnlffigen](#) tool from SML/NJ to MLton. Due to low-level details, the code generated by SML/NJ's `ml-nlffigen` is not compatible with MLton, and vice-versa. However, the generated code has the same interface, so portable client code can be written. MLton's `mlnlffigen` does not currently support C functions with `struct` or `union` arguments.

### 145.1 Usage

- You can import the ML-NLFFI Library into an MLB file with

| MLB file                                             | Description |
|------------------------------------------------------|-------------|
| <code>\$(SML_LIB)/mlnlffi-lib/mlnlffi-lib.mlb</code> |             |

- If you are porting a project from SML/NJ's [CompilationManager](#) to MLton's [ML Basis system](#) using `cm2mlb`, note that the following maps are included by default:

```
MLNLFFI Library
$c $(SML_LIB)/mlnlffi-lib
$c/c.cm $(SML_LIB)/mlnlffi-lib/mlnlffi-lib.mlb
```

This will automatically convert a `$/c.cm` import in an input `.cm` file into a `$(SML_LIB)/mlnlffi-lib/mlnlffi-lib.mlb` import in the output `.mlb` file.

### 145.2 Also see

- [Blume01](#)
- [MLNLFFIImplementation](#)
- [MLNLFFIGen](#)

## 146 MLNLFFIGen

`mlnlffigen` generates a [MLNLFFI](#) binding from a collection of `.c` files. It is based on the [CKitLibrary](#), which is primarily designed to handle standardized C and thus does not understand many (any?) compiler extensions; however, it attempts to recover from errors when seeing unrecognized definitions.

In order to work around common `gcc` extensions, it may be useful to add `-cppopt` options to the command line; for example `-cppopt '-D__extension__'` may be occasionally useful. Fortunately, most portable libraries largely avoid the use of these types of extensions in header files.

`mlnlffigen` will normally not generate bindings for `#included` files; see `-match` and `-allSU` if this is desirable.

---

## 147 MLNLFFI Implementation

MLton's implementation(s) of the [MLNLFFI](#) library differs from the SML/NJ implementation in two important ways:

- MLton cannot utilize the `Unsafe.cast` "cheat" described in Section 3.7 of [Blume01](#). (MLton's representation of [closures](#) and [aggressive representation](#) optimizations make an `Unsafe.cast` even more "unsafe" than in other implementations.)

We have considered two solutions:

- One solution is to utilize an additional type parameter (as described in Section 3.7 of [Blume01](#)):

```
signature C = sig
 type ('t, 'f, 'c) obj
 eqtype ('t, 'f, 'c) obj'
 ...
 type ('o, 'f) ptr
 eqtype ('o, 'f) ptr'
 ...
 type 'f fptr
 type 'f ptr'
 ...
 structure T : sig
 type ('t, 'f) typ
 ...
 end
end
```

The rule for `('t, 'f, 'c) obj`, `('t, 'f, 'c) ptr`, and also `('t, 'f) T.typ` is that whenever `F fptr` occurs within the instantiation of `'t`, then `'f` must be instantiated to `F`. In all other cases, `'f` will be instantiated to `unit`.

(In the actual MLton implementation, an abstract type `na f` (not-a-function) is used instead of `unit`.)

While this means that type-annotated programs may not type-check under both the SML/NJ implementation and the MLton implementation, this should not be a problem in practice. Tools, like `ml-nlffigen`, which are necessarily implementation dependent (in order to make [calls through a C function pointer](#)), may be easily extended to emit the additional type parameter. Client code which uses such generated glue-code (e.g., Section 1 of [Blume01](#)) need rarely write type-annotations, thanks to the magic of type inference.

- The above implementation suffers from two disadvantages.

First, it changes the MLNLFFI Library interface, meaning that the same program may not type-check under both the SML/NJ implementation and the MLton implementation (though, in light of type inference and the richer MLRep structure provided by MLton, this point is mostly moot).

Second, it appears to unnecessarily duplicate type information. For example, an external `C` variable of type `int (* f[3]) (int)` (that is, an array of three function pointers), would be represented by the SML type `((sint -> sint) fptr, dec dg3) arr, sint -> sint, rw) obj`. One might well ask why the `'f` instantiation `(sint -> sint` in this case) cannot be *extracted* from the `'t` instantiation `((sint -> sint) fptr, dec dg3) arr` in this case), obviating the need for a separate *function-type* type argument. There are a number of components to an complete answer to this question. Foremost is the fact that [Standard ML](#) supports neither (general) type-level functions nor intensional polymorphism.

A more direct answer for MLNLFFI is that in the SML/NJ implementation, the definition of the types `('t, 'c) obj` and `('t, 'c) ptr` are made in such a way that the type variables `'t` and `'c` are [phantom](#) (not contributing to the run-time representation of an `('t, 'c) obj` or `('t, 'c) ptr` value), despite the fact that the types `((sint -> sint) fptr, rw) ptr` and `((double -> double) fptr, rw) ptr` necessarily carry distinct (and type incompatible) run-time (C-)type information (RTTI), corresponding to the different calling conventions of the two `C` functions. The `Unsafe.cast` "cheat" overcomes the type incompatibility without introducing a new type variable (as in the first solution above).

Hence, the reason that *function-type* type cannot be extracted from the `'t` type variable instantiation is that the type of the representation of RTTI doesn't even *see* the (phantom) `'t` type variable. The solution which presents itself is to give up on the phantomness of the `'t` type variable, making it available to the representation of RTTI.

This is not without some small drawbacks. Because many of the types used to instantiate `'t` carry more structure than is strictly necessary for `'t`'s RTTI, it is sometimes necessary to wrap and unwrap RTTI to accommodate the additional structure. (In the other implementations, the corresponding operations can pass along the RTTI unchanged.) However, these coercions contribute minuscule overhead; in fact, in a majority of cases, MLton's optimizations will completely eliminate the RTTI from the final program.

The implementation distributed with MLton uses the second solution.

Bonus question: Why can't one use a [universal type](#) to eliminate the use of `Unsafe.cast`?

– Answer: ???

- MLton (in both of the above implementations) provides a richer MLRep structure, utilizing `Int<N>` and `Word<N>` structures.

```
structure MLRep = struct
 structure Char =
 struct
 structure Signed = Int8
 structure Unsigned = Word8
 (* word-style bit-operations on integers... *)
 structure <:SignedBitops:> = IntBitOps(structure I = Signed
 structure W = Unsigned)
 end
 structure Short =
 struct
 structure Signed = Int16
 structure Unsigned = Word16
 (* word-style bit-operations on integers... *)
 structure <:SignedBitops:> = IntBitOps(structure I = Signed
 structure W = Unsigned)
 end
 structure Int =
 struct
 structure Signed = Int32
 structure Unsigned = Word32
 (* word-style bit-operations on integers... *)
 structure <:SignedBitops:> = IntBitOps(structure I = Signed
 structure W = Unsigned)
 end
 structure Long =
 struct
 structure Signed = Int32
 structure Unsigned = Word32
 (* word-style bit-operations on integers... *)
 structure <:SignedBitops:> = IntBitOps(structure I = Signed
 structure W = Unsigned)
 end
 structure <:LongLong:> =
 struct
 structure Signed = Int64
 structure Unsigned = Word64
 (* word-style bit-operations on integers... *)
 structure <:SignedBitops:> = IntBitOps(structure I = Signed
 structure W = Unsigned)
 end
 structure Float = Real32
 structure Double = Real64
end
```

This would appear to be a better interface, even when an implementation must choose `Int32` and `Word32` as the representation for smaller C-types.

## 148 MLRISCLibrary

The **MLRISC Library** is a framework for retargetable and optimizing compiler back ends. The MLRISC Library is distributed with SML/NJ. Due to differences between SML/NJ and MLton, this library will not work out-of-the box with MLton.

As of 20130706, MLton includes a port of the MLRISC Library synchronized with SML/NJ version 110.76.

### 148.1 Usage

- You can import a sub-library of the MLRISC Library into an MLB file with:

| MLB file                                                   | Description                  |
|------------------------------------------------------------|------------------------------|
| <code>\$(SML_LIB)/mlrisc-lib/mlb/ALPHA.mlb</code>          | The ALPHA backend            |
| <code>\$(SML_LIB)/mlrisc-lib/mlb/AMD64.mlb</code>          | The AMD64 backend            |
| <code>\$(SML_LIB)/mlrisc-lib/mlb/AMD64-Peephole.mlb</code> | The AMD64 peephole optimizer |
| <code>\$(SML_LIB)/mlrisc-lib/mlb/CCall.mlb</code>          |                              |
| <code>\$(SML_LIB)/mlrisc-lib/mlb/CCall-sparc.mlb</code>    |                              |
| <code>\$(SML_LIB)/mlrisc-lib/mlb/CCall-x86-64.mlb</code>   |                              |
| <code>\$(SML_LIB)/mlrisc-lib/mlb/CCall-x86.mlb</code>      |                              |
| <code>\$(SML_LIB)/mlrisc-lib/mlb/Control.mlb</code>        |                              |
| <code>\$(SML_LIB)/mlrisc-lib/mlb/Graphs.mlb</code>         |                              |
| <code>\$(SML_LIB)/mlrisc-lib/mlb/HPPA.mlb</code>           | The HPPA backend             |
| <code>\$(SML_LIB)/mlrisc-lib/mlb/IA32.mlb</code>           | The IA32 backend             |
| <code>\$(SML_LIB)/mlrisc-lib/mlb/IA32-Peephole.mlb</code>  | The IA32 peephole optimizer  |
| <code>\$(SML_LIB)/mlrisc-lib/mlb/Lib.mlb</code>            |                              |
| <code>\$(SML_LIB)/mlrisc-lib/mlb/MLRISC.mlb</code>         |                              |
| <code>\$(SML_LIB)/mlrisc-lib/mlb/MLTREE.mlb</code>         |                              |
| <code>\$(SML_LIB)/mlrisc-lib/mlb/Peephole.mlb</code>       |                              |
| <code>\$(SML_LIB)/mlrisc-lib/mlb/PPC.mlb</code>            | The PPC backend              |
| <code>\$(SML_LIB)/mlrisc-lib/mlb/RA.mlb</code>             |                              |
| <code>\$(SML_LIB)/mlrisc-lib/mlb/SPARC.mlb</code>          | The Sparc backend            |
| <code>\$(SML_LIB)/mlrisc-lib/mlb/StagedAlloc.mlb</code>    |                              |
| <code>\$(SML_LIB)/mlrisc-lib/mlb/Visual.mlb</code>         |                              |

- If you are porting a project from SML/NJ's [CompilationManager](#) to MLton's [ML Basis system](#) using `cm2mlb`, note that the following map is included by default:

```
MLRISC Library
$SMLNJ-MLRISC $(SML_LIB)/mlrisc-lib/mlb
```

This will automatically convert a `$SMLNJ-MLRISC/MLRISC.cm` import in an input `.cm` file into a `$(SML_LIB)/mlrisc-lib/mlb/MLRISC.mlb` import in the output `.mlb` file.

### 148.2 Details

The following changes were made to the MLRISC Library, in addition to deriving the `.mlb` files from the `.cm` files:

- eliminate `or-patterns`: Duplicate the whole match (`p => e`) at each of the patterns.
- eliminate vector constants: Change `# [` to `Vector.fromList [`.
- eliminate `withtype` in signatures.
- eliminate sequential `withtype` expansions: Most could be rewritten as a sequence of type definitions and datatype definitions.
- eliminate higher-order functors: Every higher-order functor definition and application could be uncurried in the obvious way.
- eliminate `where <str> = <str>`: Quite painful to expand out all the flexible types in the respective structures. Furthermore, many of the implied type equalities aren't needed, but it's too hard to pick out the right ones.

### 148.3 Patch

- [MLRISC.patch](#)

## 149 MLtonArray

```
signature MLTON_ARRAY =
 sig
 val unfoldi: int * 'b * (int * 'b -> 'a * 'b) -> 'a array * 'b
 end
```

- `unfoldi (n, b, f)`  
constructs an array  $a$  of length  $n$ , whose elements  $a_i$  are determined by the equations  $a_0 = b$  and  $(a_i, b_{i+1}) = f(i, b_i)$ .

## 150 MLtonBinIO

```
signature MLTON_BIN_IO = MLTON_IO
```

See [MLtonIO](#).

---

## 151 MLtonCont

```
signature MLTON_CONT =
 sig
 type 'a t

 val callcc: ('a t -> 'a) -> 'a
 val isolate: ('a -> unit) -> 'a t
 val prepend: 'a t * ('b -> 'a) -> 'b t
 val throw: 'a t * 'a -> 'b
 val throw': 'a t * (unit -> 'a) -> 'b
 end
```

- `type 'a t`  
the type of continuations that expect a value of type `'a`.
- `callcc f`  
applies `f` to the current continuation. This copies the entire stack; hence, `callcc` takes time proportional to the size of the current stack.
- `isolate f`  
creates a continuation that evaluates `f` in an empty context. This is a constant time operation, and yields a constant size stack.
- `prepend (k, f)`  
composes a function `f` with a continuation `k` to create a continuation that first does `f` and then does `k`. This is a constant time operation.
- `throw (k, v)`  
throws value `v` to continuation `k`. This copies the entire stack of `k`; hence, `throw` takes time proportional to the size of this stack.
- `throw' (k, th)`  
a generalization of `throw` that evaluates `th ()` in the context of `k`. Thus, for example, if `th ()` raises an exception or captures another continuation, it will see `k`, not the current continuation.

### 151.1 Also see

- [MLtonContIsolateImplementation](#)

## 152 MLtonContIsolateImplementation

As noted before, it is fairly easy to get the operational behavior of `isolate` with just `callcc` and `throw`, but establishing the right space behavior is trickier. Here, we show how to start from the obvious, but inefficient, implementation of `isolate` using only `callcc` and `throw`, and *derive* an equivalent, but more efficient, implementation of `isolate` using MLton's primitive stack capture and copy operations. This isn't a formal derivation, as we are not formally showing the equivalence of the programs (though I believe that they are all equivalent, modulo the space behavior).

Here is a direct implementation of `isolate` using only `callcc` and `throw`:

```
val isolate: ('a -> unit) -> 'a t =
 fn (f: 'a -> unit) =>
 callcc
 (fn k1 =>
 let
 val x = callcc (fn k2 => throw (k1, k2))
 val _ = (f x ; Exit.topLevelSuffix ())
 handle exn => MLtonExn.topLevelHandler exn
 in
 raise Fail "MLton.Cont.isolate: return from (wrapped) func"
 end)
```

We use the standard nested `callcc` trick to return a continuation that is ready to receive an argument, execute the isolated function, and exit the program. Both `Exit.topLevelSuffix` and `MLtonExn.topLevelHandler` will terminate the program.

Throwing to an isolated function will execute the function in a *semantically* empty context, in the sense that we never re-execute the *original* continuation of the call to `isolate` (i.e., the context that was in place at the time `isolate` was called). However, we assume that the compiler isn't able to recognize that the *original* continuation is unused; for example, while we (the programmer) know that `Exit.topLevelSuffix` and `MLtonExn.topLevelHandler` will terminate the program, the compiler may only see opaque calls to unknown foreign-functions. So, that original continuation (in its entirety) is part of the continuation returned by `isolate` and throwing to the continuation returned by `isolate` will execute `f x` (with the exit wrapper) in the context of that original continuation. Thus, the garbage collector will retain everything reachable from that original continuation during the evaluation of `f x`, even though it is *semantically* garbage.

Note that this space-leak is independent of the implementation of continuations (it arises in both MLton's stack copying implementation of continuations and would arise in SML/NJ's CPS-translation implementation); we are only assuming that the implementation can't *see* the program termination, and so must retain the original continuation (and anything reachable from it).

So, we need an *empty* continuation in which to execute `f x`. (No surprise there, as that is the written description of `isolate`.) To do this, we capture a top-level continuation and throw to that in order to execute `f x`:

```
local
val base: (unit -> unit) t =
 callcc
 (fn k1 =>
 let
 val th = callcc (fn k2 => throw (k1, k2))
 val _ = (th () ; Exit.topLevelSuffix ())
 handle exn => MLtonExn.topLevelHandler exn
 in
 raise Fail "MLton.Cont.isolate: return from (wrapped) func"
 end)
in
val isolate: ('a -> unit) -> 'a t =
 fn (f: 'a -> unit) =>
 callcc
 (fn k1 =>
 let
 val x = callcc (fn k2 => throw (k1, k2))
 in
 throw (base, fn () => f x)
```

```

 end)
end

```

We presume that `base` is evaluated *early* in the program. There is a subtlety here, because one needs to believe that this `base` continuation (which technically corresponds to the entire rest of the program evaluation) *works* as an empty context; in particular, we want it to be the case that executing `f x` in the `base` context retains less space than executing `f x` in the context in place at the call to `isolate` (as occurred in the previous implementation of `isolate`). This isn't particularly easy to believe if one takes a normal substitution-based operational semantics, because it seems that the context captured and bound to `base` is arbitrarily large. However, this context is mostly unevaluated code; the only heap-allocated values that are reachable from it are those that were evaluated before the evaluation of `base` (and used in the program after the evaluation of `base`). Assuming that `base` is evaluated *early* in the program, we conclude that there are few heap-allocated values reachable from its continuation. In contrast, the previous implementation of `isolate` could capture a context that has many heap-allocated values reachable from it (because we could evaluate `isolate f late` in the program and *deep* in a call stack), which would all remain reachable during the evaluation of `f x`. [We'll return to this point later, as it is taking a slightly MLton-esque view of the evaluation of a program, and may not apply as strongly to other implementations (e.g., SML/NJ).]

Now, once we throw to `base` and begin executing `f x`, only the heap-allocated values reachable from `f` and `x` and the few heap-allocated values reachable from `base` are retained by the garbage collector. So, it seems that `base` *works* as an empty context.

But, what about the continuation returned from `isolate f`? Note that the continuation returned by `isolate` is one that receives an argument `x` and then throws to `base` to evaluate `f x`. If we used a CPS-translation implementation (and assume sufficient beta-contractions to eliminate administrative redexes), then the original continuation passed to `isolate` (i.e., the continuation bound to `k1`) will not be free in the continuation returned by `isolate f`. Rather, the only free variables in the continuation returned by `isolate f` will be `base` and `f`, so the only heap-allocated values reachable from the continuation returned by `isolate f` will be those values reachable from `base` (assumed to be few) and those values reachable from `f` (necessary in order to execute `f` at some later point).

But, MLton doesn't use a CPS-translation implementation. Rather, at each call to `callcc` in the body of `isolate`, MLton will copy the current execution stack. Thus, `k2` (the continuation returned by `isolate f`) will include execution stack at the time of the call to `isolate f`—that is, it will include the *original* continuation of the call to `isolate f`. Thus, the heap-allocated values reachable from the continuation returned by `isolate f` will include those values reachable from `base`, those values reachable from `f`, and those values reachable from the original continuation of the call to `isolate f`. So, just holding on to the continuation returned by `isolate f` will retain all of the heap-allocated values live at the time `isolate f` was called. This leaks space, since, *semantically*, the continuation returned by `isolate f` only needs the heap-allocated values reachable from `f` (and `base`).

In practice, this probably isn't a significant issue. A common use of `isolate` is implement `abort`:

```

fun abort th = throw (isolate th, ())

```

The continuation returned by `isolate th` is dead immediately after being thrown to—the continuation isn't retained, so neither is the *semantic* garbage it would have retained.

But, it is easy enough to *move* onto the *empty* context `base` the capturing of the context that we want to be returned by `isolate f`:

```

local
val base: (unit -> unit) t =
 callcc
 (fn k1 =>
 let
 val th = callcc (fn k2 => throw (k1, k2))
 val _ = (th () ; Exit.topLevelSuffix ())
 handle exn => MLtonExn.topLevelHandler exn
 in
 raise Fail "MLton.Cont.isolate: return from (wrapped) func"
 end)
in
val isolate: ('a -> unit) -> 'a t =
 fn (f: 'a -> unit) =>

```

```

callcc
(fn k1 =>
 throw (base, fn () =>
 let
 val x = callcc (fn k2 => throw (k1, k2))
 in
 throw (base, fn () => f x)
 end))
end

```

This implementation now has the right space behavior; the continuation returned by `isolate f` will only retain the heap-allocated values reachable from `f` and from `base`. (Technically, the continuation will retain two copies of the stack that was in place at the time `base` was evaluated, but we are assuming that that stack small.)

One minor inefficiency of this implementation (given MLton's implementation of continuations) is that every `callcc` and `throw` entails copying a stack (albeit, some of them are small). We can avoid this in the evaluation of `base` by using a reference cell, because `base` is evaluated at the top-level:

```

local
val base: (unit -> unit) option t =
 let
 val baseRef: (unit -> unit) option t option ref = ref NONE
 val th = callcc (fn k => (base := SOME k; NONE))
 in
 case th of
 NONE => (case !baseRef of
 NONE => raise Fail "MLton.Cont.isolate: missing base"
 | SOME base => base)
 | SOME th => let
 val _ = (th () ; Exit.topLevelSuffix ())
 handle exn => MLtonExn.topLevelHandler exn
 in
 raise Fail "MLton.Cont.isolate: return from (wrapped)
 func"
 end
 end
 end
in
val isolate: ('a -> unit) -> 'a t =
 fn (f: 'a -> unit) =>
 callcc
 (fn k1 =>
 throw (base, SOME (fn () =>
 let
 val x = callcc (fn k2 => throw (k1, k2))
 in
 throw (base, SOME (fn () => f x))
 end)))
 end))
end

```

Now, to evaluate `base`, we only copy the stack once (instead of 3 times). Because we don't have a dummy continuation around to initialize the reference cell, the reference cell holds a continuation `option`. To distinguish between the original evaluation of `base` (when we want to return the continuation) and the subsequent evaluations of `base` (when we want to evaluate a thunk), we capture a `(unit -> unit) option` continuation.

This seems to be as far as we can go without exploiting the concrete implementation of continuations in `MLtonCont`. Examining the implementation, we note that the type of continuations is given by

```
type 'a t = (unit -> 'a) -> unit
```

and the implementation of `throw` is given by

```
fun ('a, 'b) throw' (k: 'a t, v: unit -> 'a): 'b =
```

```
(k v; raise Fail "MLton.Cont.throw': return from continuation")
```

```
fun ('a, 'b) throw (k: 'a t, v: 'a): 'b = throw' (k, fn () => v)
```

Suffice to say, a continuation is simply a function that accepts a thunk to yield the thrown value and the body of the function performs the actual throw. Using this knowledge, we can create a dummy continuation to initialize `baseRef` and greatly simplify the body of `isolate`:

```
local
val base: (unit -> unit) option t =
 let
 val baseRef: (unit -> unit) option t ref =
 ref (fn _ => raise Fail "MLton.Cont.isolate: missing base")
 val th = callcc (fn k => (baseRef := k; NONE))
 in
 case th of
 NONE => !baseRef
 | SOME th => let
 val _ = (th () ; Exit.topLevelSuffix ())
 handle exn => MLtonExn.topLevelHandler exn
 in
 raise Fail "MLton.Cont.isolate: return from (wrapped)
 func"
 end
 end
 in
val isolate: ('a -> unit) -> 'a t =
 fn (f: 'a -> unit) =>
 fn (v: unit -> 'a) =>
 throw (base, SOME (f o v))
end
```

Note that this implementation of `isolate` makes it clear that the continuation returned by `isolate f` only retains the heap-allocated values reachable from `f` and `base`. It also retains only one copy of the stack that was in place at the time `base` was evaluated. Finally, it completely avoids making any copies of the stack that is in place at the time `isolate f` is evaluated; indeed, `isolate f` is a constant-time operation.

Next, suppose we limited ourselves to capturing `unit` continuations with `callcc`. We can't pass the thunk to be evaluated in the *empty* context directly, but we can use a reference cell.

```
local
val thRef: (unit -> unit) option ref = ref NONE
val base: unit t =
 let
 val baseRef: unit t ref =
 ref (fn _ => raise Fail "MLton.Cont.isolate: missing base")
 val () = callcc (fn k => baseRef := k)
 in
 case !thRef of
 NONE => !baseRef
 | SOME th =>
 let
 val _ = thRef := NONE
 val _ = (th () ; Exit.topLevelSuffix ())
 handle exn => MLtonExn.topLevelHandler exn
 in
 raise Fail "MLton.Cont.isolate: return from (wrapped) func"
 end
 end
 in
val isolate: ('a -> unit) -> 'a t =
 fn (f: 'a -> unit) =>
```

```

fn (v: unit -> 'a) =>
let
 val () = thRef := SOME (f o v)
in
 throw (base, ())
end
end

```

Note that it is important to set `thRef` to `NONE` before evaluating the thunk, so that the garbage collector doesn't retain all the heap-allocated values reachable from `f` and `v` during the evaluation of `f (v ())`. This is because `thRef` is still live during the evaluation of the thunk; in particular, it was allocated before the evaluation of `base` (and used after), and so is retained by continuation on which the thunk is evaluated.

This implementation can be easily adapted to use MLton's primitive stack copying operations.

```

local
val thRef: (unit -> unit) option ref = ref NONE
val base: Thread.preThread =
 let
 val () = Thread.copyCurrent ()
 in
 case !thRef of
 NONE => Thread.savedPre ()
 | SOME th =>
 let
 val () = thRef := NONE
 val _ = (th () ; Exit.topLevelSuffix ())
 handle exn => MLtonExn.topLevelHandler exn
 in
 raise Fail "MLton.Cont.isolate: return from (wrapped) func"
 end
 end
 in
 val isolate: ('a -> unit) -> 'a t =
 fn (f: 'a -> unit) =>
 fn (v: unit -> 'a) =>
 let
 val () = thRef := SOME (f o v)
 val new = Thread.copy base
 in
 Thread.switchTo new
 end
 end
 end
end

```

In essence, `Thread.copyCurrent` copies the current execution stack and stores it in an implicit reference cell in the runtime system, which is fetchable with `Thread.savedPre`. When we are ready to throw to the isolated function, `Thread.copy` copies the saved execution stack (because the stack is modified in place during execution, we need to retain a pristine copy in case the isolated function itself throws to other isolated functions) and `Thread.switchTo` abandons the current execution stack, installing the newly copied execution stack.

The actual implementation of `MLton.Cont.isolate` simply adds some `Thread.atomicBegin` and `Thread.atomicEnd` commands, which effectively protect the global `thRef` and accommodate the fact that `Thread.switchTo` does an implicit `Thread.atomicEnd` (used for leaving a signal handler thread).

```

local
val thRef: (unit -> unit) option ref = ref NONE
val base: Thread.preThread =
 let
 val () = Thread.copyCurrent ()
 in
 case !thRef of
 NONE => Thread.savedPre ()

```

```

 | SOME th =>
 let
 val () = thRef := NONE
 val _ = MLton.atomicEnd (* Match 1 *)
 val _ = (th () ; Exit.topLevelSuffix ())
 handle exn => MLtonExn.topLevelHandler exn
 in
 raise Fail "MLton.Cont.isolate: return from (wrapped) func"
 end
 end
 in
 val isolate: ('a -> unit) -> 'a t =
 fn (f: 'a -> unit) =>
 fn (v: unit -> 'a) =>
 let
 val _ = MLton.atomicBegin (* Match 1 *)
 val () = thRef := SOME (f o v)
 val new = Thread.copy base
 val _ = MLton.atomicBegin (* Match 2 *)
 in
 Thread.switchTo new (* Match 2 *)
 end
 end
 end
end

```

It is perhaps interesting to note that the above implementation was originally *derived* by specializing implementations of the [MLtonThread](#) `new`, `prepare`, and `switch` functions as if their only use was in the following implementation of `isolate`:

```

val isolate: ('a -> unit) -> 'a t =
 fn (f: 'a -> unit) =>
 fn (v: unit -> 'a) =>
 let
 val th = (f (v ()) ; Exit.topLevelSuffix ())
 handle exn => MLtonExn.topLevelHandler exn
 val t = MLton.Thread.prepare (MLton.Thread.new th, ())
 in
 MLton.Thread.switch (fn _ => t)
 end
end

```

It was pleasant to discover that it could equally well be *derived* starting from the `callcc` and `throw` implementation.

As a final comment, we noted that the degree to which the context of `base` could be considered *empty* (i.e., retaining few heap-allocated values) depended upon a slightly MLton-esque view. In particular, MLton does not heap allocate executable code. So, although the `base` context keeps a lot of unevaluated code *live*, such code is not heap allocated. In a system like SML/NJ, that does heap allocate executable code, one might want it to be the case that after throwing to an isolated function, the garbage collector retains only the code necessary to evaluate the function, and not any code that was necessary to evaluate the `base` context.

## 153 MLtonCross

The debian package MLton-Cross adds various targets to MLton. In combination with the emdebian project, this allows a debian system to compile SML files to other architectures.

Currently, these targets are supported:

- *Windows (MinGW)*
  - -target i586-mingw32msvc (mlton-target-i586-mingw32msvc)
  - -target amd64-mingw32msvc( mlton-target-amd64-mingw32msvc)
- *Linux (Debian)*
  - -target alpha-linux-gnu (mlton-target-alpha-linux-gnu)
  - -target arm-linux-gnueabi (mlton-target-arm-linux-gnueabi)
  - -target hppa-linux-gnu (mlton-target-hppa-linux-gnu)
  - -target i486-linux-gnu (mlton-target-i486-linux-gnu)
  - -target ia64-linux-gnu (mlton-target-ia64-linux-gnu)
  - -target mips-linux-gnu (mlton-target-mips-linux-gnu)
  - -target mipsel-linux-gnu (mlton-target-mipsel-linux-gnu)
  - -target powerpc-linux-gnu (mlton-target-powerpc-linux-gnu)
  - -target s390-linux-gnu (mlton-target-s390-linux-gnu)
  - -target sparc-linux-gnu (mlton-target-sparc-linux-gnu)
  - -target x86-64-linux-gnu (mlton-target-x86-64-linux-gnu)

### 153.1 Download

MLton-Cross is kept in-sync with the current MLton release.

- [mlton-cross\\_20100608.orig.tar.gz](#)

## 154 MLtonExn

```
signature MLTON_EXN =
 sig
 val addExnMessenger: (exn -> string option) -> unit
 val history: exn -> string list

 val defaultTopLevelHandler: exn -> 'a
 val getTopLevelHandler: unit -> (exn -> unit)
 val setTopLevelHandler: (exn -> unit) -> unit
 val topLevelHandler: exn -> 'a
 end
```

- `addExnMessenger f`  
adds `f` as a pretty-printer to be used by `General.exnMessage` for converting exceptions to strings. Messagers are tried in order from most recently added to least recently added.
- `history e`  
returns call stack at the point that `e` was first raised. Each element of the list is a file position. The elements are in reverse chronological order, i.e. the function called last is at the front of the list.  
`history e` will return `[]` unless the program is compiled with `-const 'Exn.keepHistory true'`.
- `defaultTopLevelHandler e`  
function that behaves as the default top level handler; that is, print out the unhandled exception message for `e` and exit.
- `getTopLevelHandler ()`  
get the top level handler.
- `setTopLevelHandler f`  
set the top level handler to the function `f`. The function `f` should not raise an exception or return normally.
- `topLevelHandler e`  
behaves as if the top level handler received the exception `e`.

## 155 MLtonFinalizable

```
signature MLTON_FINALIZABLE =
 sig
 type 'a t

 val addFinalizer: 'a t * ('a -> unit) -> unit
 val finalizeBefore: 'a t * 'b t -> unit
 val new: 'a -> 'a t
 val touch: 'a t -> unit
 val withValue: 'a t * ('a -> 'b) -> 'b
 end
```

A *finalizable* value is a container to which finalizers can be attached. A container holds a value, which is reachable as long as the container itself is reachable. A *finalizer* is a function that runs at some point after garbage collection determines that the container to which it is attached has become [unreachable](#). A finalizer is treated like a signal handler, in that it runs asynchronously in a separate thread, with signals blocked, and will not interrupt a critical section (see [MLtonThread](#)).

- `addFinalizer (v, f)`  
adds `f` as a finalizer to `v`. This means that sometime after the last call to `withValue` on `v` completes and `v` becomes unreachable, `f` will be called with the value of `v`.
- `finalizeBefore (v1, v2)`  
ensures that `v1` will be finalized before `v2`. A cycle of values `v = v1, ..., vn = v` with `finalizeBefore (vi, vi+1)` will result in none of the `vi` being finalized.
- `new x`  
creates a new finalizable value, `v`, with value `x`. The finalizers of `v` will run sometime after the last call to `withValue` on `v` when the garbage collector determines that `v` is unreachable.
- `touch v`  
ensures that `v`'s finalizers will not run before the call to `touch`.
- `withValue (v, f)`  
returns the result of applying `f` to the value of `v` and ensures that `v`'s finalizers will not run before `f` completes. The call to `f` is a nontail call.

### 155.1 Example

Suppose that `finalizable.sml` contains the following:

```
signature CLIST =
 sig
 type t

 val cons: int * t -> t
 val sing: int -> t
 val sum: t -> int
 end

functor CList (structure F: MLTON_FINALIZABLE
 structure P: MLTON_POINTER
 structure Prim:
 sig
 val cons: int * P.t -> P.t
 val free: P.t -> unit
 val sing: int -> P.t
```

```

 val sum: P.t -> int
 end): CLIST =
struct
 type t = P.t F.t

 fun cons (n: int, l: t) =
 F.withValue
 (l, fn w' =>
 let
 val c = F.new (Prim.cons (n, w'))
 val _ = F.addFinalizer (c, Prim.free)
 val _ = F.finalizeBefore (c, l)
 in
 c
 end)

 fun sing n =
 let
 val c = F.new (Prim.sing n)
 val _ = F.addFinalizer (c, Prim.free)
 in
 c
 end

 fun sum c = F.withValue (c, Prim.sum)
end

functor Test (structure CList: CLIST
 structure MLton: sig
 structure GC:
 sig
 val collect: unit -> unit
 end
 end) =
struct
 fun f n =
 if n = 1
 then ()
 else
 let
 val a = Array.tabulate (n, fn i => i)
 val _ = Array.sub (a, 0) + Array.sub (a, 1)
 in
 f (n - 1)
 end

 val l = CList.sing 2
 val l = CList.cons (2, l)
 val _ = MLton.GC.collect ()
 val _ = f 100
 val _ = print (concat ["listSum(1) = ",
 Int.toString (CList.sum l),
 "\n"])

 val _ = MLton.GC.collect ()
 val _ = f 100
end

```

```

structure CList =
 CList (structure F = MLton.Finalizable
 structure P = MLton.Pointer
 structure Prim =
 struct
 val cons = _import "listCons": int * P.t -> P.t;
 val free = _import "listFree": P.t -> unit;
 val sing = _import "listSing": int -> P.t;
 val sum = _import "listSum": P.t -> int;
 end)

structure S = Test (structure CList = CList
 structure MLton = MLton)

```

Suppose that `cons.c` contains the following.

```

#include <stdio.h>

typedef unsigned int uint;

typedef struct Cons {
 struct Cons *next;
 int value;
} *Cons;

Cons listCons (int n, Cons c) {
 Cons res;

 res = (Cons) malloc (sizeof(*res));
 fprintf (stderr, "0x%08x = listCons (%d)\n", (uint)res, n);
 res->next = c;
 res->value = n;
 return res;
}

Cons listSing (int n) {
 Cons res;

 res = (Cons) malloc (sizeof(*res));
 fprintf (stderr, "0x%08x = listSing (%d)\n", (uint)res, n);
 res->next = NULL;
 res->value = n;
 return res;
}

void listFree (Cons p) {
 fprintf (stderr, "listFree (0x%08x)\n", (uint)p);
 free (p);
}

int listSum (Cons c) {
 int res;

 fprintf (stderr, "listSum\n");
 res = 0;
 for (; c != NULL; c = c->next)
 res += c->value;
 return res;
}

```

We can compile these to create an executable with

```
% mlton -default-ann 'allowFFI true' finalizable.sml cons.c
```

Running this executable will create output like the following.

```
% finalizable
0x08072890 = listSing (2)
0x080728a0 = listCons (2)
0x080728b0 = listCons (2)
0x080728c0 = listCons (2)
0x080728d0 = listCons (2)
0x080728e0 = listCons (2)
0x080728f0 = listCons (2)
listSum
listSum(1) = 14
listFree (0x080728f0)
listFree (0x080728e0)
listFree (0x080728d0)
listFree (0x080728c0)
listFree (0x080728b0)
listFree (0x080728a0)
listFree (0x08072890)
```

## 155.2 Synchronous Finalizers

Finalizers in MLton are asynchronous. That is, they run at an unspecified time, interrupting the user program. It is also possible, and sometimes useful, to have synchronous finalizers, where the user program explicitly decides when to run enabled finalizers. We have considered this in MLton, and it seems possible, but there are some unresolved design issues. See the thread at

- <http://www.mlton.org/pipermail/mlton/2004-September/016570.html>

## 155.3 Also see

- [Boehm03](#)

## 156 MLtonGC

```
signature MLTON_GC =
 sig
 val collect: unit -> unit
 val pack: unit -> unit
 val setMessages: bool -> unit
 val setSummary: bool -> unit
 val unpack: unit -> unit
 structure Statistics :
 sig
 val bytesAllocated: unit -> IntInf.int
 val lastBytesLive: unit -> IntInf.int
 val numCopyingGCs: unit -> IntInf.int
 val numMarkCompactGCs: unit -> IntInf.int
 val numMinorGCs: unit -> IntInf.int
 val maxBytesLive: unit -> IntInf.int
 end
 end
end
```

- `collect ()`  
causes a garbage collection to occur.
  - `pack ()`  
shrinks the heap as much as possible so that other processes can use available RAM.
  - `setMessages b`  
controls whether diagnostic messages are printed at the beginning and end of each garbage collection. It is the same as the `gc-messages` runtime system option.
  - `setSummary b`  
controls whether a summary of garbage collection statistics is printed upon termination of the program. It is the same as the `gc-summary` runtime system option.
  - `unpack ()`  
resizes a packed heap to the size desired by the runtime.
  - `Statistics.bytesAllocated ()`  
returns bytes allocated (as of the most recent garbage collection).
  - `Statistics.lastBytesLive ()`  
returns bytes live (as of the most recent garbage collection).
  - `Statistics.numCopyingGCs ()`  
returns number of (major) copying garbage collections performed (as of the most recent garbage collection).
  - `Statistics.numMarkCompactGCs ()`  
returns number of (major) mark-compact garbage collections performed (as of the most recent garbage collection).
  - `Statistics.numMinorGCs ()`  
returns number of minor garbage collections performed (as of the most recent garbage collection).
  - `Statistics.maxBytesLive ()`  
returns maximum bytes live (as of the most recent garbage collection).
-

## 157 MLtonIntInf

```
signature MLTON_INT_INF =
 sig
 type t = IntInf.int

 val areSmall: t * t -> bool
 val gcd: t * t -> t
 val isSmall: t -> bool

 structure BigWord : WORD
 structure SmallInt : INTEGER
 datatype rep =
 Big of BigWord.word vector
 | Small of SmallInt.int
 val rep: t -> rep
 val fromRep : rep -> t
 end
```

MLton represents an arbitrary precision integer either as an unboxed word with the bottom bit set to 1 and the top bits representing a small signed integer, or as a pointer to a vector of words, where the first word indicates the sign and the rest are the limbs of a [GnuMP](#) big integer.

- type t  
the same as type `IntInf.int`.
- `areSmall (a, b)`  
returns true iff both a and b are small.
- `gcd (a, b)`  
uses the [GnuMP](#)'s fast gcd implementation.
- `isSmall a`  
returns true iff a is small.
- `BigWord : WORD`  
representation of a big `IntInf.int` as a vector of words; on 32-bit platforms, `BigWord` is likely to be equivalent to `Word32`, and on 64-bit platforms, `BigWord` is likely to be equivalent to `Word64`.
- `SmallInt : INTEGER`  
representation of a small `IntInf.int` as a signed integer; on 32-bit platforms, `SmallInt` is likely to be equivalent to `Int32`, and on 64-bit platforms, `SmallInt` is likely to be equivalent to `Int64`.
- datatype `rep`  
the underlying representation of an `IntInf.int`.
- `rep i`  
returns the underlying representation of `i`.
- `fromRep r`  
converts from the underlying representation back to `i`. If the input is not identical to the result of `rep`, the result is undefined.

## 158 MLtonIO

```
signature MLTON_IO =
 sig
 type instream
 type ostream

 val inFd: instream -> Posix.IO.file_desc
 val mkstemp: string -> string * ostream
 val mkstemp: {prefix: string, suffix: string} -> string * ostream
 val newIn: Posix.IO.file_desc * string -> instream
 val newOut: Posix.IO.file_desc * string -> ostream
 val outFd: ostream -> Posix.IO.file_desc
 val tempPrefix: string -> string
 end
```

- `inFd ins`  
returns the file descriptor corresponding to `ins`.
- `mkstemp s`  
like the C `mkstemp` function, generates and open a temporary file with prefix `s`.
- `mkstemp {prefix, suffix}`  
like `mkstemp`, except it has both a prefix and suffix.
- `newIn (fd, name)`  
creates a new instream from file descriptor `fd`, with `name` used in any `Io` exceptions later raised.
- `newOut (fd, name)`  
creates a new ostream from file descriptor `fd`, with `name` used in any `Io` exceptions later raised.
- `outFd out`  
returns the file descriptor corresponding to `out`.
- `tempPrefix s`  
adds a suitable system or user specific prefix (directory) for temp files.

## 159 MLtonltimer

```
signature MLTON_ITIMER =
 sig
 datatype t =
 Prof
 | Real
 | Virtual

 val set: t * {interval: Time.time, value: Time.time} -> unit
 val signal: t -> Posix.Signal.signal
 end
```

- `set (t, {interval, value})`  
sets the interval timer (using `setitimer`) specified by `t` to the given interval and value.
- `signal t`  
returns the signal corresponding to `t`.

## 160 MLtonLibraryProject

We have a [MLton Library repository](#) that is intended to collect libraries.

<https://github.com/MLton/mltonlib>

Libraries are kept in the `master` branch, and are grouped according to domain name, in the Java package style. For example, [VesaKarvonen](#), who works at `ssh.com`, has been putting code at:

<https://github.com/MLton/mltonlib/tree/master/com/ssh>

[StephenWeeks](#), owning `sweeks.com`, has been putting code at:

<https://github.com/MLton/mltonlib/tree/master/com/sweeks>

A "library" is a subdirectory of some such directory. For example, Stephen's basis-library replacement library is at

<https://github.com/MLton/mltonlib/tree/master/com/sweeks/basic>

We use "transparent per-library branching" to handle library versioning. Each library has an "unstable" subdirectory in which work happens. When one is happy with a library, one tags it by copying it to a stable version directory. Stable libraries are immutable — when one refers to a stable library, one always gets exactly the same code. No one has actually made a stable library yet, but, when I'm ready to tag my library, I was thinking that I would do something like copying

<https://github.com/MLton/mltonlib/tree/master/com/sweeks/basic/unstable>

to

<https://github.com/MLton/mltonlib/tree/master/com/sweeks/basic/v1>

So far, libraries in the MLton repository have been licensed under MLton's [License](#). We haven't decided on whether that will be a requirement to be in the repository or not. For the sake of simplicity (a single license) and encouraging widest use of code, contributors are encouraged to use that license. But it may be too strict to require it.

If someone wants to contribute a new library to our repository or to work on an old one, they can make a pull request. If people want to work in their own repository, they can do so — that's the point of using domain names to prevent clashes. The idea is that a user should be able to bring library collections in from many different repositories without problems. And those libraries could even work with each other.

At some point we may want to settle on an [MLBasisPathMap](#) variable for the root of the library project. Or, we could reuse `SML_LIB`, and migrate what we currently keep there into the library infrastructure.

## 161 MLtonMonoArray

```
signature MLTON_MONO_ARRAY =
 sig
 type t
 type elem
 val fromPoly: elem array -> t
 val toPoly: t -> elem array
 end
```

- type t  
 type of monomorphic array
- type elem  
 type of array elements
- fromPoly a  
 type cast a polymorphic array to its monomorphic counterpart; the argument and result arrays share the same identity
- toPoly a  
 type cast a monomorphic array to its polymorphic counterpart; the argument and result arrays share the same identity

## 162 MLtonMonoVector

```
signature MLTON_MONO_VECTOR =
 sig
 type t
 type elem
 val fromPoly: elem vector -> t
 val toPoly: t -> elem vector
 end
```

- type t  
 type of monomorphic vector
- type elem  
 type of vector elements
- fromPoly v  
 type cast a polymorphic vector to its monomorphic counterpart; in MLton, this is a constant-time operation
- toPoly v  
 type cast a monomorphic vector to its polymorphic counterpart; in MLton, this is a constant-time operation

## 163 MLtonPlatform

```
signature MLTON_PLATFORM =
 sig
 structure Arch:
 sig
 datatype t = Alpha | AMD64 | ARM | ARM64 | HPPA | IA64 | m68k
 | MIPS | PowerPC | PowerPC64 | S390 | Sparc | X86

 val fromString: string -> t option
 val host: t
 val toString: t -> string
 end

 structure OS:
 sig
 datatype t = AIX | Cygwin | Darwin | FreeBSD | Hurd | HPUX
 | Linux | MinGW | NetBSD | OpenBSD | Solaris

 val fromString: string -> t option
 val host: t
 val toString: t -> string
 end
 end
 end
```

- datatype Arch.t  
processor architectures
- Arch.fromString a  
converts from string to architecture. Case insensitive.
- Arch.host  
the architecture for which the program is compiled.
- Arch.toString  
string for architecture.
- datatype OS.t  
operating systems
- OS.fromString  
converts from string to operating system. Case insensitive.
- OS.host  
the operating system for which the program is compiled.
- OS.toString  
string for operating system.

## 164 MLtonPointer

```
signature MLTON_POINTER =
 sig
 eqtype t

 val add: t * word -> t
 val compare: t * t -> order
 val diff: t * t -> word
 val getInt8: t * int -> Int8.int
 val getInt16: t * int -> Int16.int
 val getInt32: t * int -> Int32.int
 val getInt64: t * int -> Int64.int
 val getPointer: t * int -> t
 val getReal32: t * int -> Real32.real
 val getReal64: t * int -> Real64.real
 val getWord8: t * int -> Word8.word
 val getWord16: t * int -> Word16.word
 val getWord32: t * int -> Word32.word
 val getWord64: t * int -> Word64.word
 val null: t
 val setInt8: t * int * Int8.int -> unit
 val setInt16: t * int * Int16.int -> unit
 val setInt32: t * int * Int32.int -> unit
 val setInt64: t * int * Int64.int -> unit
 val setPointer: t * int * t -> unit
 val setReal32: t * int * Real32.real -> unit
 val setReal64: t * int * Real64.real -> unit
 val setWord8: t * int * Word8.word -> unit
 val setWord16: t * int * Word16.word -> unit
 val setWord32: t * int * Word32.word -> unit
 val setWord64: t * int * Word64.word -> unit
 val sizeofPointer: word
 val sub: t * word -> t
 end
```

- `eqtype t`  
the type of pointers, i.e. machine addresses.
- `add (p, w)`  
returns the pointer `w` bytes after than `p`. Does not check for overflow.
- `compare (p1, p2)`  
compares the pointer `p1` to the pointer `p2` (as addresses).
- `diff (p1, p2)`  
returns the number of bytes `w` such that `add (p2, w) = p1`. Does not check for overflow.
- `get<X> (p, i)`  
returns the object stored at index `i` of the array of `X` objects pointed to by `p`. For example, `getWord32 (p, 7)` returns the 32-bit word stored 28 bytes beyond `p`.
- `null`  
the null pointer, i.e. 0.
- `set<X> (p, i, v)`  
assigns `v` to the object stored at index `i` of the array of `X` objects pointed to by `p`. For example, `setWord32 (p, 7, w)` stores the 32-bit word `w` at the address 28 bytes beyond `p`.

- `sizeofPointer`  
size, in bytes, of a pointer.
  - `sub (p, w)`  
returns the pointer `w` bytes before `p`. Does not check for overflow.
-

## 165 MLtonProcEnv

```
signature MLTON_PROC_ENV =
 sig
 type gid

 val setenv: {name: string, value: string} -> unit
 val setgroups: gid list -> unit
 end
```

- `setenv {name, value}`  
like the C `setenv` function. Does not require name or value to be null terminated.
- `setgroups grps`  
like the C `setgroups` function.

## 166 MLtonProcess

```
signature MLTON_PROCESS =
 sig
 type pid

 val spawn: {args: string list, path: string} -> pid
 val spawnenv: {args: string list, env: string list, path: string} -> pid
 val spawnp: {args: string list, file: string} -> pid

 type ('stdin, 'stdout, 'stderr) t

 type input
 type output

 type none
 type chain
 type any

 exception MisuseOfForget
 exception DoublyRedirected

 structure Child:
 sig
 type ('use, 'dir) t

 val binIn: (BinIO.instream, input) t -> BinIO.instream
 val binOut: (BinIO.outstream, output) t -> BinIO.outstream
 val fd: (Posix.FileSys.file_desc, 'dir) t -> Posix.FileSys.file_desc
 val remember: (any, 'dir) t -> ('use, 'dir) t
 val textIn: (TextIO.instream, input) t -> TextIO.instream
 val textOut: (TextIO.outstream, output) t -> TextIO.outstream
 end

 structure Param:
 sig
 type ('use, 'dir) t

 val child: (chain, 'dir) Child.t -> (none, 'dir) t
 val fd: Posix.FileSys.file_desc -> (none, 'dir) t
 val file: string -> (none, 'dir) t
 val forget: ('use, 'dir) t -> (any, 'dir) t
 val null: (none, 'dir) t
 val pipe: ('use, 'dir) t
 val self: (none, 'dir) t
 end

 val create:
 {args: string list,
 env: string list option,
 path: string,
 stderr: ('stderr, output) Param.t,
 stdin: ('stdin, input) Param.t,
 stdout: ('stdout, output) Param.t}
 -> ('stdin, 'stdout, 'stderr) t
 val getStderr: ('stdin, 'stdout, 'stderr) t -> ('stderr, input) Child.t
 val getStdin: ('stdin, 'stdout, 'stderr) t -> ('stdin, output) Child.t
 val getStdout: ('stdin, 'stdout, 'stderr) t -> ('stdout, input) Child.t
 val kill: ('stdin, 'stdout, 'stderr) t * Posix.Signal.signal -> unit
 val reap: ('stdin, 'stdout, 'stderr) t -> Posix.Process.exit_status
end
```

## 166.1 Spawn

The `spawn` functions provide an alternative to the `fork/exec` idiom that is typically used to create a new process. On most platforms, the `spawn` functions are simple wrappers around `fork/exec`. However, under Windows, the `spawn` functions are primitive. All `spawn` functions return the process id of the spawned process. They differ in how the executable is found and the environment that it uses.

- `spawn {args, path}`  
starts a new process running the executable specified by `path` with the arguments `args`. Like `Posix.Process.exec`.
- `spawnenv {args, env, path}`  
starts a new process running the executable specified by `path` with the arguments `args` and environment `env`. Like `Posix.Process.exece`.
- `spawnp {args, file}`  
search the `PATH` environment variable for an executable named `file`, and start a new process running that executable with the arguments `args`. Like `Posix.Process.execp`.

## 166.2 Create

`MLton.Process.create` provides functionality similar to `Unix.executeInEnv`, but provides more control over the input, output, and error streams. In addition, `create` works on all platforms, including Cygwin and MinGW (Windows) where `Posix.fork` is unavailable. For greatest portability programs should still use the standard `Unix.execute`, `Unix.executeInEnv`, and `OS.Process.system`.

The following types and sub-structures are used by the `create` function. They provide static type checking of correct stream usage.

### 166.2.1 Child

- `('use, 'dir) Child.t`  
This represents a handle to one of a child's standard streams. The `'dir` is viewed with respect to the parent. Thus a `('a, input) Child.t` handle means that the parent may input the output from the child.
- `Child.{bin,text}{In,Out} h`  
These functions take a handle and bind it to a stream of the named type. The type system will detect attempts to reverse the direction of a stream or to use the same stream in multiple, incompatible ways.
- `Child.fd h`  
This function behaves like the other `Child.*` functions; it opens a stream. However, it does not enforce that you read or write from the handle. If you use the descriptor in an inappropriate direction, the behavior is undefined. Furthermore, this function may potentially be unavailable on future MLton host platforms.
- `Child.remember h`  
This function takes a stream of use `any` and resets the use of the stream so that the stream may be used by `Child.*`. An `any` stream may have had use `none` or `'use` prior to calling `Param.forget`. If the stream was `none` and is used, `MisuseOfForget` is raised.

### 166.2.2 Param

- `('use, 'dir) Param.t`  
This is a handle to an input/output source and will be passed to the created child process. The `'dir` is relative to the child process. Input means that the child process will read from this stream.

- `Param.child h`  
Connect the stream of the new child process to the stream of a previously created child process. A single child stream should be connected to only one child process or else `DoublyRedirected` will be raised.
- `Param.fd fd`  
This creates a stream from the provided file descriptor which will be closed when `create` is called. This function may not be available on future MLton host platforms.
- `Param.forget h`  
This hides the type of the actual parameter as `any`. This is useful if you are implementing an application which conditionally attaches the child process to files or pipes. However, you must ensure that your use after `Child.remember` matches the original type.
- `Param.file s`  
Open the given file and connect it to the child process. Note that the file will be opened only when `create` is called. So any exceptions will be raised there and not by this function. If used for `input`, the file is opened read-only. If used for `output`, the file is opened read-write.
- `Param.null`  
In some situations, the child process should have its output discarded. The `null` param when passed as `stdout` or `stderr` does this. When used for `stdin`, the child process will either receive `EOF` or a failure condition if it attempts to read from `stdin`.
- `Param.pipe`  
This will connect the input/output of the child process to a pipe which the parent process holds. This may later form the input to one of the `Child.*` functions and/or the `Param.child` function.
- `Param.self`  
This will connect the input/output of the child process to the corresponding stream of the parent process.

### 166.2.3 Process

- `type ('stdin, 'stdout, 'stderr) t`  
represents a handle to a child process. The type arguments capture how the named stream of the child process may be used.
- `type any`  
bypasses the type system in situations where an application does not want the it to enforce correct usage. See `Child.remember` and `Param.forget`.
- `type chain`  
means that the child process's stream was connected via a pipe to the parent process. The parent process may pass this pipe in turn to another child, thus chaining them together.
- `type input, output`  
record the direction that a stream flows. They are used as a part of `Param.t` and `Child.t` and is detailed there.
- `type none`  
means that the child process's stream may not be used by the parent process. This happens when the child process is connected directly to some source.  
The types `BinIO.instream`, `BinIO.outstream`, `TextIO.instream`, `TextIO.outstream`, and `Posix.FileSys.file_desc` are also valid types with which to instantiate child streams.
- `exception MisuseOfForget`  
may be raised if `Child.remember` and `Param.forget` are used to bypass the normal type checking. This exception will only be raised in cases where the `forget` mechanism allows a misuse that would be impossible with the type-safe versions.

- `exception DoublyRedirected`  
raised if a stream connected to a child process is redirected to two separate child processes. It is safe, though bad style, to use the a `Child.t` with the same `Child.*` function repeatedly.
- `create {args, path, env, stderr, stdin, stdout}`  
starts a child process with the given command-line `args` (excluding the program name). `path` should be an absolute path to the executable run in the new child process; relative paths work, but are less robust. Optionally, the environment may be overridden with `env` where each string element has the form "key=value". The `std*` options must be provided by the `Param.*` functions documented above.  
Processes which are `create-d` must be either `reap-ed` or `kill-ed`.
- `getStd{in,out,err} proc`  
gets a handle to the specified stream. These should be used by the `Child.*` functions. Failure to use a stream connected via pipe to a child process may result in runtime dead-lock and elicits a compiler warning.
- `kill (proc, sig)`  
terminates the child process immediately. The signal may or may not mean anything depending on the host platform. A good value is `Posix.Signal.term`.
- `reap proc`  
waits for the child process to terminate and return its exit status.

### 166.3 Important usage notes

When building an application with many pipes between child processes, it is important to ensure that there are no cycles in the undirected pipe graph. If this property is not maintained, deadlocks are a very serious potential bug which may only appear under difficult to reproduce conditions.

The danger lies in that most operating systems implement pipes with a fixed buffer size. If process A has two output pipes which process B reads, it can happen that process A blocks writing to pipe 2 because it is full while process B blocks reading from pipe 1 because it is empty. This same situation can happen with any undirected cycle formed between processes (vertexes) and pipes (undirected edges) in the graph.

It is possible to make this safe using low-level I/O primitives for polling. However, these primitives are not very portable and difficult to use properly. A far better approach is to make sure you never create a cycle in the first place.

For these reasons, the `Unix.executeInEnv` is a very dangerous function. Be careful when using it to ensure that the child process only operates on either `stdin` or `stdout`, but not both.

### 166.4 Example use of `MLton.Process.create`

The following example program launches the `ipconfig` utility, pipes its output through `grep`, and then reads the result back into the program.

```
open MLton.Process
val p =
 create {args = ["/all"],
 env = NONE,
 path = "C:\\WINDOWS\\system32\\ipconfig.exe",
 stderr = Param.self,
 stdin = Param.null,
 stdout = Param.pipe}
val q =
 create {args = ["IP-Ad"],
 env = NONE,
 path = "C:\\msys\\bin\\grep.exe",
 stderr = Param.self,
 stdin = Param.child (getStdout p),
```

```
 stdout = Param.pipe}
fun suck h =
 case TextIO.inputLine h of
 NONE => ()
 | SOME s => (print (" " ^ s ^ "\n"); suck h)
val () = suck (Child.textIn (getStdout q))
```

## 167 MLtonProfile

```
signature MLTON_PROFILE =
 sig
 structure Data:
 sig
 type t

 val equals: t * t -> bool
 val free: t -> unit
 val malloc: unit -> t
 val write: t * string -> unit
 end

 val isOn: bool
 val withData: Data.t * (unit -> 'a) -> 'a
 end
 end
```

`MLton.Profile` provides [Profiling](#) control from within the program, allowing you to profile individual portions of your program. With `MLton.Profile`, you can create many units of profiling data (essentially, mappings from functions to counts) during a run of a program, switch between them while the program is running, and output multiple `mlmon.out` files.

- `isOn`  
a compile-time constant that is false only when compiling `-profile no`.
- `type Data.t`  
the type of a unit of profiling data. In order to most efficiently execute non-profiled programs, when compiling `-profile no` (the default), `Data.t` is equivalent to `unit ref`.
- `Data.equals (x, y)`  
returns true if the `x` and `y` are the same unit of profiling data.
- `Data.free x`  
frees the memory associated with the unit of profiling data `x`. It is an error to free the current unit of profiling data or to free a previously freed unit of profiling data. When compiling `-profile no`, `Data.free x` is a no-op.
- `Data.malloc ()`  
returns a new unit of profiling data. Each unit of profiling data is allocated from the process address space (but is *not* in the MLton heap) and consumes memory proportional to the number of source functions. When compiling `-profile no`, `Data.malloc ()` is equivalent to allocating a new `unit ref`.
- `write (x, f)`  
writes the accumulated ticks in the unit of profiling data `x` to file `f`. It is an error to write a previously freed unit of profiling data. When compiling `-profile no`, `write (x, f)` is a no-op. A profiled program will always write the current unit of profiling data at program exit to a file named `mlmon.out`.
- `withData (d, f)`  
runs `f` with `d` as the unit of profiling data, and returns the result of `f` after restoring the current unit of profiling data. When compiling `-profile no`, `withData (d, f)` is equivalent to `f ()`.

### 167.1 Example

Here is an example, taken from the `examples/profiling` directory, showing how to profile the executions of the `fib` and `tak` functions separately. Suppose that `fib-tak.sml` contains the following.

```

structure Profile = MLton.Profile

val fibData = Profile.Data.malloc ()
val takData = Profile.Data.malloc ()

fun wrap (f, d) x =
 Profile.withData (d, fn () => f x)

val rec fib =
 fn 0 => 0
 | 1 => 1
 | n => fib (n - 1) + fib (n - 2)
val fib = wrap (fib, fibData)

fun tak (x,y,z) =
 if not (y < x)
 then z
 else tak (tak (x - 1, y, z),
 tak (y - 1, z, x),
 tak (z - 1, x, y))
val tak = wrap (tak, takData)

val rec f =
 fn 0 => ()
 | n => (fib 38; f (n-1))
val _ = f 2

val rec g =
 fn 0 => ()
 | n => (tak (18,12,6); g (n-1))
val _ = g 500

fun done (data, file) =
 (Profile.Data.write (data, file)
 ; Profile.Data.free data)

val _ = done (fibData, "mlmon.fib.out")
val _ = done (takData, "mlmon.tak.out")

```

### Compile and run the program.

```

% mlton -profile time fib-tak.sml
% ./fib-tak

```

### Separately display the profiling data for fib

```

% mlprof fib-tak mlmon.fib.out
5.77 seconds of CPU time (0.00 seconds GC)
function cur
----- -----
fib 96.9%
<unknown> 3.1%

```

### and for tak

```

% mlprof fib-tak mlmon.tak.out
0.68 seconds of CPU time (0.00 seconds GC)
function cur
----- -----
tak 100.0%

```

Combine the data for `fib` and `tak` by calling `mlprof` with multiple `mlmon.out` files.

```
% mlprof fib-tak mlmon.fib.out mlmon.tak.out mlmon.out
6.45 seconds of CPU time (0.00 seconds GC)
function cur

fib 86.7%
tak 10.5%
<unknown> 2.8%
```

## 168 MLtonRandom

```
signature MLTON_RANDOM =
 sig
 val alphaNumChar: unit -> char
 val alphaNumString: int -> string
 val rand: unit -> word
 val seed: unit -> word option
 val srand: word -> unit
 val useed: unit -> word option
 end
```

- `alphaNumChar ()`  
returns a random alphanumeric character.
- `alphaNumString n`  
returns a string of length `n` of random alphanumeric characters.
- `rand ()`  
returns the next pseudo-random number.
- `seed ()`  
returns a random word from `/dev/random`. Useful as an arg to `srand`. If `/dev/random` can not be read from, `seed ()` returns `NONE`. A call to `seed` may block until enough random bits are available.
- `srand w`  
sets the seed used by `rand` to `w`.
- `useed ()`  
returns a random word from `/dev/urandom`. Useful as an arg to `srand`. If `/dev/urandom` can not be read from, `useed ()` returns `NONE`. A call to `useed` will never block — it will instead return lower quality random bits.

## 169 MLtonReal

```
signature MLTON_REAL =
 sig
 type t

 val fromWord: word -> t
 val fromLargeWord: LargeWord.word -> t
 val toWord: IEEEReal.rounding_mode -> t -> word
 val toLargeWord: IEEEReal.rounding_mode -> t -> LargeWord.word
 end
```

- type t  
the type of reals. For `MLton.LargeReal` this is `LargeReal.real`, for `MLton.Real` this is `Real.real`, for `MLton.Real32` this is `Real32.real`, for `MLton.Real64` this is `Real64.real`.
- fromWord w
- fromLargeWord w  
convert the word `w` to a real value. If the value of `w` is larger than (the appropriate) `REAL.maxFinite`, then infinity is returned. If `w` cannot be exactly represented as a real value, then the current rounding mode is used to determine the resulting value.
- toWord mode r
- toLargeWord mode r  
convert the argument `r` to a word type using the specified rounding mode. They raise `Overflow` if the result is not representable, in particular, if `r` is an infinity. They raise `Domain` if `r` is NaN.
- `MLton.Real32.castFromWord w`
- `MLton.Real64.castFromWord w`  
convert the argument `w` to a real type as a bit-wise cast.
- `MLton.Real32.castToWord r`
- `MLton.Real64.castToWord r`  
convert the argument `r` to a word type as a bit-wise cast.

## 170 MLtonRlimit

```
signature MLTON_RLIMIT =
 sig
 structure RLim : sig
 type t
 val castFromSysWord: SysWord.word -> t
 val castToSysWord: t -> SysWord.word
 end

 val infinity: RLim.t

 type t

 val coreFileSize: t (* CORE max core file size *)
 val cpuTime: t (* CPU CPU time in seconds *)
 val dataSize: t (* DATA max data size *)
 val fileSize: t (* FSIZE Maximum filesize *)
 val numFiles: t (* NOFILE max number of open files *)
 val lockedInMemorySize: t (* MEMLOCK max locked address space *)
 val numProcesses: t (* NPROC max number of processes *)
 val residentSetSize: t (* RSS max resident set size *)
 val stackSize: t (* STACK max stack size *)
 val virtualMemorySize: t (* AS virtual memory limit *)

 val get: t -> {hard: rlim, soft: rlim}
 val set: t * {hard: rlim, soft: rlim} -> unit
 end
```

MLton.Rlimit provides a wrapper around the C `getrlimit` and `setrlimit` functions.

- `type Rlim.t`  
the type of resource limits.
- `infinity`  
indicates that a resource is unlimited.
- `type t`  
the types of resources that can be inspected and modified.
- `get r`  
returns the current hard and soft limits for resource `r`. May raise `OS.SysErr`.
- `set (r, {hard, soft})`  
sets the hard and soft limits for resource `r`. May raise `OS.SysErr`.

## 171 MLtonRusage

```
signature MLTON_RUSAGE =
 sig
 type t = {utime: Time.time, (* user time *)
 stime: Time.time} (* system time *)

 val measureGC: bool -> unit
 val rusage: unit -> {children: t, gc: t, self: t}
 end
```

- `type t`  
corresponds to a subset of the C `struct rusage`.
- `measureGC b`  
controls whether garbage collection time is separately measured during program execution. This affects the behavior of both `rusage` and `Timer.checkCPUTimes`, both of which will return `gc` times of zero with `measureGC false`. Garbage collection time is always measured when either `gc-messages` or `gc-summary` is given as a [runtime system option](#).
- `rusage ()`  
corresponds to the C `getrusage` function. It returns the resource usage of the exited children, the garbage collector, and the process itself. The `self` component includes the usage of the `gc` component, regardless of whether `measureGC` is `true` or `false`. If `rusage` is used in a program, either directly, or indirectly via the `Timer` structure, then `measureGC true` is automatically called at the start of the program (it can still be disabled by user code later).

## 172 MLtonSignal

```
signature MLTON_SIGNAL =
 sig
 type t = Posix.Signal.signal
 type signal = t

 structure Handler:
 sig
 type t

 val default: t
 val handler: (Thread.Runnable.t -> Thread.Runnable.t) -> t
 val ignore: t
 val isDefault: t -> bool
 val isIgnore: t -> bool
 val simple: (unit -> unit) -> t
 end

 structure Mask:
 sig
 type t

 val all: t
 val allBut: signal list -> t
 val block: t -> unit
 val getBlocked: unit -> t
 val isMember: t * signal -> bool
 val none: t
 val setBlocked: t -> unit
 val some: signal list -> t
 val unblock: t -> unit
 end

 val getHandler: t -> Handler.t
 val handled: unit -> Mask.t
 val prof: t
 val restart: bool ref
 val setHandler: t * Handler.t -> unit
 val suspend: Mask.t -> unit
 val vtalrm: t
 end
```

Signals handlers are functions from (runnable) threads to (runnable) threads. When a signal arrives, the corresponding signal handler is invoked, its argument being the thread that was interrupted by the signal. The signal handler runs asynchronously, in its own thread. The signal handler returns the thread that it would like to resume execution (this is often the thread that it was passed). It is an error for a signal handler to raise an exception that is not handled within the signal handler itself.

A signal handler is never invoked while the running thread is in a critical section (see [MLtonThread](#)). Invoking a signal handler implicitly enters a critical section and the normal return of a signal handler implicitly exits the critical section; hence, a signal handler is never interrupted by another signal handler.

- type t  
the type of signals.
- type Handler.t  
the type of signal handlers.
- Handler.default  
handles the signal with the default action.

- 
- `Handler.handler f`  
returns a handler `h` such that when a signal `s` is handled by `h`, `f` will be passed the thread that was interrupted by `s` and should return the thread that will resume execution.
  - `Handler.ignore`  
is a handler that will ignore the signal.
  - `Handler.isDefault`  
returns true if the handler is the default handler.
  - `Handler.isIgnore`  
returns true if the handler is the ignore handler.
  - `Handler.simple f`  
returns a handler that executes `f ()` and does not switch threads.
  - `type Mask.t`  
the type of signal masks, which are sets of blocked signals.
  - `Mask.all`  
a mask of all signals.
  - `Mask.allBut l`  
a mask of all signals except for those in `l`.
  - `Mask.block m`  
blocks all signals in `m`.
  - `Mask.getBlocked ()`  
gets the signal mask `m`, i.e. a signal is blocked if and only if it is in `m`.
  - `Mask.isMember (m, s)`  
returns true if the signal `s` is in `m`.
  - `Mask.none`  
a mask of no signals.
  - `Mask.setBlocked m`  
sets the signal mask to `m`, i.e. a signal is blocked if and only if it is in `m`.
  - `Mask.some l`  
a mask of the signals in `l`.
  - `Mask.unblock m`  
unblocks all signals in `m`.
  - `getHandler s`  
returns the current handler for signal `s`.
  - `handled ()`  
returns the signal mask `m` corresponding to the currently handled signals; i.e., a signal is handled if and only if it is in `m`.
  - `prof`  
`SIGPROF`, the profiling signal.
  - `restart`  
dynamically determines the behavior of interrupted system calls; when `true`, interrupted system calls are restarted; when `false`, interrupted system calls raise `OS.SysError`.
-

- `setHandler (s, h)`  
sets the handler for signal `s` to `h`.
- `suspend m`  
temporarily sets the signal mask to `m` and suspends until an unmasked signal is received and handled, at which point `suspend` resets the mask and returns.
- `vtalarm`  
`SIGVTALRM`, the signal for virtual timers.

## 172.1 Interruptible System Calls

Signal handling interacts in a non-trivial way with those functions in the [Basis Library](#) that correspond directly to interruptible system calls (a subset of those functions that may raise `OS.SysError`). The desire is that these functions should have predictable semantics. The principal concerns are:

1. System calls that are interrupted by signals should, by default, be restarted; the alternative is to raise

```
OS.SysError (Posix.Error.errorMessage Posix.Error.intr,
 SOME Posix.Error.intr)
```

This behavior is determined dynamically by the value of `Signal.restart`.

2. Signal handlers should always get a chance to run (when outside a critical region). If a system call is interrupted by a signal, then the signal handler will run before the call is restarted or `OS.SysError` is raised; that is, before the `Signal.restart` check.
3. A system call that must be restarted while in a critical section will be restarted with the handled signals blocked (and the previously blocked signals remembered). This encourages the system call to complete, allowing the program to make progress towards leaving the critical section where the signal can be handled. If the system call completes, the set of blocked signals are restored to those previously blocked.



```
structure Word8Vector: MLTON_MONO_VECTOR where type t = Word8Vector.vector
 where type elem = Word8Vector.elem
structure World: MLTON_WORLD
end
```

## 173.1 Substructures

- [MLtonArray](#)
- [MLtonBinIO](#)
- [MLtonCont](#)
- [MLtonExn](#)
- [MLtonFinalizable](#)
- [MLtonGC](#)
- [MLtonIntInf](#)
- [MLtonIO](#)
- [MLtonItimer](#)
- [MLtonMonoArray](#)
- [MLtonMonoVector](#)
- [MLtonPlatform](#)
- [MLtonPointer](#)
- [MLtonProcEnv](#)
- [MLtonProcess](#)
- [MLtonRandom](#)
- [MLtonReal](#)
- [MLtonRlimit](#)
- [MLtonRusage](#)
- [MLtonSignal](#)
- [MLtonSyslog](#)
- [MLtonTextIO](#)
- [MLtonThread](#)
- [MLtonVector](#)
- [MLtonWeak](#)
- [MLtonWord](#)
- [MLtonWorld](#)

## 173.2 Values

- `eq (x, y)`  
returns true if `x` and `y` are equal as pointers. For simple types like `char`, `int`, and `word`, this is the same as `equals`. For arrays, datatypes, strings, tuples, and vectors, this is a simple pointer equality. The semantics is a bit murky.
- `equal (x, y)`  
returns true if `x` and `y` are structurally equal. For equality types, this is the same as [PolymorphicEquality](#). For other types, it is a conservative approximation of equivalence.
- `hash x`  
returns a structural hash of `x`. The hash function is consistent between execution of the same program, but may not be consistent between different programs.
- `isMLton`  
is always `true` in a MLton implementation, and is always `false` in a stub implementation.
- `share x`  
maximizes sharing in the heap for the object graph reachable from `x`.
- `shareAll ()`  
maximizes sharing in the heap by sharing space for equivalent immutable objects. A call to `shareAll` performs a major garbage collection, and takes time proportional to the size of the heap.
- `size x`  
returns the amount of heap space (in bytes) taken by the value of `x`, including all objects reachable from `x` by following pointers. It takes time proportional to the size of `x`. See below for an example.

## 173.3 Example of MLton.size

This example, `size.sml`, demonstrates the application of `MLton.size` to many different kinds of objects.

```
fun 'a printSize (name: string, value: 'a): unit=
 (print "The size of "
 ; print name
 ; print " is "
 ; print (Int.toString (MLton.size value))
 ; print " bytes.\n")

val l = [1, 2, 3, 4]

val _ =
 (
 printSize ("an int list of length 4", l)
 ; printSize ("a string of length 10", "0123456789")
 ; printSize ("an int array of length 10", Array.tabulate (10, fn _ => 0))
 ; printSize ("a double array of length 10",
 Array.tabulate (10, fn _ => 0.0))
 ; printSize ("an array of length 10 of 2-ples of ints",
 Array.tabulate (10, fn i => (i, i + 1)))
 ; printSize ("a useless function", fn _ => 13)
)

(* This is here so that the list is "useful".
 * If it were removed, then the optimizer (remove-unused-constructors)
 * would remove l entirely.
 *)
val _ = if 10 = foldl (op +) 0 l
```

```
 then ()
 else raise Fail "bug"

local
 open MLton.Cont
in
 val rc: int option t option ref = ref NONE
 val _ =
 case callcc (fn k: int option t => (rc := SOME k; throw (k, NONE))) of
 NONE => ()
 | SOME i => print (concat [Int.toString i, "\n"])
 end

 val _ = printSize ("a continuation option ref", rc)

 val _ =
 case !rc of
 NONE => ()
 | SOME k => (rc := NONE; MLton.Cont.throw (k, SOME 13))
```

Compile and run as usual.

```
% mlton size.sml
% ./size
The size of an int list of length 4 is 48 bytes.
The size of a string of length 10 is 24 bytes.
The size of an int array of length 10 is 52 bytes.
The size of a double array of length 10 is 92 bytes.
The size of an array of length 10 of 2-ples of ints is 92 bytes.
The size of a useless function is 0 bytes.
The size of a continuation option ref is 4544 bytes.
13
The size of a continuation option ref is 8 bytes.
```

Note that sizes are dependent upon the target platform and compiler optimizations.

## 174 MLtonSyslog

```
signature MLTON_SYSLOG =
 sig
 type openflag

 val CONS : openflag
 val NDELAY : openflag
 val NOWAIT : openflag
 val ODELAY : openflag
 val PERROR : openflag
 val PID : openflag

 type facility

 val AUTHPRIV : facility
 val CRON : facility
 val DAEMON : facility
 val KERN : facility
 val LOCAL0 : facility
 val LOCAL1 : facility
 val LOCAL2 : facility
 val LOCAL3 : facility
 val LOCAL4 : facility
 val LOCAL5 : facility
 val LOCAL6 : facility
 val LOCAL7 : facility
 val LPR : facility
 val MAIL : facility
 val NEWS : facility
 val SYSLOG : facility
 val USER : facility
 val UUCP : facility

 type loglevel

 val EMERG : loglevel
 val ALERT : loglevel
 val CRIT : loglevel
 val ERR : loglevel
 val WARNING : loglevel
 val NOTICE : loglevel
 val INFO : loglevel
 val DEBUG : loglevel

 val closelog: unit -> unit
 val log: loglevel * string -> unit
 val openlog: string * openflag list * facility -> unit
 end
```

MLton.Syslog is a complete interface to the system logging facilities. See `man 3 syslog` for more details.

- `closelog ()`  
closes the connection to the system logger.
- `log (l, s)`  
logs message `s` at a loglevel `l`.
- `openlog (name, flags, facility)`  
opens a connection to the system logger. `name` will be prefixed to each message, and is typically set to the program name.

## 175 MLtonTextIO

```
signature MLTON_TEXT_IO = MLTON_IO
```

See [MLtonIO](#).

---

## 176 MLtonThread

```
signature MLTON_THREAD =
 sig
 structure AtomicState:
 sig
 datatype t = NonAtomic | Atomic of int
 end

 val atomically: (unit -> 'a) -> 'a
 val atomicBegin: unit -> unit
 val atomicEnd: unit -> unit
 val atomicState: unit -> AtomicState.t

 structure Runnable:
 sig
 type t
 end

 type 'a t

 val atomicSwitch: ('a t -> Runnable.t) -> 'a
 val new: ('a -> unit) -> 'a t
 val prepend: 'a t * ('b -> 'a) -> 'b t
 val prepare: 'a t * 'a -> Runnable.t
 val switch: ('a t -> Runnable.t) -> 'a
 end
 end
```

`MLton.Thread` provides access to MLton's user-level thread implementation (i.e. not OS-level threads). Threads are lightweight data structures that represent a paused computation. Runnable threads are threads that will begin or continue computing when switch-ed to. `MLton.Thread` does not include a default scheduling mechanism, but it can be used to implement both preemptive and non-preemptive threads.

- `type AtomicState.t`  
the type of atomic states.
- `atomically f`  
runs `f` in a critical section.
- `atomicBegin ()`  
begins a critical section.
- `atomicEnd ()`  
ends a critical section.
- `atomicState ()`  
returns the current atomic state.
- `type Runnable.t`  
the type of threads that can be resumed.
- `type 'a t`  
the type of threads that expect a value of type `'a`.
- `atomicSwitch f`  
like `switch`, but assumes an atomic calling context. Upon switch-ing back to the current thread, an implicit `atomicEnd` is performed.

- `new f`  
creates a new thread that, when run, applies `f` to the value given to the thread. `f` must terminate by `switch`ing to another thread or exiting the process.
- `prepend (t, f)`  
creates a new thread (destroying `t` in the process) that first applies `f` to the value given to the thread and then continues with `t`. This is a constant time operation.
- `prepare (t, v)`  
prepares a new runnable thread (destroying `t` in the process) that will evaluate `t` on `v`.
- `switch f`  
applies `f` to the current thread to get `rt`, and then start running thread `rt`. It is an error for `f` to perform another `switch`. `f` is guaranteed to run atomically.

### 176.1 Example of non-preemptive threads

```

structure Queue:
 sig
 type 'a t

 val new: unit -> 'a t
 val enqueue: 'a t * 'a -> unit
 val dequeue: 'a t -> 'a option
 end =
 struct
 datatype 'a t = T of {front: 'a list ref, back: 'a list ref}

 fun new () = T {front = ref [], back = ref []}

 fun enqueue (T {back, ...}, x) = back := x :: !back

 fun dequeue (T {front, back}) =
 case !front of
 [] => (case !back of
 [] => NONE
 | l => let val l = rev l
 in case l of
 [] => raise Fail "dequeue"
 | x :: l => (back := []; front := l; SOME x)
 end)
 | x :: l => (front := l; SOME x)
 end

 end

structure Thread:
 sig
 val exit: unit -> 'a
 val run: unit -> unit
 val spawn: (unit -> unit) -> unit
 val yield: unit -> unit
 end =
 struct
 open MLton
 open Thread

 val topLevel: Thread.Runnable.t option ref = ref NONE

 local
 val threads: Thread.Runnable.t Queue.t = Queue.new ()
 end
 end

```

```

in
 fun ready (t: Thread.Runnable.t) : unit =
 Queue.enqueue(threads, t)
 fun next () : Thread.Runnable.t =
 case Queue.dequeue threads of
 NONE => valOf (!topLevel)
 | SOME t => t
end

fun 'a exit (): 'a = switch (fn _ => next ())

fun new (f: unit -> unit): Thread.Runnable.t =
 Thread.prepare
 (Thread.new (fn () => ((f () handle _ => exit ())
 ; exit ())),
 ())

fun schedule t = (ready t; next ())

fun yield (): unit = switch (fn t => schedule (Thread.prepare (t, ())))

val spawn = ready o new

fun run(): unit =
 (switch (fn t =>
 (topLevel := SOME (Thread.prepare (t, ()))
 ; next()))
 ; topLevel := NONE)
end

val rec loop =
 fn 0 => ()
 | n => (print(concat[Int.toString n, "\n"])
 ; Thread.yield()
 ; loop(n - 1))

val rec loop' =
 fn 0 => ()
 | n => (Thread.spawn (fn () => loop n); loop' (n - 2))

val _ = Thread.spawn (fn () => loop' 10)

val _ = Thread.run ()

val _ = print "success\n"

```

## 176.2 Example of preemptive threads

```

structure Queue:
 sig
 type 'a t

 val new: unit -> 'a t
 val enqueue: 'a t * 'a -> unit
 val dequeue: 'a t -> 'a option
 end =
 struct
 datatype 'a t = T of {front: 'a list ref, back: 'a list ref}

 fun new () = T {front = ref [], back = ref []}
 end

```

```

fun enqueue (T {back, ...}, x) = back := x :: !back

fun dequeue (T {front, back}) =
 case !front of
 [] => (case !back of
 [] => NONE
 | l => let val l = rev l
 in case l of
 [] => raise Fail "dequeue"
 | x :: l => (back := []; front := l; SOME x)
 end)
 | x :: l => (front := l; SOME x)
 end

structure Thread:
sig
 val exit: unit -> 'a
 val run: unit -> unit
 val spawn: (unit -> unit) -> unit
 val yield: unit -> unit
end =
struct
 open Posix.Signal
 open MLton
 open Itimer Signal Thread

 val topLevel: Thread.Runnable.t option ref = ref NONE

 local
 val threads: Thread.Runnable.t Queue.t = Queue.new ()
 in
 fun ready (t: Thread.Runnable.t) : unit =
 Queue.enqueue(threads, t)
 fun next () : Thread.Runnable.t =
 case Queue.dequeue threads of
 NONE => valOf (!topLevel)
 | SOME t => t
 end

 fun 'a exit (): 'a = switch (fn _ => next ())

 fun new (f: unit -> unit): Thread.Runnable.t =
 Thread.prepare
 (Thread.new (fn () => ((f () handle _ => exit ())
 ; exit ())),
 ())

 fun schedule t = (ready t; next ())

 fun yield (): unit = switch (fn t => schedule (Thread.prepare (t, ())))

 val spawn = ready o new

 fun setItimer t =
 Itimer.set (Itimer.Real,
 {value = t,
 interval = t})

 fun run (): unit =
 (switch (fn t =>
 (topLevel := SOME (Thread.prepare (t, ())))

```

```
 ; new (fn () => (setHandler (alarm, Handler.handler schedule)
 ; setItimer (Time.fromMilliseconds 20))))
 ; setItimer Time.zeroTime
 ; ignore alarm
 ; topLevel := NONE)
end

val rec delay =
 fn 0 => ()
 | n => delay (n - 1)

val rec loop =
 fn 0 => ()
 | n => (delay 500000; loop (n - 1))

val rec loop' =
 fn 0 => ()
 | n => (Thread.spawn (fn () => loop n); loop' (n - 1))

val _ = Thread.spawn (fn () => loop' 10)

val _ = Thread.run ()

val _ = print "success\n"
```

## 177 MLtonVector

```
signature MLTON_VECTOR =
 sig
 val create: int -> {done: unit -> 'a vector,
 sub: int -> 'a,
 update: int * 'a -> unit}
 val unfoldi: int * 'b * (int * 'b -> 'a * 'b) -> 'a vector * 'b
 end
```

- `create n`

initiates the construction a vector  $v$  of length  $n$ , returning functions to manipulate the vector. The `done` function may be called to return the created vector; it is an error to call `done` before all entries have been initialized; it is an error to call `done` after having called `done`. The `sub` function may be called to return an initialized vector entry; it is not an error to call `sub` after having called `done`. The `update` function may be called to initialize a vector entry; it is an error to call `update` after having called `done`. One must initialize vector entries in order from lowest to highest; that is, before calling `update (i, x)`, one must have already called `update (j, x)` for all  $j$  in  $[0, i)$ . The `done`, `sub`, and `update` functions are all constant-time operations.

- `unfoldi (n, b, f)`

constructs a vector  $v$  of length  $n$ , whose elements  $v_i$  are determined by the equations  $v_0 = b$  and  $(v_i, b_{i+1}) = f(i, b_i)$ .

## 178 MLtonWeak

```
signature MLTON_WEAK =
 sig
 type 'a t

 val get: 'a t -> 'a option
 val new: 'a -> 'a t
 end
```

A weak pointer is a pointer to an object that is nulled if the object becomes [unreachable](#) due to garbage collection. The weak pointer does not itself cause the object it points to be retained by the garbage collector—only other strong pointers can do that. For objects that are not allocated in the heap, like integers, a weak pointer will always be nulled. So, if `w: int Weak.t`, then `Weak.get w = NONE`.

- `type 'a t`  
the type of weak pointers to objects of type 'a
- `get w`  
returns `NONE` if the object pointed to by `w` no longer exists. Otherwise, returns `SOME` of the object pointed to by `w`.
- `new x`  
returns a weak pointer to `x`.

## 179 MLtonWord

```
signature MLTON_WORD =
 sig
 type t

 val bswap: t -> t
 val rol: t * word -> t
 val ror: t * word -> t
 end
```

- type t

the type of words. For `MLton.LargeWord` this is `LargeWord.word`, for `MLton.Word` this is `Word.word`, for `MLton.Word8` this is `Word8.word`, for `MLton.Word16` this is `Word16.word`, for `MLton.Word32` this is `Word32.word`, for `MLton.Word64` this is `Word64.word`.

- bswap w  
byte swap.
- rol (w, w')  
rotates left (circular).
- ror (w, w')  
rotates right (circular).

## 180 MLtonWorld

```
signature MLTON_WORLD =
 sig
 datatype status = Clone | Original

 val load: string -> 'a
 val save: string -> status
 val saveThread: string * Thread.Runnable.t -> unit
 end
```

- `datatype status`  
specifies whether a world is original or restarted (a clone).
- `load f`  
loads the saved computation from file `f`.
- `save f`  
saves the entire state of the computation to the file `f`. The computation can then be restarted at a later time using `World.load` or the `load-world` [runtime option](#). The call to `save` in the original computation returns `Original` and the call in the restarted world returns `Clone`.
- `saveThread (f, rt)`  
saves the entire state of the computation to the file `f` that will resume with thread `rt` upon restart.

### 180.1 Notes

Executables that save and load worlds are incompatible with [address space layout randomization \(ASLR\)](#) of the executable (though, not of shared libraries). The state of a computation includes addresses into the code and data segments of the executable (e.g., static runtime-system data, return addresses); such addresses are invalid when interpreted by the executable loaded at a different base address.

Executables that save and load worlds should be compiled with an option to suppress the generation of position-independent executables.

- [Darwin 11 \(Mac OS X Lion\) and higher](#): `-link-opt -fno-PIE`

### 180.2 Example

Suppose that `save-world.sml` contains the following.

```
open MLton.World

val _ =
 case save "world" of
 Original => print "I am the original\n"
 | Clone => print "I am the clone\n"
```

Then, if we compile `save-world.sml` and run it, the `Original` branch will execute, and a file named `world` will be created.

```
% mlton save-world.sml
% ./save-world
I am the original
```

We can then load `world` using the `load-world` [run time option](#).

```
% ./save-world @MLton load-world world --
I am the clone
```

## 181 MLULex

**MLULex** is a scanner generator for [Standard ML](#).

### 181.1 Also see

- [MLAntlr](#)
  - [MLLPTLibrary](#)
  - [OwensEtAl09](#)
-

## 182 MLYacc

[MLYacc](#) is a parser generator for [Standard ML](#) modeled after the Yacc parser generator.

A version of MLYacc, ported from the [SML/NJ](#) sources, is distributed with MLton.

### 182.1 Also see

- [mlyacc.pdf](#)
  - [MLLex](#)
  - [TarditiAppel00](#)
  - [Price09](#)
-

## 183 Monomorphise

[Monomorphise](#) is a translation pass from the [XML IntermediateLanguage](#) to the [SXML IntermediateLanguage](#).

### 183.1 Description

Monomorphisation eliminates polymorphic values and datatype declarations by duplicating them for each type at which they are used.

Consider the following [XML](#) program.

```
datatype 'a t = T of 'a
fun 'a f (x: 'a) = T x
val a = f 1
val b = f 2
val z = f (3, 4)
```

The result of monomorphising this program is the following [SXML](#) program:

```
datatype t1 = T1 of int
datatype t2 = T2 of int * int
fun f1 (x: int) = T1 x
fun f2 (x: int * int) = T2 x
val a = f1 1
val b = f1 2
val z = f2 (3, 4)
```

### 183.2 Implementation

- [monomorphise.sig](#)
- [monomorphise.fun](#)

### 183.3 Details and Notes

The monomorphiser works by making one pass over the entire program. On the way down, it creates a cache for each variable declared in a polymorphic declaration that maps a lists of type arguments to a new variable name. At a variable reference, it consults the cache (based on the types the variable is applied to). If there is already an entry in the cache, it is used. If not, a new entry is created. On the way up, the monomorphiser duplicates a variable declaration for each entry in the cache.

As with variables, the monomorphiser records all of the type at which constructors are used. After the entire program is processed, the monomorphiser duplicates each datatype declaration and its associated constructors.

The monomorphiser duplicates all of the functions declared in a `fun` declaration as a unit. Consider the following program

```
fun 'a f (x: 'a) = g x
and g (y: 'a) = f y
val a = f 13
val b = g 14
val c = f (1, 2)
```

and its monomorphisation

```
fun f1 (x: int) = g1 x
and g1 (y: int) = f1 y
fun f2 (x : int * int) = g2 x
and g2 (y : int * int) = f2 y
val a = f1 13
val b = g1 14
val c = f2 (1, 2)
```

### 183.4 Pathological datatype declarations

SML allows a pathological polymorphic datatype declaration in which recursive uses of the defined type constructor are applied to different type arguments than the definition. This has been disallowed by others on type theoretic grounds. A canonical example is the following.

```
datatype 'a t = A of 'a | B of ('a * 'a) t
val z : int t = B (B (A ((1, 2), (3, 4))))
```

The presence of the recursion in the datatype declaration might appear to cause the need for the monomorphiser to create an infinite number of types. However, due to the absence of polymorphic recursion in SML, there are in fact only a finite number of instances of such types in any given program. The monomorphiser translates the above program to the following one.

```
datatype t1 = B1 of t2
datatype t2 = B2 of t3
datatype t3 = A3 of (int * int) * (int * int)
val z : int t = B1 (B2 (A3 ((1, 2), (3, 4))))
```

It is crucial that the monomorphiser be allowed to drop unused constructors from datatype declarations in order for the translation to terminate.

## 184 MoscowML

Moscow ML is a [Standard ML implementation](#). It is a byte-code compiler, so it compiles code quickly, but the code runs slowly. See [Performance](#).

---

## 185 Multi

[Multi](#) is an analysis pass for the [SSA IntermediateLanguage](#), invoked from [ConstantPropagation](#) and [LocalRef](#).

### 185.1 Description

This pass analyzes the control flow of a [SSA](#) program to determine which [SSA](#) functions and blocks might be executed more than once or by more than one thread. It also determines when a program uses threads and when functions and blocks directly or indirectly invoke `Thread_copyCurrent`.

### 185.2 Implementation

- `multi.sig`
- `multi.fun`

### 185.3 Details and Notes

---

## 186 Mutable

Mutable is an adjective meaning "can be modified". In [Standard ML](#), ref cells and arrays are mutable, while all other values are [immutable](#).

---

## 187 NeedsReview

This page documents some patches and bug fixes that need additional review by experienced developers:

- Bug in transparent signature match:
    - What is an *original* interface and why does the equivalence of original interfaces implies the equivalence of the actual interfaces?
    - <http://www.mlton.org/pipermail/mlton/2007-September/029991.html>
    - <http://www.mlton.org/pipermail/mlton/2007-September/029995.html>
    - SVN Revision: [r6046](#)
  - Bug in [DeepFlatten](#) pass:
    - Should we allow argument to `Weak_new` to be flattened?
    - SVN Revision: [r6189](#) (regression test demonstrating bug)
    - SVN Revision: [r6191](#)
-

## 188 NumericLiteral

Numeric literals in [Standard ML](#) can be written in either decimal or hexadecimal notation. Sometimes it can be convenient to write numbers down in other bases. Fortunately, using [Fold](#), it is possible to define a concise syntax for numeric literals that allows one to write numeric constants in any base and of various types (`int`, `IntInf.int`, `word`, and more).

We will define constants `I`, `II`, `W`, and ``` so that, for example,

```
I 10 `1`2`3 $
```

denotes `123:int` in base 10, while

```
II 8 `2`3 $
```

denotes `19:IntInf.int` in base 8, and

```
W 2 `1`1`0`1 $
```

denotes `0w13: word`.

Here is the code.

```
structure Num =
 struct
 fun make (op *, op +, i2x) iBase =
 let
 val xBase = i2x iBase
 in
 Fold.fold
 ((i2x 0,
 fn (i, x) =>
 if 0 <= i andalso i < iBase then
 x * xBase + i2x i
 else
 raise Fail (concat
 ["Num: ", Int.toString i,
 " is not a valid\
 \ digit in base ",
 Int.toString iBase])),
 fst)
 end

 fun I ? = make (op *, op +, id) ?
 fun II ? = make (op *, op +, IntInf.fromInt) ?
 fun W ? = make (op *, op +, Word.fromInt) ?

 fun ` ? = Fold.step1 (fn (i, (x, step)) =>
 (step (i, x), step)) ?

 val a = 10
 val b = 11
 val c = 12
 val d = 13
 val e = 14
 val f = 15
 end
```

where

```
fun fst (x, _) = x
```

The idea is for the fold to start with zero and to construct the result one digit at a time, with each stepper multiplying the previous result by the base and adding the next digit. The code is abstracted in two different ways for extra generality. First, the `make` function abstracts over the various primitive operations (addition, multiplication, etc) that are needed to construct a number. This allows the same code to be shared for constants `I`, `II`, `W` used to write down the various numeric types. It also allows users to add new constants for additional numeric types, by supplying the necessary arguments to `make`.

Second, the step function, ```, is abstracted over the actual construction operation, which is created by `make`, and passed along the fold. This allows the same constant, ```, to be used for all numeric types. The alternative approach, having a different step function for each numeric type, would be more painful to use.

On the surface, it appears that the code checks the digits dynamically to ensure they are valid for the base. However, MLton will simplify everything away at compile time, leaving just the final numeric constant.

---

## 189 ObjectOrientedProgramming

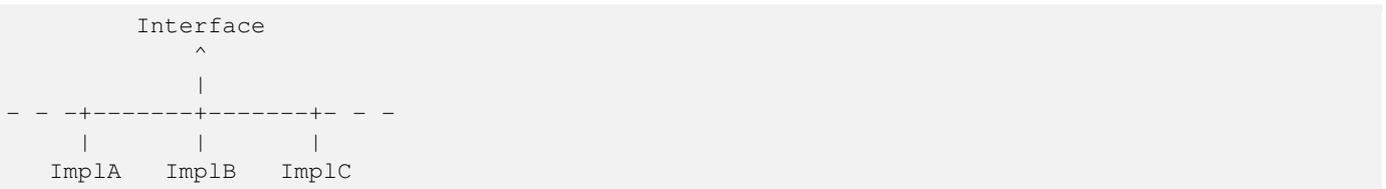
Standard ML does not have explicit support for object-oriented programming. Here are some papers that show how to express certain object-oriented concepts in SML.

- [OO Programming styles in ML](#)
- [Object-oriented programming and Standard ML](#)
- [mGTK: An SML binding of Gtk+](#)
- [Phantom Types and Subtyping](#)

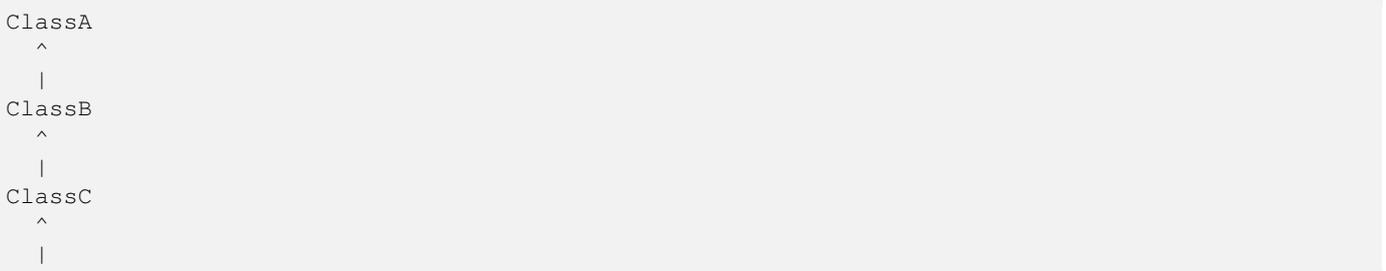
The question of OO programming in SML comes up every now and then. The following discusses a simple object-oriented (OO) programming technique in Standard ML. The reader is assumed to be able to read Java and SML code.

### 189.1 Motivation

SML doesn't provide subtyping, but it does provide parametric polymorphism, which can be used to encode some forms of subtyping. Most articles on OO programming in SML concentrate on such encoding techniques. While those techniques are interesting — and it is recommended to read such articles — and sometimes useful, it seems that basically all OO gurus agree that (deep) subtyping (or inheritance) hierarchies aren't as practical as they were thought to be in the early OO days. "Good", flexible, "OO" designs tend to have a flat structure



and deep inheritance hierarchies



tend to be signs of design mistakes. There are good underlying reasons for this, but a thorough discussion is not in the scope of this article. However, the point is that perhaps the encoding of subtyping is not as important as one might believe. In the following we ignore subtyping and rather concentrate on a very simple and basic dynamic dispatch technique.

### 189.2 Dynamic Dispatch Using a Recursive Record of Functions

Quite simply, the basic idea is to implement a "virtual function table" using a record that is wrapped inside a (possibly recursive) datatype. Let's first take a look at a simple concrete example.

Consider the following Java interface:

```

public interface Counter {
 public void inc();
 public int get();
}

```

We can translate the `Counter` interface to SML as follows:

```
datatype counter = Counter of {inc : unit -> unit, get : unit -> int}
```

Each value of type `counter` can be thought of as an object that responds to two messages `inc` and `get`. To actually send messages to a counter, it is useful to define auxiliary functions

```
local
 fun mk m (Counter t) = m t ()
in
 val cGet = mk#get
 val cInc = mk#inc
end
```

that basically extract the "function table" `t` from a counter object and then select the specified method `m` from the table.

Let's then implement a simple function that increments a counter until a given maximum is reached:

```
fun incUpto counter max = while cGet counter < max do cInc counter
```

You can easily verify that the above code compiles even without any concrete implementation of a counter, thus it is clear that it doesn't depend on a particular counter implementation.

Let's then implement a couple of counters. First consider the following Java class implementing the `Counter` interface given earlier.

```
public class BasicCounter implements Counter {
 private int cnt;
 public BasicCounter(int initialCnt) { this.cnt = initialCnt; }
 public void inc() { this.cnt += 1; }
 public int get() { return this.cnt; }
}
```

We can translate the above to SML as follows:

```
fun newBasicCounter initialCnt = let
 val cnt = ref initialCnt
in
 Counter {inc = fn () => cnt := !cnt + 1,
 get = fn () => !cnt}
end
```

The SML function `newBasicCounter` can be described as a constructor function for counter objects of the `BasicCounter` "class". We can also have other counter implementations. Here is the constructor for a counter decorator that logs messages:

```
fun newLoggedCounter counter =
 Counter {inc = fn () => (print "inc\n" ; cInc counter),
 get = fn () => (print "get\n" ; cGet counter)}
```

The `incUpto` function works just as well with objects of either class:

```
val aCounter = newBasicCounter 0
val () = incUpto aCounter 5
val () = print (Int.toString (cGet aCounter) ^ "\n")

val aCounter = newLoggedCounter (newBasicCounter 0)
val () = incUpto aCounter 5
val () = print (Int.toString (cGet aCounter) ^ "\n")
```

In general, a dynamic dispatch interface is represented as a record type wrapped inside a datatype. Each field of the record corresponds to a public method or field of the object:

```
datatype interface =
 Interface of {method : t1 -> t2,
 immutableField : t,
 mutableField : t ref}
```

The reason for wrapping the record inside a datatype is that records, in SML, can not be recursive. However, SML datatypes can be recursive. A record wrapped in a datatype can contain fields that contain the datatype. For example, an interface such as `Cloneable`

```
datatype cloneable = Cloneable of {clone : unit -> cloneable}
```

can be represented using recursive datatypes.

Like in OO languages, interfaces are abstract and can not be instantiated to produce objects. To be able to instantiate objects, the constructors of a concrete class are needed. In SML, we can implement constructors as simple functions from arbitrary arguments to values of the interface type. Such a constructor function can encapsulate arbitrary private state and functions using lexical closure. It is also easy to share implementations of methods between two or more constructors.

While the `Counter` example is rather trivial, it should not be difficult to see that this technique quite simply doesn't require a huge amount of extra verbiage and is more than usable in practice.

### 189.3 SML Modules and Dynamic Dispatch

One might wonder about how SML modules and the dynamic dispatch technique work together. Let's investigate! Let's use a simple dispenser framework as a concrete example. (Note that this isn't intended to be an introduction to the SML module system.)

#### 189.3.1 Programming with SML Modules

Using SML signatures we can specify abstract data types (ADTs) such as dispensers. Here is a signature for an "abstract" functional (as opposed to imperative) dispenser:

```
signature ABSTRACT_DISPENSER = sig
 type 'a t
 val isEmpty : 'a t -> bool
 val push : 'a * 'a t -> 'a t
 val pop : 'a t -> ('a * 'a t) option
end
```

The term "abstract" in the name of the signature refers to the fact that the signature gives no way to instantiate a dispenser. It has nothing to do with the concept of abstract data types.

Using SML functors we can write "generic" algorithms that manipulate dispensers of an unknown type. Here are a couple of very simple algorithms:

```
functor DispenserAlgs (D : ABSTRACT_DISPENSER) = struct
 open D

 fun pushAll (xs, d) = foldl push d xs

 fun popAll d = let
 fun lp (xs, NONE) = rev xs
 | lp (xs, SOME (x, d)) = lp (x::xs, pop d)
 in
 lp ([], pop d)
 end

 fun cp (from, to) = pushAll (popAll from, to)
end
```

As one can easily verify, the above compiles even without any concrete dispenser structure. Functors essentially provide a form a static dispatch that one can use to break compile-time dependencies.

We can also give a signature for a concrete dispenser

```
signature DISPENSER = sig
 include ABSTRACT_DISPENSER
 val empty : 'a t
end
```

and write any number of concrete structures implementing the signature. For example, we could implement stacks

```
structure Stack :> DISPENSER = struct
 type 'a t = 'a list
 val empty = []
 val isEmpty = null
 val push = op ::
 val pop = List.getItem
end
```

and queues

```
structure Queue :> DISPENSER = struct
 datatype 'a t = T of 'a list * 'a list
 val empty = T ([], [])
 val isEmpty = fn T ([], _) => true | _ => false
 val normalize = fn ([], ys) => (rev ys, []) | q => q
 fun push (y, T (xs, ys)) = T (normalize (xs, y::ys))
 val pop = fn (T (x::xs, ys)) => SOME (x, T (normalize (xs, ys))) | _ => NONE
end
```

One can now write code that uses either the `Stack` or the `Queue` dispenser. One can also instantiate the previously defined functor to create functions for manipulating dispensers of a type:

```
structure S = DispenserAlgs (Stack)
val [4,3,2,1] = S.popAll (S.pushAll ([1,2,3,4], Stack.empty))

structure Q = DispenserAlgs (Queue)
val [1,2,3,4] = Q.popAll (Q.pushAll ([1,2,3,4], Queue.empty))
```

There is no dynamic dispatch involved at the module level in SML. An attempt to do dynamic dispatch

```
val q = Q.push (1, Stack.empty)
```

will give a type error.

### 189.3.2 Combining SML Modules and Dynamic Dispatch

Let's then combine SML modules and the dynamic dispatch technique introduced in this article. First we define an interface for dispensers:

```
structure Dispenser = struct
 datatype 'a t =
 I of {isEmpty : unit -> bool,
 push : 'a -> 'a t,
 pop : unit -> ('a * 'a t) option}

 fun O m (I t) = m t

 fun isEmpty t = O#isEmpty t ()
 fun push (v, t) = O#push t v
 fun pop t = O#pop t ()
end
```

The `Dispenser` module, which we can think of as an interface for dispensers, implements the `ABSTRACT_DISPENSER` signature using the dynamic dispatch technique, but we leave the signature ascription until later.

Then we define a `DispenserClass` functor that makes a "class" out of a given dispenser module:

```
functor DispenserClass (D : DISPENSER) : DISPENSER = struct
 open Dispenser

 fun make d =
 I {isEmpty = fn () => D.isEmpty d,
 push = fn x => make (D.push (x, d)),
 pop = fn () =>
 case D.pop d of
 NONE => NONE
 | SOME (x, d) => SOME (x, make d)}

 val empty =
 I {isEmpty = fn () => true,
 push = fn x => make (D.push (x, D.empty)),
 pop = fn () => NONE}
end
```

Finally we seal the `Dispenser` module:

```
structure Dispenser : ABSTRACT_DISPENSER = Dispenser
```

This isn't necessary for type safety, because the unsealed `Dispenser` module does not allow one to break encapsulation, but makes sure that only the `DispenserClass` functor can create dispenser classes (because the constructor `Dispenser.I` is no longer accessible).

Using the `DispenserClass` functor we can turn any concrete dispenser module into a dispenser class:

```
structure StackClass = DispenserClass (Stack)
structure QueueClass = DispenserClass (Queue)
```

Each dispenser class implements the same dynamic dispatch interface and the `ABSTRACT_DISPENSER`-signature.

Because the dynamic dispatch `Dispenser` module implements the `ABSTRACT_DISPENSER`-signature, we can use it to instantiate the `DispenserAlgs`-functor:

```
structure D = DispenserAlgs (Dispenser)
```

The resulting `D` module, like the `Dispenser` module, works with any dispenser class and uses dynamic dispatch:

```
val [4, 3, 2, 1] = D.popAll (D.pushAll ([1, 2, 3, 4], StackClass.empty))
val [1, 2, 3, 4] = D.popAll (D.pushAll ([1, 2, 3, 4], QueueClass.empty))
```

## 190 OCaml

OCaml is a variant of ML and is similar to Standard ML.

### 190.1 OCaml and SML

Here's a comparison of some aspects of the OCaml and SML languages.

- Standard ML has a formal [Definition](#), while OCaml is specified by its lone implementation and informal documentation.
- Standard ML has a number of [compilers](#), while OCaml has only one.
- OCaml has built-in support for object-oriented programming, while Standard ML does not (however, see [ObjectOrientedProgramming](#)).
- Andreas Rossberg has a [side-by-side comparison](#) of the syntax of SML and OCaml.

### 190.2 OCaml and MLton

Here's a comparison of some aspects of OCaml and MLton.

- Performance
    - Both OCaml and MLton have excellent performance.
    - MLton performs extensive [WholeProgramOptimization](#), which can provide substantial improvements in large, modular programs.
    - MLton uses native types, like 32-bit integers, without any penalty due to tagging or boxing. OCaml uses 31-bit integers with a penalty due to tagging, and 32-bit integers with a penalty due to boxing.
    - MLton uses native types, like 64-bit floats, without any penalty due to boxing. OCaml, in some situations, boxes 64-bit floats.
    - MLton represents arrays of all types unboxed. In OCaml, only arrays of 64-bit floats are unboxed, and then only when it is syntactically apparent.
    - MLton represents records compactly by reordering and packing the fields.
    - In MLton, polymorphic and monomorphic code have the same performance. In OCaml, polymorphism can introduce a performance penalty.
    - In MLton, module boundaries have no impact on performance. In OCaml, moving code between modules can cause a performance penalty.
    - MLton's [ForeignFunctionInterface](#) is simpler than OCaml's.
  - Tools
    - OCaml has a debugger, while MLton does not.
    - OCaml supports separate compilation, while MLton does not.
    - OCaml compiles faster than MLton.
    - MLton supports profiling of both time and allocation.
  - Libraries
    - OCaml has more available libraries.
  - Community
    - OCaml has a larger community than MLton.
    - MLton has a very responsive [developer list](#).
-

## 191 OpenGL

There are at least two interfaces to OpenGL for MLton/SML, both of which should be considered alpha quality.

- [MikeThomas](#) built a low-level interface, directly translating many of the functions, covering GL, GLU, and GLUT. This is available in the MLton [Sources](#): [opengl](#). The code contains a number of small, standard OpenGL examples translated to SML.
- [ChrisClearwater](#) has written at least an interface to GL, and possibly more. See
  - <http://mlton.org/pipermail/mlton/2005-January/026669.html>

[Contact](#) us for more information or an update on the status of these projects.

---

## 192 OperatorPrecedence

Standard ML has a built in notion of precedence for certain symbols. Every program that includes the [Basis Library](#) automatically gets the following infix declarations. Higher number indicates higher precedence.

```
infix 7 * / mod div
infix 6 + - ^
infixr 5 :: @
infix 4 = <> > >= < <=
infix 3 := o
infix 0 before
```

## 193 OptionalArguments

Standard ML does not have built-in support for optional arguments. Nevertheless, using [Fold](#), it is easy to define functions that take optional arguments.

For example, suppose that we have the following definition of a function  $f$ .

```
fun f (i, r, s) =
 concat [Int.toString i, ", ", Real.toString r, ", ", s]
```

Using the `OptionalArg` structure described below, we can define a function  $f'$ , an optionalized version of  $f$ , that takes 0, 1, 2, or 3 arguments. Embedded within  $f'$  will be default values for  $i$ ,  $r$ , and  $s$ . If  $f'$  gets no arguments, then all the defaults are used. If  $f'$  gets one argument, then that will be used for  $i$ . Two arguments will be used for  $i$  and  $r$  respectively. Three arguments will override all default values. Calls to  $f'$  will look like the following.

```
f' $
f' `2 $
f' `2 `3.0 $
f' `2 `3.0 `"four" $
```

The optional argument indicator, ```, is not special syntax --- it is a normal SML value, defined in the `OptionalArg` structure below.

Here is the definition of  $f'$  using the `OptionalArg` structure, in particular, `OptionalArg.make` and `OptionalArg.D`.

```
val f' =
 fn z =>
 let open OptionalArg in
 make (D 1) (D 2.0) (D "three") $
 end (fn i & r & s => f (i, r, s))
 z
```

The definition of  $f'$  is eta expanded as with all uses of `fold`. A call to `OptionalArg.make` is supplied with a variable number of defaults (in this case, three), the end-of-arguments terminator, `$`, and the function to run, taking its arguments as an  $n$ -ary [product](#). In this case, the function simply converts the product to an ordinary tuple and calls  $f$ . Often, the function body will simply be written directly.

In general, the definition of an optional-argument function looks like the following.

```
val f =
 fn z =>
 let open OptionalArg in
 make (D <default1>) (D <default2>) ... (D <defaultn>) $
 end (fn x1 & x2 & ... & xn =>
 <function code goes here>)
 z
```

Here is the definition of `OptionalArg`.

```
structure OptionalArg =
 struct
 val make =
 fn z =>
 Fold.fold
 ((id, fn (f, x) => f x),
 fn (d, r) => fn func =>
 Fold.fold ((id, d ()), fn (f, d) =>
 let
 val d & () = r (id, f d)
 in
 func d
 end))
```

```

z

fun D d = Fold.step0 (fn (f, r) =>
 (fn ds => f (d & ds),
 fn (f, a & b) => r (fn x => f a & x, b)))

val ` =
 fn z =>
 Fold.step1 (fn (x, (f, _ & d)) => (fn d => f (x & d), d))
z
end

```

`OptionalArg.make` uses a nested fold. The first fold accumulates the default values in a product, associated to the right, and a reversal function that converts a product (of the same arity as the number of defaults) from right associativity to left associativity. The accumulated defaults are used by the second fold, which recurs over the product, replacing the appropriate component as it encounters optional arguments. The second fold also constructs a "fill" function, `f`, that is used to reconstruct the product once the end-of-arguments is reached. Finally, the finisher reconstructs the product and uses the reversal function to convert the product from right associative to left associative, at which point it is passed to the user-supplied function.

Much of the complexity comes from the fact that while recurring over a product from left to right, one wants it to be right-associative, e.g., look like

```
a & (b & (c & d))
```

but the user function in the end wants the product to be left associative, so that the product argument pattern can be written without parentheses (since `&` is left associative).

### 193.1 Labelled optional arguments

In addition to the positional optional arguments described above, it is sometimes useful to have labelled optional arguments. These allow one to define a function, `f`, with defaults, say `a` and `b`. Then, a caller of `f` can supply values for `a` and `b` by name. If no value is supplied then the default is used.

Labelled optional arguments are a simple extension of [FunctionalRecordUpdate](#) using post composition. Suppose, for example, that one wants a function `f` with labelled optional arguments `a` and `b` with default values `0` and `0.0` respectively. If one has a functional-record-update function `updateAB` for records with `a` and `b` fields, then one can define `f` in the following way.

```

val f =
 fn z =>
 Fold.post
 (updateAB {a = 0, b = 0.0},
 fn {a, b} => print (concat [Int.toString a, " ",
 Real.toString b, "\n"]))
z

```

The idea is that `f` is the post composition (using `Fold.post`) of the actual code for the function with a functional-record updater that starts with the defaults.

Here are some example calls to `f`.

```

val () = f $
val () = f (U#a 13) $
val () = f (U#a 13) (U#b 17.5) $
val () = f (U#b 17.5) (U#a 13) $

```

Notice that a caller can supply neither of the arguments, either of the arguments, or both of the arguments, and in either order. All that matter is that the arguments be labelled correctly (and of the right type, of course).

Here is another example.

```
val f =
 fn z =>
 Fold.post
 (updateBCD {b = 0, c = 0.0, d = "<>"},
 fn {b, c, d} =>
 print (concat [Int.toString b, " ",
 Real.toString c, " ",
 d, "\n"]))
z
```

Here are some example calls.

```
val () = f $
val () = f (U#d "goodbye") $
val () = f (U#d "hello") (U#b 17) (U#c 19.3) $
```

## 194 OtherSites

Other sites that have a MLton page (or more).

- [Advogato](#)
  - [Debian GNU/Linux \(developer\)](#)
  - [FreeBSD](#)
  - [freshmeat](#)
  - [freshports](#)
  - [GNU](#)
  - [icewalkers](#)
  - [Ubuntu](#)
  - [wikipedia](#)
-

## 195 Overloading

In [Standard ML](#), constants (like `13`, `0w13`, `13.0`) are overloaded, meaning that they can denote a constant of the appropriate type as determined by context. SML defines the overloading classes *Int*, *Real*, and *Word*, which denote the sets of types that integer, real, and word constants may take on. In MLton, these are defined as follows.

|             |                                                                                                                          |
|-------------|--------------------------------------------------------------------------------------------------------------------------|
| <i>Int</i>  | <code>Int2.int, Int3.int, ... Int32.int, Int64.int, Int.int, IntInf.int, LargeInt.int, FixedInt.int, Position.int</code> |
| <i>Real</i> | <code>Real32.real, Real64.real, Real.real, LargeReal.real</code>                                                         |
| <i>Word</i> | <code>Word2.word, Word3.word, ... Word32.word, Word64.word, Word.word, LargeWord.word, SysWord.word</code>               |

The [Definition](#) allows flexibility in how much context is used to resolve overloading. It says that the context is *no larger than the smallest enclosing structure-level declaration*, but that *an implementation may require that a smaller context determines the type*. MLton uses the largest possible context allowed by SML in resolving overloading. If the type of a constant is not determined by context, then it takes on a default type. In MLton, these are defined as follows.

|             |                        |
|-------------|------------------------|
| <i>Int</i>  | <code>Int.int</code>   |
| <i>Real</i> | <code>Real.real</code> |
| <i>Word</i> | <code>Word.word</code> |

Other implementations may use a smaller context or different default types.

### 195.1 Also see

- [discussion of overloading in the Basis Library](#)

### 195.2 Examples

- The following program is rejected.

```
structure S:
 sig
 val x: Word8.word
 end =
 struct
 val x = 0w0
 end
```

The smallest enclosing structure declaration for `0w0` is `val x = 0w0`. Hence, `0w0` receives the default type for words, which is `Word.word`.

## 196 PackedRepresentation

[PackedRepresentation](#) is an analysis pass for the [SSA2 IntermediateLanguage](#), invoked from [ToRSSA](#).

### 196.1 Description

This pass analyzes a [SSA2](#) program to compute a packed representation for each object.

### 196.2 Implementation

- [representation.sig](#)
- [packed-representation.fun](#)

### 196.3 Details and Notes

Has a special case to make sure that `true` is represented as 1 and `false` is represented as 0.

---

## 197 ParallelMove

[ParallelMove](#) is a rewrite pass, agnostic in the [IntermediateLanguage](#) which it produces.

### 197.1 Description

This function computes a sequence of individual moves to effect a parallel move (with possibly overlapping froms and tos).

### 197.2 Implementation

- [parallel-move.sig](#)
- [parallel-move.fun](#)

### 197.3 Details and Notes

---

## 198 Performance

This page compares the performance of a number of SML compilers on a range of benchmarks. For a [performance comparison](#) of many different languages, including MLton, see the [Computer Language Shootout](#).

This page compares the following SML compiler versions.

- [MLton 20051202](#)
- [ML Kit 4.1.4](#)
- [Moscow ML 2.00](#)
- [Poly/ML 4.1.3](#)
- [SML/NJ 110.57](#)

There are tables for [run time](#), [code size](#), and [compile time](#).

### 198.1 Setup

All benchmarks were compiled and run on a 2.6 GHz Celeron with 2G of RAM. The benchmarks were compiled with the default settings for all the compilers, except for Moscow ML, which was passed the `-orthodox -standalone -toplevel` switches. The Poly/ML executables were produced by `use-ing` the file, followed by a `PolyML.commit`. The SML/NJ executables were produced by wrapping the entire program in a `local` declaration whose body performs an `SMLofNJ.exportFn`.

For more details, or if you want to run the benchmarks yourself, please see the [benchmark](#) directory of our [Sources](#).

All of the benchmarks are available for download from this page. Some of the benchmarks were obtained from the SML/NJ benchmark suite. Some of the benchmarks expect certain input files to exist in the [DATA](#) subdirectory.

- `hamlet.sml hamlet-input.sml`
- `ray.sml ray`
- `raytrace.sml chess.gml`
- `vliw.sml ndotprod.s`

### 198.2 Run-time ratio

The following table gives the ratio of the run time of each benchmark when compiled by another compiler to the run time when compiled by MLton. That is, the larger the number, the slower the generated code runs. A number larger than one indicates that the corresponding compiler produces code that runs more slowly than MLton. A \* in an entry means the compiler failed to compile the benchmark or that the benchmark failed to run.

| benchmark                      | MLton | ML-Kit | MosML | Poly/ML | SML/NJ |
|--------------------------------|-------|--------|-------|---------|--------|
| <code>barnes-hut.sml</code>    | 1.0   | *      | *     | *       | 1.6    |
| <code>boyer.sml</code>         | 1.0   | *      | 10.1  | 1.9     | 3.1    |
| <code>checksum.sml</code>      | 1.0   | *      | *     | *       | *      |
| <code>count-graphs.sml</code>  | 1.0   | 7.3    | 60.7  | 4.2     | 3.8    |
| <code>DLX Simulator.sml</code> | 1.0   | *      | *     | *       | *      |
| <code>fft.sml</code>           | 1.0   | 1.2    | *     | 24.2    | 0.8    |
| <code>fib.sml</code>           | 1.0   | 0.9    | 5.0   | 1.2     | 1.3    |
| <code>flat-array.sml</code>    | 1.0   | 2.2    | 35.0  | 1041.6  | 13.4   |
| <code>hamlet.sml</code>        | 1.0   | *      | *     | *       | 3.1    |
| <code>imp-for.sml</code>       | 1.0   | 2.8    | 63.0  | 5.1     | 5.6    |
| <code>knuth-bendix.sml</code>  | 1.0   | *      | 19.8  | 4.8     | 4.6    |

| benchmark             | MLton | ML-Kit | MosML | Poly/ML | SML/NJ |
|-----------------------|-------|--------|-------|---------|--------|
| lexgen.sml            | 1.0   | 2.5    | 5.0   | 1.7     | 1.5    |
| life.sml              | 1.0   | 1.7    | 30.6  | 7.7     | 1.4    |
| logic.sml             | 1.0   | *      | 9.4   | 1.2     | 2.1    |
| mandelbrot.sml        | 1.0   | 4.2    | 34.0  | 51.1    | 1.3    |
| matrix-multiply.sml   | 1.0   | 8.3    | 42.5  | 13.2    | 5.3    |
| md5.sml               | 1.0   | *      | *     | *       | *      |
| merge.sml             | 1.0   | *      | *     | 1.1     | 7.9    |
| mlyacc.sml            | 1.0   | 1.5    | 8.2   | 1.2     | 2.2    |
| model-elimination.sml | 1.0   | *      | *     | *       | 2.6    |
| mpuz.sml              | 1.0   | 2.3    | 78.2  | 4.6     | 4.1    |
| nucleic.sml           | 1.0   | *      | *     | 23.5    | 0.8    |
| output1.sml           | 1.0   | 30.7   | 61.4  | 16.2    | 14.4   |
| peek.sml              | 1.0   | 15.2   | 176.9 | 17.9    | 11.3   |
| psdes-random.sml      | 1.0   | 5.0    | *     | *       | 2.7    |
| ratio-regions.sml     | 1.0   | 2.0    | 34.7  | 2.1     | 5.4    |
| ray.sml               | 1.0   | *      | 14.8  | 22.3    | 0.8    |
| raytrace.sml          | 1.0   | *      | *     | *       | 3.3    |
| simple.sml            | 1.0   | 1.7    | 19.3  | 7.3     | 2.4    |
| smith-normal-form.sml | 1.0   | *      | *     | *       | >1000  |
| tailfib.sml           | 1.0   | 1.0    | 51.9  | 3.2     | 1.4    |
| tak.sml               | 1.0   | 1.2    | 17.0  | 1.3     | 2.0    |
| tensor.sml            | 1.0   | *      | *     | *       | 7.4    |
| tsp.sml               | 1.0   | 3.4    | 31.8  | *       | 17.7   |
| tyan.sml              | 1.0   | *      | 15.7  | 1.0     | 1.6    |
| vector-concat.sml     | 1.0   | 1.2    | 20.4  | 2.0     | 20.4   |
| vector-rev.sml        | 1.0   | 2.2    | 41.9  | 2.3     | 152.4  |
| vliw.sml              | 1.0   | *      | *     | *       | 2.5    |
| wc-input1.sml         | 1.0   | 11.1   | *     | 7.5     | 17.2   |
| wc-scanStream.sml     | 1.0   | 22.1   | *     | 203.7   | 11.5   |
| zebra.sml             | 1.0   | 3.9    | 30.2  | 3.4     | 8.5    |
| zern.sml              | 1.0   | *      | *     | *       | 2.6    |

Note: for SML/NJ, the `smith-normal-form.sml` benchmark was killed after running for over 25,000 seconds.

### 198.3 Code size

The following table gives the code size of each benchmark in bytes. The size for MLton and the ML Kit is the sum of text and data for the standalone executable as reported by `size`. The size for Moscow ML is the size in bytes of the executable `a.out`. The size for Poly/ML is the difference in size of the database before the session start and after the commit. The size for SML/NJ is the size of the heap file created by `exportFn` and does not include the size of the SML/NJ runtime system (approximately 100K). A \* in an entry means that the compiler failed to compile the benchmark.

| benchmark        | MLton     | ML-Kit  | MosML   | Poly/ML | SML/NJ    |
|------------------|-----------|---------|---------|---------|-----------|
| barnes-hut.sml   | 103,231   | *       | *       | *       | 433,216   |
| boyer.sml        | 138,518   | 163,204 | 116,300 | 122,880 | 526,376   |
| checksum.sml     | 52,794    | *       | *       | *       | *         |
| count-graphs.sml | 66,838    | 84,124  | 84,613  | 98,304  | 454,776   |
| DLXSimulator.sml | 129,398   | *       | *       | *       | *         |
| fft.sml          | 64,797    | 80,240  | 84,046  | 65,536  | 434,256   |
| fib.sml          | 47,738    | 18,588  | 79,892  | 49,152  | 415,488   |
| flat-array.sml   | 47,762    | 23,820  | 80,034  | 49,152  | 410,680   |
| hamlet.sml       | 1,256,813 | *       | *       | *       | 1,412,360 |
| imp-for.sml      | 47,626    | 19,372  | 80,040  | 57,344  | 400,424   |
| knuth-bendix.sml | 109,126   | 93,400  | 88,439  | 180,224 | 431,144   |

| benchmark             | MLton   | ML-Kit  | MosML   | Poly/ML   | SML/NJ    |
|-----------------------|---------|---------|---------|-----------|-----------|
| lexgen.sml            | 203,559 | 208,332 | 104,883 | 196,608   | 501,824   |
| life.sml              | 66,130  | 78,084  | 83,390  | 65,536    | 414,760   |
| logic.sml             | 106,614 | 116,880 | 87,251  | 114,688   | 440,360   |
| mandelbrot.sml        | 47,690  | 77,004  | 81,340  | 57,344    | 404,520   |
| matrix-multiply.sml   | 49,181  | 87,016  | 82,417  | 57,344    | 435,256   |
| md5.sml               | 77,646  | *       | *       | *         | *         |
| merge.sml             | 49,318  | 24,296  | 80,090  | 49,152    | 400,432   |
| mlyacc.sml            | 507,431 | 473,748 | 148,286 | 2,850,816 | 820,336   |
| model-elimination.sml | 638,084 | *       | *       | *         | 1,009,880 |
| mpuz.sml              | 50,594  | 73,232  | 82,382  | 81,920    | 408,616   |
| nucleic.sml           | 199,181 | 258,552 | *       | 221,184   | 487,480   |
| output1.sml           | 80,720  | 63,336  | 80,187  | 49,152    | 399,400   |
| peek.sml              | 76,302  | 62,092  | 81,621  | 57,344    | 403,544   |
| psdes-random.sml      | 48,402  | 25,196  | *       | *         | 421,944   |
| ratio-regions.sml     | 73,914  | 95,924  | 87,482  | 73,728    | 443,448   |
| ray.sml               | 183,243 | 108,848 | 89,859  | 147,456   | 493,712   |
| raytrace.sml          | 265,332 | *       | *       | *         | 636,112   |
| simple.sml            | 222,914 | 192,032 | 94,396  | 475,136   | 756,840   |
| smith-normal-form.sml | 181,686 | *       | *       | 131,072   | 558,224   |
| tailfib.sml           | 47,434  | 18,804  | 79,943  | 57,344    | 399,400   |
| tak.sml               | 47,818  | 18,580  | 79,908  | 57,344    | 411,392   |
| tensor.sml            | 97,677  | *       | *       | *         | 450,672   |
| tsp.sml               | 82,190  | 97,716  | 86,146  | *         | 425,024   |
| tyan.sml              | 134,910 | 137,800 | 91,586  | 196,608   | 477,272   |
| vector-concat.sml     | 49,018  | 23,924  | 80,194  | 49,152    | 410,680   |
| vector-rev.sml        | 48,246  | 24,104  | 80,078  | 57,344    | 410,680   |
| vliw.sml              | 393,762 | *       | *       | *         | 731,304   |
| wc-input1.sml         | 101,850 | 129,212 | 85,771  | 49,152    | 404,520   |
| wc-scanStream.sml     | 109,106 | 129,708 | 85,947  | 49,152    | 405,544   |
| zebra.sml             | 141,146 | 41,532  | 83,422  | 90,112    | 419,896   |
| zern.sml              | 91,087  | *       | *       | *         | 479,384   |

## 198.4 Compile time

The following table gives the compile time of each benchmark in seconds. A \* in an entry means that the compiler failed to compile the benchmark.

| benchmark           | MLton | ML-Kit | MosML | Poly/ML | SML/NJ |
|---------------------|-------|--------|-------|---------|--------|
| barnes-hut.sml      | 8.28  | *      | *     | *       | 1.37   |
| boyer.sml           | 8.14  | 8.99   | 0.39  | 0.12    | 3.20   |
| checksum.sml        | 5.45  | *      | *     | *       | *      |
| count-graphs.sml    | 6.12  | 2.06   | 0.14  | 0.05    | 0.90   |
| DLX Simulator.sml   | 9.81  | *      | *     | *       | *      |
| fft.sml             | 5.95  | 1.32   | 0.11  | 0.05    | 0.69   |
| fib.sml             | 5.45  | 0.60   | 0.05  | 0.02    | 0.22   |
| flat-array.sml      | 5.33  | 0.61   | 0.04  | 0.01    | 0.25   |
| hamlet.sml          | 85.70 | *      | *     | *       | 88.87  |
| imp-for.sml         | 5.37  | 0.73   | 0.05  | 0.01    | 0.25   |
| knuth-bendix.sml    | 7.09  | 4.11   | 0.19  | 0.12    | 1.60   |
| lexgen.sml          | 11.02 | 7.21   | 0.40  | 0.26    | 3.63   |
| life.sml            | 5.84  | 2.16   | 0.10  | 0.04    | 0.64   |
| logic.sml           | 7.02  | 4.82   | 0.22  | 0.09    | 1.68   |
| mandelbrot.sml      | 5.41  | 0.75   | 0.06  | 0.02    | 0.29   |
| matrix-multiply.sml | 5.39  | 0.77   | 0.06  | 0.01    | 0.30   |
| md5.sml             | 6.01  | *      | *     | *       | *      |

| <b>benchmark</b>      | <b>MLton</b> | <b>ML-Kit</b> | <b>MosML</b> | <b>Poly/ML</b> | <b>SML/NJ</b> |
|-----------------------|--------------|---------------|--------------|----------------|---------------|
| merge.sml             | 5.41         | 0.62          | 0.06         | 0.02           | 0.26          |
| mlyacc.sml            | 24.70        | 40.69         | 3.35         | 1.08           | 18.04         |
| model-elimination.sml | 25.04        | *             | *            | *              | 28.79         |
| mpuz.sml              | 5.41         | 1.07          | 0.07         | 0.03           | 0.45          |
| nucleic.sml           | 14.24        | 24.79         | *            | 0.36           | 2.78          |
| output1.sml           | 6.05         | 0.68          | 0.05         | 0.01           | 0.23          |
| peek.sml              | 6.04         | 0.70          | 0.05         | 0.02           | 0.25          |
| psdes-random.sml      | 5.39         | 0.75          | *            | *              | 64.13         |
| ratio-regions.sml     | 6.63         | 4.02          | 0.21         | 0.11           | 1.50          |
| ray.sml               | 9.51         | 3.02          | 0.15         | 0.08           | 1.03          |
| raytrace.sml          | 13.92        | *             | *            | *              | 5.08          |
| simple.sml            | 11.40        | 13.19         | 0.43         | 0.21           | 3.76          |
| smith-normal-form.sml | 8.90         | *             | *            | 0.10           | 2.25          |
| tailfib.sml           | 5.35         | 0.64          | 0.05         | 0.02           | 0.24          |
| tak.sml               | 5.36         | 0.62          | 0.05         | 0.01           | 0.22          |
| tensor.sml            | 8.75         | *             | *            | *              | 2.81          |
| tsp.sml               | 6.50         | 1.93          | 0.15         | *              | 0.66          |
| tyan.sml              | 8.86         | 6.25          | 0.30         | 0.17           | 2.28          |
| vector-concat.sml     | 5.52         | 0.68          | 0.05         | 0.01           | 0.25          |
| vector-rev.sml        | 5.33         | 0.64          | 0.05         | 0.02           | 0.26          |
| vliw.sml              | 18.28        | *             | *            | *              | 13.12         |
| wc-input1.sml         | 6.85         | 0.68          | 0.07         | 0.02           | 0.27          |
| wc-scanStream.sml     | 7.07         | 0.69          | 0.06         | 0.02           | 0.29          |
| zebra.sml             | 8.57         | 2.30          | 0.09         | 0.04           | 0.78          |
| zern.sml              | 6.20         | *             | *            | *              | 0.65          |

## 199 PhantomType

A phantom type is a type that has no run-time representation, but is used to force the type checker to ensure invariants at compile time. This is done by augmenting a type with additional arguments (phantom type variables) and expressing constraints by choosing phantom types to stand for the phantom types in the types of values.

### 199.1 Also see

- [Blume01](#)
  - dimensions
  - C type system
- [FluetPucella02](#)
  - subtyping
- socket module in [Basis Library](#)

## 200 PlatformSpecificNotes

Here are notes about using MLton on the following platforms.

### 200.1 Operating Systems

- [AIX](#)
- [Cygwin](#)
- [Darwin](#)
- [FreeBSD](#)
- [HPUX](#)
- [Linux](#)
- [MinGW](#)
- [NetBSD](#)
- [OpenBSD](#)
- [Solaris](#)

### 200.2 Architectures

- [AMD64](#)
- [HPPA](#)
- [PowerPC](#)
- [PowerPC64](#)
- [Sparc](#)
- [X86](#)

### 200.3 Also see

- [PortingMLton](#)
-

## 201 PolyEqual

[PolyEqual](#) is an optimization pass for the [SSA IntermediateLanguage](#), invoked from [SSASimplify](#).

### 201.1 Description

This pass implements polymorphic equality.

### 201.2 Implementation

- [poly-equal.fun](#)

### 201.3 Details and Notes

For each datatype, tycon, and vector type, it builds an equality function and translates calls to `MLton_equal` into calls to that function.

Also generates calls to `IntInf_equal` and `Word_equal`.

For tuples, it does the equality test inline; i.e., it does not create a separate equality function for each tuple type.

All equality functions are created only if necessary, i.e., if equality is actually used at a type.

Optimizations:

- for datatypes that are enumerations, do not build a case dispatch, just use `MLton_eq`, as the backend will represent these as ints
- deep equality always does an `MLton_eq` test first
- If one argument to `=` is a constant and the type will get translated to an `IntOrPointer`, then just use `eq` instead of the full equality. This is important for implementing code like the following efficiently:

```
if x = 0 ... (where x is of type IntInf.int)
```

- Also convert pointer equality on scalar types to type specific primitives.

## 202 PolyHash

[PolyHash](#) is an optimization pass for the [SSA IntermediateLanguage](#), invoked from [SSASimplify](#).

### 202.1 Description

This pass implements polymorphic, structural hashing.

### 202.2 Implementation

- `poly-hash.fun`

### 202.3 Details and Notes

For each datatype, tycon, and vector type, it builds an equality function and translates calls to `MLton_hash` into calls to that function.

For tuples, it does the equality test inline; i.e., it does not create a separate equality function for each tuple type.

All equality functions are created only if necessary, i.e., if equality is actually used at a type.

---

## 203 PolyML

Poly/ML is a [Standard ML implementation](#).

### 203.1 Also see

- [Matthews95](#)
-

## 204 PolymorphicEquality

Polymorphic equality is a built-in function in [Standard ML](#) that compares two values of the same type for equality. It is specified as

```
val = : 'a * 'a -> bool
```

The `'a` in the specification are [equality type variables](#), and indicate that polymorphic equality can only be applied to values of an [equality type](#). It is not allowed in SML to rebind `=`, so a programmer is guaranteed that `=` always denotes polymorphic equality.

### 204.1 Equality of ground types

Ground types like `char`, `int`, and `word` may be compared (to values of the same type). For example, `13 = 14` is type correct and yields `false`.

### 204.2 Equality of reals

The one ground type that can not be compared is `real`. So, `13.0 = 14.0` is not type correct. One can use `Real.==` to compare reals for equality, but beware that this has different algebraic properties than polymorphic equality.

See <http://standardml.org/Basis/real.html> for a discussion of why `real` is not an equality type.

### 204.3 Equality of functions

Comparison of functions is not allowed.

### 204.4 Equality of immutable types

Polymorphic equality can be used on [immutable](#) values like tuples, records, lists, and vectors. For example,

```
(1, 2, 3) = (4, 5, 6)
```

is a type-correct expression yielding `false`, while

```
[1, 2, 3] = [1, 2, 3]
```

is type correct and yields `true`.

Equality on immutable values is computed by structure, which means that values are compared by recursively descending the data structure until ground types are reached, at which point the ground types are compared with primitive equality tests (like comparison of characters). So, the expression

```
[1, 2, 3] = [1, 1 + 1, 1 + 1 + 1]
```

is guaranteed to yield `true`, even though the lists may occupy different locations in memory.

Because of structural equality, immutable values can only be compared if their components can be compared. For example, `[1, 2, 3]` can be compared, but `[1.0, 2.0, 3.0]` can not. The SML type system uses [equality types](#) to ensure that structural equality is only applied to valid values.

## 204.5 Equality of mutable values

In contrast to immutable values, polymorphic equality of **mutable** values (like ref cells and arrays) is performed by pointer comparison, not by structure. So, the expression

```
ref 13 = ref 13
```

is guaranteed to yield `false`, even though the ref cells hold the same contents.

Because equality of mutable values is not structural, arrays and refs can be compared *even if their components are not equality types*. Hence, the following expression is type correct (and yields true).

```
let
 val r = ref 13.0
in
 r = r
end
```

## 204.6 Equality of datatypes

Polymorphic equality of datatypes is structural. Two values of the same datatype are equal if they are of the same **variant** and if the **variant**'s arguments are equal (recursively). So, with the datatype

```
datatype t = A | B of t
```

then `B (B A) = B A` is type correct and yields `false`, while `A = A` and `B A = B A` yield `true`.

As polymorphic equality descends two values to compare them, it uses pointer equality whenever it reaches a mutable value. So, with the datatype

```
datatype t = A of int ref | ...
```

then `A (ref 13) = A (ref 13)` is type correct and yields `false`, because the pointer equality on the two ref cells yields `false`.

One weakness of the SML type system is that datatypes do not inherit the special property of the `ref` and `array` type constructors that allows them to be compared regardless of their component type. For example, after declaring

```
datatype 'a t = A of 'a ref
```

one might expect to be able to compare two values of type `real t`, because pointer comparison on a ref cell would suffice. Unfortunately, the type system can only express that a user-defined datatype **admits equality** or not. In this case, `t` admits equality, which means that `int t` can be compared but that `real t` can not. We can confirm this with the program

```
datatype 'a t = A of 'a ref
fun f (x: real t, y: real t) = x = y
```

on which MLton reports the following error.

```
Error: z.sml 2.34.
 Function applied to incorrect argument.
 expects: [<equality>] * [<equality>]
 but got: [<non-equality>] * [<non-equality>]
 in: = (x, y)
```

## 204.7 Implementation

Polymorphic equality is implemented by recursively descending the two values being compared, stopping as soon as they are determined to be unequal, or exploring the entire values to determine that they are equal. Hence, polymorphic equality can take time proportional to the size of the smaller value.

MLton uses some optimizations to improve performance.

- When computing structural equality, first do a pointer comparison. If the comparison yields `true`, then stop and return `true`, since the structural comparison is guaranteed to do so. If the pointer comparison fails, then recursively descend the values.
- If a datatype is an enum (e.g. `datatype t = A | B | C`), then a single comparison suffices to compare values of the datatype. No case dispatch is required to determine whether the two values are of the same [variant](#).
- When comparing a known constant non-value-carrying [variant](#), use a single comparison. For example, the following code will compile into a single comparison for `A = x`.

```
datatype t = A | B | C of ...
fun f x = ... if A = x then ...
```

- When comparing a small constant `IntInf.int` to another `IntInf.int`, use a single comparison against the constant. No case dispatch is required.

## 204.8 Also see

- [AdmitsEquality](#)
- [EqualityType](#)
- [EqualityTypeVariable](#)

## 205 Polyvariance

Polyvariance is an optimization pass for the [SXML IntermediateLanguage](#), invoked from [SXMLSimplify](#).

### 205.1 Description

This pass duplicates a higher-order, `let` bound function at each variable reference, if the cost is smaller than some threshold.

### 205.2 Implementation

- `polyvariance.fun`

### 205.3 Details and Notes

---

## 206 Poplog

**POPLOG** is a development environment that includes implementations of a number of languages, including [Standard ML](#).

While POPLOG is actively developed, the [ML](#) support predates [SML'97](#), and there is no support for the [Basis Library specification](#).

### 206.1 Also see

- [Mixed-language programming in ML and Pop-11](#).
-

## 207 Porting MLton

Porting MLton to a new target platform (architecture or OS) involves the following steps.

1. Make the necessary changes to the scripts, runtime system, [Basis Library](#) implementation, and compiler.
2. Get the regressions working using a cross compiler.
3. [Cross compile](#) MLton and bootstrap on the target.

MLton has a native code generator only for AMD64 and X86, so, if you are porting to another architecture, you must use the C code generator. These notes do not cover building a new native code generator.

Some of the following steps will not be necessary if MLton already supports the architecture or operating system you are porting to.

### 207.1 What code to change

- Scripts.

- In `bin/platform`, add new cases to define `$HOST_OS` and `$HOST_ARCH`.

- Runtime system.

The goal of this step is to be able to successfully run `make` in the `runtime` directory on the target machine.

- In `platform.h`, add a new case to include `platform/<arch>.h` and `platform/<os>.h`.
- In `platform/<arch>.h`:
  - \* `define MLton_Platform_Arch_host`.
- In `platform/<os>.h`:
  - \* include platform-specific includes.
  - \* `define MLton_Platform_OS_host`.
  - \* define all of the `HAS_*` macros.
- In `platform/<os>.c` implement any platform-dependent functions that the runtime needs.
- Add rounding mode control to `basis/Real/IEEEReal.c` for the new arch (if not `HAS_FEROUND`)
- Compile and install the [GnuMP](#). This varies from platform to platform. In `platform/<os>.h`, you need to include the appropriate `gmp.h`.

- Basis Library implementation (`basis-library/*`)

- In `primitive/prim-mlton.sml`:
  - \* Add a new variant to the `MLton.Platform.Arch.t` datatype.
  - \* modify the constants that define `MLton.Platform.Arch.host` to match with `MLton_Platform_Arch_host`, as set in `runtime/platform/<arch>.h`.
  - \* Add a new variant to the `MLton.Platform.OS.t` datatype.
  - \* modify the constants that define `MLton.Platform.OS.host` to match with `MLton_Platform_OS_host`, as set in `runtime/platform/<os>.h`.
- In `mlton/platform.{sig,sml}` add a new variant.
- In `sml-nj/sml-nj.sml`, modify `getOSKind`.
- Look at all the uses of `MLton.Platform` in the Basis Library implementation and see if you need to do anything special. You might use the following command to see where to look.

```
find basis-library -type f | xargs grep 'MLton\.Platform'
```

If in doubt, leave the code alone and wait to see what happens when you run the regression tests.

- Compiler.

- In `lib/stubs/mlton-stubs/platform.sig` add any new variants, as was done in the Basis Library.
- In `lib/stubs/mlton-stubs/mlton.sml` add any new variants in `MLton.Platform`, as was done in the Basis Library.

The string used to identify a particular architecture or operating system must be the same (except for possibly case of letters) in the scripts, runtime, Basis Library implementation, and compiler (stubs). In `mlton/main/main.fun`, MLton itself uses the conversions to and from strings:

```
MLton.Platform.{Arch,OS}.{from,to}String
```

If there is a mismatch, you may see the error message `strange arch` or `strange os`.

## 207.2 Running the regressions with a cross compiler

When porting to a new platform, it is always best to get all (or as many as possible) of the regressions working before moving to a self compile. It is easiest to do this by modifying and rebuilding the compiler on a working machine and then running the regressions with a cross compiler. It is not easy to build a gcc cross compiler, so we recommend generating the C and assembly on a working machine (using MLton's `-target` and `-stop g` flags, copying the generated files to the target machine, then compiling and linking there.

1. Remake the compiler on a working machine.
2. Use `bin/add-cross` to add support for the new target. In particular, this should create `build/lib/targets/<target>/` with the platform-specific necessary cross-compilation information.
3. Run the regression tests with the cross-compiler. To cross-compile all the tests, do

```
bin/regression -cross <target>
```

This will create all the executables. Then, copy `bin/regression` and the `regression` directory to the target machine, and do

```
bin/regression -run-only <target>
```

This should run all the tests.

Repeat this step, interleaved with appropriate compiler modifications, until all the regressions pass.

## 207.3 Bootstrap

Once you've got all the regressions working, you can build MLton for the new target. As with the regressions, the idea for bootstrapping is to generate the C and assembly on a working machine, copy it to the target machine, and then compile and link there. Here's the sequence of steps.

1. On a working machine, with the newly rebuilt compiler, in the `mlton` directory, do:

```
mlton -stop g -target <target> mlton.mlb
```

2. Copy to the target machine.
3. On the target machine, move the libraries to the right place. That is, in `build/lib/targets`, do:

```
rm -rf self
mv <target> self
```

4. On the target machine, compile and link MLton. That is, in the mlton directory, do something like:

```
gcc -c -Ibuild/lib/include -Ibuild/lib/targets/self/include -O1 -w mlton/mlton.*.[cs]
gcc -o build/lib/mlton-compile \
 -Ibuild/lib/targets/self \
 -L/usr/local/lib \
 mlton.*.o \
 -lmlton -lgmp -lgdtoa -lm
```

5. At this point, MLton should be working and you can finish the rest of a usual make on the target machine.

```
make basis-no-check script mlbpathmap targetmap constants libraries tools
```

There are other details to get right, like making sure that the tools directories were clean so that the tools are rebuilt on the new platform, but hopefully this structure works. Once you've got a compiler on the target machine, you should test it by running all the regressions normally (i.e. without the `-cross` flag) and by running a couple rounds of self compiles.

## 207.4 Also see

The above description is based on the following emails sent to the MLton list.

- <http://www.mlton.org/pipermail/mlton/2002-October/013110.html>
- <http://www.mlton.org/pipermail/mlton/2004-July/016029.html>

## 208 PrecedenceParse

[PrecedenceParse](#) is an analysis/rewrite pass for the [AST IntermediateLanguage](#), invoked from [Elaborate](#).

### 208.1 Description

This pass rewrites [AST](#) function clauses, expressions, and patterns to resolve [OperatorPrecedence](#).

### 208.2 Implementation

- [precedence-parse.sig](#)
- [precedence-parse.fun](#)

### 208.3 Details and Notes

---

## 209 Printf

Programmers coming from C or Java often ask if [Standard ML](#) has a `printf` function. It does not. However, it is possible to implement your own version with only a few lines of code.

Here is a definition for `printf` and `fprintf`, along with format specifiers for booleans, integers, and reals.

```
structure Printf =
 struct
 fun $ (_, f) = f (fn p => p ()) ignore
 fun fprintf out f = f (out, id)
 val printf = fn z => fprintf TextIO.stdout z
 fun one ((out, f), make) g =
 g (out, fn r =>
 f (fn p =>
 make (fn s =>
 r (fn () => (p (); TextIO.output (out, s))))))
 fun ` x s = one (x, fn f => f s)
 fun spec to x = one (x, fn f => f o to)
 val B = fn z => spec Bool.toString z
 val I = fn z => spec Int.toString z
 val R = fn z => spec Real.toString z
 end
```

Here's an example use.

```
val () = printf ``Int="I`` Bool="B`` Real="R``\n" $ 1 false 2.0
```

This prints the following.

```
Int=1 Bool=false Real=2.0
```

In general, a use of `printf` looks like

```
printf <spec1> ... <specn> $ <arg1> ... <argm>
```

where each `<speci>` is either a specifier like `B`, `I`, or `R`, or is an inline string, like ``"foo"`. A backtick (```) must precede each inline string. Each `<argi>` must be of the appropriate type for the corresponding specifier.

SML `printf` is more powerful than its C counterpart in a number of ways. In particular, the function produced by `printf` is a perfectly ordinary SML function, and can be passed around, used multiple times, etc. For example:

```
val f: int -> bool -> unit = printf ``Int="I`` Bool="B``\n" $
val () = f 1 true
val () = f 2 false
```

The definition of `printf` is even careful to not print anything until it is fully applied. So, examples like the following will work as expected.

```
val f: int -> bool -> unit = printf ``Int="I`` Bool="B``\n" $ 13
val () = f true
val () = f false
```

It is also easy to define new format specifiers. For example, suppose we wanted format specifiers for characters and strings.

```
val C = fn z => spec Char.toString z
val S = fn z => spec (fn s => s) z
```

One can define format specifiers for more complex types, e.g. pairs of integers.

```

val I2 =
 fn z =>
 spec (fn (i, j) =>
 concat ["(", Int.toString i, ", ", Int.toString j, ")"])
 z

```

Here's an example use.

```

val () = printf `Test "I2` a string "S`\n" $ (1, 2) "hello"

```

## 209.1 Printf via Fold

`printf` is best viewed as a special case of variable-argument [Fold](#) that inductively builds a function as it processes its arguments. Here is the definition of a `Printf` structure in terms of fold. The structure is equivalent to the above one, except that it uses the standard `$` instead of a specialized one.

```

structure Printf =
 struct
 fun fprintf out =
 Fold.fold ((out, id), fn (_, f) => f (fn p => p ()) ignore)

 val printf = fn z => fprintf TextIO.stdout z

 fun one ((out, f), make) =
 (out, fn r =>
 f (fn p =>
 make (fn s =>
 r (fn () => (p (); TextIO.output (out, s))))))

 val ` =
 fn z => Fold.step1 (fn (s, x) => one (x, fn f => f s)) z

 fun spec to = Fold.step0 (fn x => one (x, fn f => f o to))

 val B = fn z => spec Bool.toString z
 val I = fn z => spec Int.toString z
 val R = fn z => spec Real.toString z
 end

```

Viewing `printf` as a fold opens up a number of possibilities. For example, one can name parts of format strings using the fold idiom for naming sequences of steps.

```

val IB = fn u => Fold.fold u `Int="I" Bool="B
val () = printf IB` "IB`\n" $ 1 true 3 false

```

One can even parametrize over partial format strings.

```

fun XB X = fn u => Fold.fold u `X="X" Bool="B
val () = printf (XB I)` "(XB R)`\n" $ 1 true 2.0 false

```

## 209.2 Also see

- [PrintfGentle](#)
- [Functional Unparsing](#)

## 210 PrintfGentle

This page provides a gentle introduction and derivation of [Printf](#), with sections and arrangement more suitable to a talk.

### 210.1 Introduction

SML does not have `printf`. Could we define it ourselves?

```
val () = printf ("here's an int %d and a real %f.\n", 13, 17.0)
val () = printf ("here's three values (%d, %f, %f).\n", 13, 17.0, 19.0)
```

What could the type of `printf` be?

This obviously can't work, because SML functions take a fixed number of arguments. Actually they take one argument, but if that's a tuple, it can only have a fixed number of components.

### 210.2 From tupling to currying

What about currying to get around the typing problem?

```
val () = printf "here's an int %d and a real %f.\n" 13 17.0
val () = printf "here's three values (%d, %f, %f).\n" 13 17.0 19.0
```

That fails for a similar reason. We need two types for `printf`.

```
val printf: string -> int -> real -> unit
val printf: string -> int -> real -> real -> unit
```

This can't work, because `printf` can only have one type. SML doesn't support programmer-defined overloading.

### 210.3 Overloading and dependent types

Even without worrying about number of arguments, there is another problem. The type of `printf` depends on the format string.

```
val () = printf "here's an int %d and a real %f.\n" 13 17.0
val () = printf "here's a real %f and an int %d.\n" 17.0 13
```

Now we need

```
val printf: string -> int -> real -> unit
val printf: string -> real -> int -> unit
```

Again, this can't possibly working because SML doesn't have overloading, and types can't depend on values.

### 210.4 Idea: express type information in the format string

If we express type information in the format string, then different uses of `printf` can have different types.

```
type 'a t (* the type of format strings *)
val printf: 'a t -> 'a
infix D F
val fs1: (int -> real -> unit) t = "here's an int \"D\" and a real \"F\".\n"
val fs2: (int -> real -> real -> unit) t =
 "here's three values (\"D\", \"F\", \"F\")\n"
val () = printf fs1 13 17.0
val () = printf fs2 13 17.0 19.0
```

Now, our two calls to `printf` type check, because the format string specializes `printf` to the appropriate type.

## 210.5 The types of format characters

What should the type of format characters `D` and `F` be? Each format character requires an additional argument of the appropriate type to be supplied to `printf`.

Idea: guess the final type that will be needed for `printf` the format string and verify it with each format character.

```
type ('a, 'b) t (* 'a = rest of type to verify, 'b = final type *)
val ` : string -> ('a, 'a) t (* guess the type, which must be verified *)
val D: (int -> 'a, 'b) t * string -> ('a, 'b) t (* consume an int *)
val F: (real -> 'a, 'b) t * string -> ('a, 'b) t (* consume a real *)
val printf: (unit, 'a) t -> 'a
```

Don't worry. In the end, type inference will guess and verify for us.

## 210.6 Understanding guess and verify

Now, let's build up a format string and a specialized `printf`.

```
infix D F
val f0 = `"here's an int "`
val f1 = f0 D " and a real "
val f2 = f1 F ".\n"
val p = printf f2
```

These definitions yield the following types.

```
val f0: (int -> real -> unit, int -> real -> unit) t
val f1: (real -> unit, int -> real -> unit) t
val f2: (unit, int -> real -> unit) t
val p: int -> real -> unit
```

So, `p` is a specialized `printf` function. We could use it as follows

```
val () = p 13 17.0
val () = p 14 19.0
```

## 210.7 Type checking this using a functor

```
signature PRINTF =
 sig
 type ('a, 'b) t
 val ` : string -> ('a, 'a) t
 val D: (int -> 'a, 'b) t * string -> ('a, 'b) t
 val F: (real -> 'a, 'b) t * string -> ('a, 'b) t
 val printf: (unit, 'a) t -> 'a
 end

functor Test (P: PRINTF) =
 struct
 open P
 infix D F

 val () = printf (`"here's an int "D" and a real "F".\n") 13 17.0
 val () = printf (`"here's three values ("D", "F ", "F").\n") 13 17.0 19.0
 end
```

## 210.8 Implementing Printf

Think of a format character as a formatter transformer. It takes the formatter for the part of the format string before it and transforms it into a new formatter that first does the left hand bit, then does its bit, then continues on with the rest of the format string.

```
structure Printf: PRINTF =
 struct
 datatype ('a, 'b) t = T of (unit -> 'a) -> 'b

 fun printf (T f) = f (fn () => ())

 fun ` s = T (fn a => (print s; a ()))

 fun D (T f, s) =
 T (fn g => f (fn () => fn i =>
 (print (Int.toString i); print s; g ())))

 fun F (T f, s) =
 T (fn g => f (fn () => fn i =>
 (print (Real.toString i); print s; g ())))

 end
```

## 210.9 Testing printf

```
structure Z = Test (Printf)
```

## 210.10 User-definable formats

The definition of the format characters is pretty much the same. Within the `Printf` structure we can define a format character generator.

```
val newFormat: ('a -> string) -> ('a -> 'b, 'c) t * string -> ('b, 'c) t =
 fn toString => fn (T f, s) =>
 T (fn th => f (fn () => fn a => (print (toString a); print s ; th ())))
val D = fn z => newFormat Int.toString z
val F = fn z => newFormat Real.toString z
```

## 210.11 A core Printf

We can now have a very small `PRINTF` signature, and define all the format strings externally to the core module.

```
signature PRINTF =
 sig
 type ('a, 'b) t
 val ` : string -> ('a, 'a) t
 val newFormat: ('a -> string) -> ('a -> 'b, 'c) t * string -> ('b, 'c) t
 val printf: (unit, 'a) t -> 'a
 end

structure Printf: PRINTF =
 struct
 datatype ('a, 'b) t = T of (unit -> 'a) -> 'b

 fun printf (T f) = f (fn () => ())
```

```

fun ` s = T (fn a => (print s; a ()))

fun newFormat toString (T f, s) =
 T (fn th =>
 f (fn () => fn a =>
 (print (toString a)
 ; print s
 ; th ())))
end

```

## 210.12 Extending to fprintf

One can implement fprintf by threading the outstream through all the transformers.

```

signature PRINTF =
 sig
 type ('a, 'b) t
 val ` : string -> ('a, 'a) t
 val fprintf: (unit, 'a) t * TextIO.outstream -> 'a
 val newFormat: ('a -> string) -> ('a -> 'b, 'c) t * string -> ('b, 'c) t
 val printf: (unit, 'a) t -> 'a
 end

structure Printf: PRINTF =
 struct
 type out = TextIO.outstream
 val output = TextIO.output

 datatype ('a, 'b) t = T of (out -> 'a) -> out -> 'b

 fun fprintf (T f, out) = f (fn _ => ()) out

 fun printf t = fprintf (t, TextIO.stdOut)

 fun ` s = T (fn a => fn out => (output (out, s); a out))

 fun newFormat toString (T f, s) =
 T (fn g =>
 f (fn out => fn a =>
 (output (out, toString a)
 ; output (out, s)
 ; g out)))
 end
end

```

## 210.13 Notes

- Lesson: instead of using dependent types for a function, express the the dependency in the type of the argument.
- If `fprintf` is partially applied, it will do the printing then and there. Perhaps this could be fixed with some kind of terminator. A syntactic or argument terminator is not necessary. A formatter can either be eager (as above) or lazy (as below). A lazy formatter accumulates enough state to print the entire string. The simplest lazy formatter concatenates the strings as they become available:

```

structure PrintfLazyConcat: PRINTF =
 struct
 datatype ('a, 'b) t = T of (string -> 'a) -> string -> 'b

 fun printf (T f) = f print ""
 end

```

```
fun ` s = T (fn th => fn s' => th (s' ^ s))

fun newFormat toString (T f, s) =
 T (fn th =>
 f (fn s' => fn a =>
 th (s' ^ toString a ^ s)))
end
```

It is somewhat more efficient to accumulate the strings as a list:

```
structure PrintfLazyList: PRINTF =
 struct
 datatype ('a, 'b) t = T of (string list -> 'a) -> string list -> 'b

 fun printf (T f) = f (List.app print o List.rev) []

 fun ` s = T (fn th => fn ss => th (s::ss))

 fun newFormat toString (T f, s) =
 T (fn th =>
 f (fn ss => fn a =>
 th (s::toString a::ss)))
 end
```

## 210.14 Also see

- [Printf](#)
- [Functional Unparsing](#)

## 211 ProductType

Standard ML has special syntax for products (tuples). A product type is written as

```
t1 * t2 * ... * tN
```

and a product pattern is written as

```
(p1, p2, ..., pN)
```

In most situations the syntax is quite convenient. However, there are situations where the syntax is cumbersome. There are also situations in which it is useful to construct and destruct n-ary products inductively, especially when using [Fold](#).

In such situations, it is useful to have a binary product datatype with an infix constructor defined as follows.

```
datatype ('a, 'b) product = & of 'a * 'b
infix &
```

With these definitions, one can write an n-ary product as a nested binary product quite conveniently.

```
x1 & x2 & ... & xn
```

Because of left associativity, this is the same as

```
((x1 & x2) & ...) & xn)
```

Because `&` is a constructor, the syntax can also be used for patterns.

The symbol `&` is inspired by the Curry-Howard isomorphism: the proof of a conjunction  $(A \ \& \ B)$  is a pair of proofs  $(a, \ b)$ .

### 211.1 Example: parser combinators

A typical parser combinator library provides a combinator that has a type of the form.

```
'a parser * 'b parser -> ('a * 'b) parser
```

and produces a parser for the concatenation of two parsers. When more than two parsers are concatenated, the result of the resulting parser is a nested structure of pairs

```
(...((p1, p2), p3)..., pN)
```

which is somewhat cumbersome.

By using a product type, the type of the concatenation combinator then becomes

```
'a parser * 'b parser -> ('a, 'b) product parser
```

While this doesn't stop the nesting, it makes the pattern significantly easier to write. Instead of

```
(...((p1, p2), p3)..., pN)
```

the pattern is written as

```
p1 & p2 & p3 & ... & pN
```

which is considerably more concise.

### 211.2 Also see

- [VariableArityPolymorphism](#)
- [Utilities](#)

## 212 Profiling

With MLton and `mlprof`, you can profile your program to find out bytes allocated, execution counts, or time spent in each function. To profile your program, compile with `-profile kind`, where *kind* is one of `alloc`, `count`, or `time`. Then, run the executable, which will write an `mlmon.out` file when it finishes. You can then run `mlprof` on the executable and the `mlmon.out` file to see the performance data.

Here are the three kinds of profiling that MLton supports.

- [ProfilingAllocation](#)
- [ProfilingCounts](#)
- [ProfilingTime](#)

### 212.1 Next steps

- [CallGraphs](#) to visualize profiling data.
  - [HowProfilingWorks](#)
  - [MLmon](#)
  - [MLtonProfile](#) to selectively profile parts of your program.
  - [ProfilingTheStack](#)
  - [ShowProf](#)
-

## 213 ProfilingAllocation

With MLton and `mlprof`, you can [profile](#) your program to find out how many bytes each function allocates. To do so, compile your program with `-profile alloc`. For example, suppose that `list-rev.sml` is the following.

```
fun append (l1, l2) =
 case l1 of
 [] => l2
 | x :: l1 => x :: append (l1, l2)

fun rev l =
 case l of
 [] => []
 | x :: l => append (rev l, [x])

val l = List.tabulate (1000, fn i => i)
val _ = 1 + hd (rev l)
```

Compile and run `list-rev` as follows.

```
% mlton -profile alloc list-rev.sml
% ./list-rev
% mlprof -show-line true list-rev mlmon.out
6,030,136 bytes allocated (108,336 bytes by GC)
 function cur

append list-rev.sml: 1 97.6%
<gc> 1.8%
<main> 0.4%
rev list-rev.sml: 6 0.2%
```

The data shows that most of the allocation is done by the `append` function defined on line 1 of `list-rev.sml`. The table also shows how special functions like `gc` and `main` are handled: they are printed with surrounding brackets. C functions are displayed similarly. In this example, the allocation done by the garbage collector is due to stack growth, which is usually the case.

The run-time performance impact of allocation profiling is noticeable, because it inserts additional C calls for object allocation.

Compile with `-profile alloc -profile-branch true` to find out how much allocation is done in each branch of a function; see [ProfilingCounts](#) for more details on `-profile-branch`.

## 214 ProfilingCounts

With MLton and `mlprof`, you can [profile](#) your program to find out how many times each function is called and how many times each branch is taken. To do so, compile your program with `-profile count -profile-branch true`. For example, suppose that `tak.sml` contains the following.

```
structure Tak =
 struct
 fun tak1 (x, y, z) =
 let
 fun tak2 (x, y, z) =
 if y >= x
 then z
 else
 tak1 (tak2 (x - 1, y, z),
 tak2 (y - 1, z, x),
 tak2 (z - 1, x, y))
 in
 if y >= x
 then z
 else
 tak1 (tak2 (x - 1, y, z),
 tak2 (y - 1, z, x),
 tak2 (z - 1, x, y))
 end
 end
 end

val rec f =
 fn 0 => ()
 | ~1 => print "this branch is not taken\n"
 | n => (Tak.tak1 (18, 12, 6) ; f (n-1))

val _ = f 5000

fun uncalled () = ()
```

Compile with count profiling and run the program.

```
% mlton -profile count -profile-branch true tak.sml
% ./tak
```

Display the profiling data, along with raw counts and file positions.

```
% mlprof -raw true -show-line true tak mlmon.out
623,610,002 ticks
```

function	cur	raw
Tak.tak1.tak2 tak.sml: 5	38.2%	(238,530,000)
Tak.tak1.tak2.<true> tak.sml: 7	27.5%	(171,510,000)
Tak.tak1 tak.sml: 3	10.7%	(67,025,000)
Tak.tak1.<true> tak.sml: 14	10.7%	(67,025,000)
Tak.tak1.tak2.<false> tak.sml: 9	10.7%	(67,020,000)
Tak.tak1.<false> tak.sml: 16	2.0%	(12,490,000)
f tak.sml: 23	0.0%	(5,001)
f.<branch> tak.sml: 25	0.0%	(5,000)
f.<branch> tak.sml: 23	0.0%	(1)
uncalled tak.sml: 29	0.0%	(0)
f.<branch> tak.sml: 24	0.0%	(0)

Branches are displayed with lexical nesting followed by `<branch>` where the function name would normally be, or `<true>` or `<false>` for if-expressions. It is best to run `mlprof` with `-show-line true` to help identify the branch.

One use of `-profile count` is as a code-coverage tool, to help find code in your program that hasn't been tested. For this reason, `mlprof` displays functions and branches even if they have a count of zero. As the above output shows, the branch on line 24 was never taken and the function defined on line 29 was never called. To see zero counts, it is best to run `mlprof` with `-raw true`, since some code (e.g. the branch on line 23 above) will show up with `0.0%` but may still have been executed and hence have a nonzero raw count.

---

## 215 ProfilingTheStack

For all forms of [Profiling](#), you can gather counts for all functions on the stack, not just the currently executing function. To do so, compile your program with `-profile-stack true`. For example, suppose that `list-rev.sml` contains the following.

```
fun append (l1, l2) =
 case l1 of
 [] => l2
 | x :: l1 => x :: append (l1, l2)

fun rev l =
 case l of
 [] => []
 | x :: l => append (rev l, [x])

val l = List.tabulate (1000, fn i => i)
val _ = 1 + hd (rev l)
```

Compile with stack profiling and then run the program.

```
% mlton -profile alloc -profile-stack true list-rev.sml
% ./list-rev
```

Display the profiling data.

```
% mlprof -show-line true list-rev mlmon.out
6,030,136 bytes allocated (108,336 bytes by GC)
 function cur stack GC

append list-rev.sml: 1 97.6% 97.6% 1.4%
<gc> 1.8% 0.0% 1.8%
<main> 0.4% 98.2% 1.8%
rev list-rev.sml: 6 0.2% 97.6% 1.8%
```

In the above table, we see that `rev`, defined on line 6 of `list-rev.sml`, is only responsible for 0.2% of the allocation, but is on the stack while 97.6% of the allocation is done by the user program and while 1.8% of the allocation is done by the garbage collector.

The run-time performance impact of `-profile-stack true` can be noticeable since there is some extra bookkeeping at every nontail call and return.

## 216 ProfilingTime

With MLton and `mlprof`, you can [profile](#) your program to find out how much time is spent in each function over an entire run of the program. To do so, compile your program with `-profile time`. For example, suppose that `tak.sml` contains the following.

```
structure Tak =
 struct
 fun tak1 (x, y, z) =
 let
 fun tak2 (x, y, z) =
 if y >= x
 then z
 else
 tak1 (tak2 (x - 1, y, z),
 tak2 (y - 1, z, x),
 tak2 (z - 1, x, y))
 in
 if y >= x
 then z
 else
 tak1 (tak2 (x - 1, y, z),
 tak2 (y - 1, z, x),
 tak2 (z - 1, x, y))
 end
 end
 end

val rec f =
 fn 0 => ()
 | ~1 => print "this branch is not taken\n"
 | n => (Tak.tak1 (18, 12, 6) ; f (n-1))

val _ = f 5000

fun uncalled () = ()
```

Compile with time profiling and run the program.

```
% mlton -profile time tak.sml
% ./tak
```

Display the profiling data.

```
% mlprof tak mlmon.out
6.00 seconds of CPU time (0.00 seconds GC)
function cur

Tak.tak1.tak2 75.8%
Tak.tak1 24.2%
```

This example shows how `mlprof` indicates lexical nesting: as a sequence of period-separated names indicating the structures and functions in which a function definition is nested. The profiling data shows that roughly three-quarters of the time is spent in the `Tak.tak1.tak2` function, while the rest is spent in `Tak.tak1`.

Display raw counts in addition to percentages with `-raw true`.

```
% mlprof -raw true tak mlmon.out
6.00 seconds of CPU time (0.00 seconds GC)
function cur raw

Tak.tak1.tak2 75.8% (4.55s)
Tak.tak1 24.2% (1.45s)
```

Display the file name and line number for each function in addition to its name with `-show-line true`.

```
% mlprof -show-line true tak mlmon.out
6.00 seconds of CPU time (0.00 seconds GC)
 function cur

Tak.tak1.tak2 tak.sml: 5 75.8%
Tak.tak1 tak.sml: 3 24.2%
```

Time profiling is designed to have a very small performance impact. However, in some cases there will be a run-time performance cost, which may perturb the results. There is more likely to be an impact with `-codegen c` than `-codegen native`.

You can also compile with `-profile time -profile-branch true` to find out how much time is spent in each branch of a function; see [ProfilingCounts](#) for more details on `-profile-branch`.

## 216.1 Caveats

With `-profile time`, use of the following in your program will cause a run-time error, since they would interfere with the profiler signal handler.

- `MLton.Itimer.set (MLton.Itimer.Prof, ...)`
- `MLton.Signal.setHandler (MLton.Signal.prof, ...)`

Also, because of the random sampling used to implement `-profile time`, it is best to have a long running program (at least tens of seconds) in order to get reasonable time

## 217 Projects

We have lots of ideas for projects to improve MLton, many of which we do not have time to implement, or at least haven't started on yet. Here is a list of some of those improvements, ranging from the easy (1 week) to the difficult (several months). If you have any interest in working on one of these, or some other improvement to MLton not listed here, please send mail to [MLton-devel@mlton.org](mailto:MLton-devel@mlton.org).

- Port to new platform: Windows (native, not Cygwin or MinGW), ...
- Source-level debugger
- Heap profiler
- Interfaces to libraries: OpenGL, Gtk+, D-BUS, ...
- More libraries written in SML (see [mltonlib](#))
- Additional constant types: `structure Real80: REAL, ...`
- An IDE (possibly integrated with [Eclipse](#))
- Port MLRISC and use for code generation
- Optimizations
  - Improved closure representation  
Right now, MLton's closure conversion algorithm uses a simple flat closure to represent each function.
    - \* <http://www.mlton.org/pipermail/mlton/2003-October/024570.html>
    - \* <http://www.mlton.org/pipermail/mlton-user/2007-July/001150.html>
    - \* [ShaoAppel94](#)
  - Elimination of array bounds checks in loops
  - Elimination of overflow checks on array index computations
  - Common-subexpression elimination of repeated array subscripts
  - Loop-invariant code motion, especially for tuple selects
  - Partial redundancy elimination
    - \* <http://www.mlton.org/pipermail/mlton/2006-April/028598.html>
  - Loop unrolling, especially for small loops
  - Auto-vectorization, for MMX/SSE/3DNow!/AltiVec (see the [work done on GCC](#))
  - Optimize `MLton_eq`: pointer equality is necessarily false when one of the arguments is freshly allocated in the block
- Analyses
  - Uncaught exception analysis

## 218 Pronounce

Here is how "MLton" sounds.

"MLton" is pronounced in two syllables, with stress on the first syllable. The first syllable sounds like the word *mill* (as in "steel mill"), the second like the word *tin* (as in "cookie tin").

---

## 219 PropertyList

A property list is a dictionary-like data structure into which properties (name-value pairs) can be inserted and from which properties can be looked up by name. The term comes from the Lisp language, where every symbol has a property list for storing information, and where the names are typically symbols and keys can be any type of value.

Here is an SML signature for property lists such that for any type of value a new property can be dynamically created to manipulate that type of value in a property list.

```
signature PROPERTY_LIST =
 sig
 type t

 val new: unit -> t
 val newProperty: unit -> {add: t * 'a -> unit,
 peek: t -> 'a option}

 end
```

Here is a functor demonstrating the use of property lists. It first creates a property list, then two new properties (of different types), and adds a value to the list for each property.

```
functor Test (P: PROPERTY_LIST) =
 struct
 val pl = P.new ()

 val {add = addInt: P.t * int -> unit, peek = peekInt} = P.newProperty ()
 val {add = addReal: P.t * real -> unit, peek = peekReal} = P.newProperty ()

 val () = addInt (pl, 13)
 val () = addReal (pl, 17.0)
 val s1 = Int.toString (valOf (peekInt pl))
 val s2 = Real.toString (valOf (peekReal pl))
 val () = print (concat [s1, " ", s2, "\n"])

 end
```

Applied to an appropriate implementation `PROPERTY_LIST`, the `Test` functor will produce the following output.

```
13 17.0
```

### 219.1 Implementation

Because property lists can hold values of any type, their implementation requires a [UniversalType](#). Given that, a property list is simply a list of elements of the universal type. Adding a property adds to the front of the list, and looking up a property scans the list.

```
functor PropertyList (U: UNIVERSAL_TYPE): PROPERTY_LIST =
 struct
 datatype t = T of U.t list ref

 fun new () = T (ref [])

 fun 'a newProperty () =
 let
 val (inject, out) = U.embed ()
 fun add (T r, a: 'a): unit = r := inject a :: (!r)
 fun peek (T r) =
 Option.map (valOf o out) (List.find (isSome o out) (!r))
 in
 {add = add, peek = peek}
 end

 end
```

If `U`: `UNIVERSAL_TYPE`, then we can test our code as follows.

```
structure Z = Test (PropertyList (U))
```

Of course, a serious implementation of property lists would have to handle duplicate insertions of the same property, as well as the removal of elements in order to avoid space leaks.

## 219.2 Also see

- MLton relies heavily on property lists for attaching information to syntax tree nodes in its intermediate languages. See [property-list.sig](#) and [property-list.fun](#).
  - The [MLRISCLibrary](#) uses property lists extensively.
-

## 220 Pygments

**Pygments** is a generic syntax highlighter. Here is a *lexer* for highlighting [Standard ML](#).

- `sml_lexer` — Provides highlighting of keywords, special constants, and (nested) comments.

### 220.1 Install and use

- Checkout all files and install as a **Pygments** plugin.

```
$ git clone https://github.com/MLton/mlton.git mlton
$ cd mlton/ide/pygments
$ python setup.py install
```

- Invoke `pygmentize` with `-l sml`.

### 220.2 Feedback

Comments and suggestions should be directed to [MatthewFluet](#).

---

## 221 RayRacine

Using SML in some *Semantic Web* stuff. Anyone interested in similar, please contact me. GreyLensman on #sml on IRC or rracine at this domain adelphia with a dot here net.

Current areas of coding.

1. Pretty solid, high performance Rete implementation - base functionality is complete.
  2. N3 parser - mostly complete
  3. RDF parser based on fxx - not started.
  4. Swerve HTTP server - 1/2 done.
  5. SPARQL implementation - not started.
  6. Persistent engine based on BerkelyDB - not started.
  7. Native implementation of Postgresql protocol - underway, ways to go.
  8. I also have a small change to the MLton compiler to add `packWord<N>` - changes compile but needs some more work, clean-up and unit tests.
-

## 222 Reachability

Reachability is a notion dealing with the graph of heap objects maintained at runtime. Nodes in the graph are heap objects and edges correspond to the pointers between heap objects. As the program runs, it allocates new objects (adds nodes to the graph), and those new objects can contain pointers to other objects (new edges in the graph). If the program uses mutable objects (refs or arrays), it can also change edges in the graph.

At any time, the program has access to some finite set of *root* nodes, and can only ever access nodes that are reachable by following edges from these root nodes. Nodes that are *unreachable* can be garbage collected.

### 222.1 Also see

- [MLtonFinalizable](#)
  - [MLtonWeak](#)
-

## 223 Redundant

[Redundant](#) is an optimization pass for the [SSA IntermediateLanguage](#), invoked from [SSASimplify](#).

### 223.1 Description

The redundant SSA optimization eliminates redundant function and label arguments; an argument of a function or label is redundant if it is always the same as another argument of the same function or label. The analysis finds an equivalence relation on the arguments of a function or label, such that all arguments in an equivalence class are redundant with respect to the other arguments in the equivalence class; the transformation selects one representative of each equivalence class and drops the binding occurrence of non-representative variables and renames use occurrences of the non-representative variables to the representative variable. The analysis finds the equivalence classes via a fixed-point analysis. Each vector of arguments to a function or label is initialized to equivalence classes that equate all arguments of the same type; one could start with an equivalence class that equates all arguments, but arguments of different type cannot be redundant. Variables bound in statements are initialized to singleton equivalence classes. The fixed-point analysis repeatedly refines these equivalence classes on the formals by the equivalence classes of the actuals.

### 223.2 Implementation

- `redundant.fun`

### 223.3 Details and Notes

The reason [Redundant](#) got put in was due to some output of the [ClosureConvert](#) pass converter where the environment record, or components of it, were passed around in several places. That may have been more relevant with polyvariant analyses (which are long gone). But it still seems possibly relevant, especially with more aggressive flattening, which should reveal some fields in nested closure records that are redundant.

## 224 RedundantTests

[RedundantTests](#) is an optimization pass for the [SSA IntermediateLanguage](#), invoked from [SSASimplify](#).

### 224.1 Description

This pass simplifies conditionals whose results are implied by a previous conditional test.

### 224.2 Implementation

- `redundant-tests.fun`

### 224.3 Details and Notes

An additional test will sometimes eliminate the overflow test when adding or subtracting 1. In particular, it will eliminate it in the following cases:

```
if x < y
 then ... x + 1 ...
else ... y - 1 ...
```

## 225 References

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

### 225.1 A

- [An Experimental Analysis of Self-Adjusting Computation](#) Umut Acar, Guy Blelloch, Matthias Blume, and Kanat Tangwongsan. [PLDI](#) 2006.
- [Compiling with Continuations](#) (addall). ISBN 0521416957. Andrew W. Appel. Cambridge University Press, 1992.
- [A Critique of Standard ML](#). Andrew W. Appel. [JFP](#) 1993.
- [Modern Compiler Implementation in ML](#) (addall). ISBN 0521582741 Andrew W. Appel. Cambridge University Press, 1998.
- [Shrinking Lambda Expressions in Linear Time](#). Andrew Appel and Trevor Jim. [JFP](#) 1997.
- [A lexical analyzer generator for Standard ML. Version 1.6.0](#) Andrew W. Appel, James S. Mattson, and David R. Tarditi. 1994

### 225.2 B

- [Tree Pattern Matching for ML](#). Marianne Baudinet, David MacQueen. 1985.  
 Describes the match compiler used in an early version of [SML/NJ](#).
- [Compiling Standard ML to Java Bytecodes](#). Nick Benton, Andrew Kennedy, and George Russell. [ICFP](#) 1998.
- [Interlanguage Working Without Tears: Blending SML with Java](#). Nick Benton and Andrew Kennedy. [ICFP](#) 1999.
- [Exceptional Syntax](#). Nick Benton and Andrew Kennedy. [JFP](#) 2001.
- [Adventures in Interoperability: The SML.NET Experience](#). Nick Benton, Andrew Kennedy, and Claudio Russo. [PPDP](#) 2004.
- [Shrinking Reductions in SML.NET](#). Nick Benton, Andrew Kennedy, Sam Lindley and Claudio Russo. [IFL](#) 2004.  
 Describes a linear-time implementation of an [Appel-Jim shrinker](#), using a mutable IL, and shows that it yields nice speedups in SML.NET's compile times. There are also benchmarks showing that SML.NET when compiled by MLton runs roughly five times faster than when compiled by SML/NJ.
- [Embedded Interpreters](#). Nick Benton. [JFP](#) 2005.
- [The Edinburgh SML Library](#). Dave Berry. University of Edinburgh Technical Report ECS-LFCS-91-148, 1991.
- [A semantics for ML concurrency primitives](#). Dave Berry, Robin Milner, and David N. Turner. [POPL](#) 1992.
- [Lessons From the Design of a Standard ML Library](#). Dave Berry. [JFP](#) 1993.
- [Compiling SML to Java Bytecode](#). Peter Bertelsen. Master's Thesis, 1998.
- [OO Programming styles in ML](#). Bernard Berthomieu. LAAS Report #2000111, 2000.
- [No-Longer-Foreign: Teaching an ML compiler to speak C "natively"](#). Matthias Blume. [BABEL](#) 2001.
- [Portable library descriptions for Standard ML](#). Matthias Blume. 2001.
- [Destructors, Finalizers, and Synchronization](#). Hans Boehm. [POPL](#) 2003.

Discusses a number of issues in the design of finalizers. Many of the design choices are consistent with [MLtonFinalizable](#).

### 225.3 C

- [Flow-directed Closure Conversion for Typed Languages](#). Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. [ESOP](#) 2000.  
Describes MLton's closure-conversion algorithm, which translates from its simply-typed higher-order intermediate language to its simply-typed first-order intermediate language.
- [A Parallel, Real-Time Garbage Collector](#). Perry Cheng and Guy E. Blelloch. [PLDI](#) 2001.
- [QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs](#). Koen Claessen and John Hughes. [ICFP](#) 2000.
- [Proper Tail Recursion and Space Efficiency](#). William D. Clinger. [PLDI](#) 1998.
- [Adding Threads to Standard ML](#). Eric C. Cooper and J. Gregory Morrisett. CMU Technical Report CMU-CS-90-186, 1990.
- [Stream Fusion: From Lists to Streams to Nothing at All](#). Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Submitted for publication. April 2007.

### 225.4 D

- [Principal Type-Schemes for Functional Programs](#). Luis Damas and Robin Milner. [POPL](#) 1982.
- [Functional Unparsing](#). Olivier Danvy. BRICS Technical Report RS 98-12, 1998.
- [Enhancements to eXene](#). Dustin B. Deboer. Master of Science Thesis, 2005.  
Describes ways to improve widget concurrency, handling of input focus, X resources and selections.
- [A Concurrent, Generational Garbage Collector for a Multithreaded Implementation of ML](#). Damien Doligez and Xavier Leroy. [POPL](#) 1993.
- [Modular Type Classes](#). Derek Dreyer, Robert Harper, Manuel M.T. Chakravarty. University of Chicago Technical Report TR-2007-02, 2006.
- [Principal Type Schemes for Modular Programs](#). Derek Dreyer and Matthias Blume. [ESOP](#) 2007.
- [Extensional Polymorphism](#). Catherin Dubois, Francois Rouaix, and Pierre Weis. [POPL](#) 1995.  
An extension of ML that allows the definition of ad-hoc polymorphic functions by inspecting the type of their argument.

### 225.5 E

- [Garbage Collection Safety for Region-based Memory Management](#). Martin Elsman. [TLDI](#) 2003.
- [Type-Specialized Serialization with Sharing](#) Martin Elsman. University of Copenhagen. IT University Technical Report TR-2004-43, 2004.

### 225.6 F

- [The Little MLer \(addall\)](#). ISBN 026256114X. Matthias Felleisen and Dan Freidman. The MIT Press, 1998.
- [Kill-Safe Synchronization Abstractions](#). Matthew Flatt and Robert Bruce Findler. [PLDI](#) 2004.
- [Contification Using Dominators](#). Matthew Fluet and Stephen Weeks. [ICFP](#) 2001.

Describes contification, a generalization of tail-recursion elimination that is an optimization operating on MLton's static single assignment (SSA) intermediate language.

- [Phantom Types and Subtyping](#). Matthew Fluet and Riccardo Pucella. [TCS](#) 2002.
-

- **Generic Polymorphism in ML**. J. Furuse. [JFLA](#) 2001.

The formalism behind G'CAML, which has an approach to ad-hoc polymorphism based on [Dubois95](#), the differences being in how type checking works and an improved compilation approach for typecase that does the matching at compile time, not run time.

## 225.7 G

- **A Multi-Threaded Higher-order User Interface Toolkit**. Emden R. Gansner and John H. Reppy. User Interface Software, 1993.
- **The Standard ML Basis Library**. ([addall](#)) ISBN 0521794781. Emden R. Gansner and John H. Reppy. Cambridge University Press, 2004.

An introduction and overview of the [Basis Library](#), followed by a detailed description of each module. The module descriptions are also available [online](#).

- **Region-based Memory Management in Cyclone**. Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. [PLDI](#) 2002.

## 225.8 H

- **Combining Region Inference and Garbage Collection**. Niels Hallenberg, Martin Elsman, and Mads Tofte. [PLDI](#) 2002.
- **Introduction to Programming Using SML** ([addall](#)). ISBN 0201398206. Michael R. Hansen, Hans Rischel. Addison-Wesley, 1999.
- **Typing First-Class Continuations in ML**. Robert Harper, Bruce F. Duba, and David MacQueen. [JFP](#) 1993.
- **On the Type Structure of Standard ML**. Robert Harper and John C. Mitchell. [TOPLAS](#) 1992.
- **On the Practicality and Desirability of Highly-concurrent, Mostly-functional Programming**. Carl H. Hauser and David B. Benson. [ACSD](#) 2004.

Describes the use of [Concurrent ML](#) in implementing the Ped text editor. Argues that using large numbers of threads and message passing style are a practical and effective ways of modularizing a program.

- **A Functional Description of TeX's Formula Layout**. Reinhold Heckmann and Reinhard Wilhelm. [JFP](#) 1997.
- **Safe and Flexible Memory Management in Cyclone**. Mike Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. University of Maryland Technical Report CS-TR-4514, 2003.
- **Compiling HOL4 to Native Code**. Joe Hurd. [TPHOLs](#) 2004.

Describes a port of HOL from Moscow ML to MLton, the difficulties encountered in compiling large programs, and the speedups achieved (roughly 10x).

## 225.9 I

## 225.10 J

- **Garbage Collection: Algorithms for Automatic Memory Management** ([addall](#)). ISBN 0471941484. Richard Jones. John Wiley & Sons, 1999.

**225.11 K**

- [Mistakes and Ambiguities in the Definition of Standard ML](#). Stefan Kahrs. University of Edinburgh Technical Report ECS-LFCS-93-257, 1993.

Describes a number of problems with the [1990 Definition](#), many of which were fixed in the [1997 Definition](#).

Also see the [addenda](#) published in 1996.

- [Generics for the Working ML'er](#). Vesa Karvonen. [ML](#) 2007.
- [Pickler Combinators](#). Andrew Kennedy. [JFP](#) 2004.
- [sml2java: A Source To Source Translator](#). Justin Koser, Haakon Larsen, Jeffrey A. Vaughan. [DPCOOL](#) 2003.

**225.12 L**

- [Faster Algorithms for Finding Minimal Consistent DFAs](#). Kevin Lang. 1999.
- [mGTK: An SML binding of Gtk+](#). Ken Larsen and Henning Niss. USENIX Annual Technical Conference, 2004.
- [The ZINC Experiment: an Economical Implementation of the ML Language](#). Xavier Leroy. Technical report 117, INRIA, 1990.

A detailed explanation of the design and implementation of a bytecode compiler and interpreter for ML with a machine model aimed at efficient implementation.

- [Polymorphism by Name for References and Continuations](#). Xavier Leroy. [POPL](#) 1993.
- [MLRISC Annotations](#). Allen Leung and Lal George. 1998.

**225.13 M**

- [Asynchronous Exceptions in Haskell](#). Simon Marlow, Simon Peyton Jones, Andy Moran and John Reppy. [PLDI](#) 2001.

An asynchronous exception is a signal that one thread can send to another, and is useful for the receiving thread to treat as an exception so that it can clean up locks or other state relevant to its current context.

- [An Ideal Model for Recursive Polymorphic Types](#). David MacQueen, Gordon Plotkin, Ravi Sethi. [POPL](#) 1984.
- [A Distributed Concurrent Implementation of Standard ML](#). David Matthews. University of Edinburgh Technical Report ECS-LFCS-91-174, 1991.
- [Papers on Poly/ML](#). David C. J. Matthews. University of Edinburgh Technical Report ECS-LFCS-95-335, 1995.
- [That About Wraps it Up: Using FIX to Handle Errors Without Exceptions, and Other Programming Tricks](#). Bruce J. McAdam. University of Edinburgh Technical Report ECS-LFCS-97-375, 1997.
- [A Just-In-Time Backend for Moscow ML 2.00 in SML](#). Bjarke Meier, Kristian Nørgaard. Masters Thesis, 2003.

A just-in-time compiler using GNU Lightning, showing a speedup of up to four times over Moscow ML's usual bytecode interpreter.

The full report is only available in Danish.

- A Theory of Type Polymorphism in Programming. Robin Milner. Journal of Computer and System Sciences, 1978.
- [How ML Evolved](#). Robin Milner. Polymorphism—The ML/LCF/Hope Newsletter, 1983.
- [Commentary on Standard ML \(online pdf\)](#). ([addall](#)) ISBN 0262631327. Robin Milner and Mads Tofte. The MIT Press, 1990.

Introduces and explains the notation and approach used in [The Definition of Standard ML](#).

- [The Definition of Standard ML](#). (addall) ISBN 0262631326. Robin Milner, Mads Tofte, and Robert Harper. The MIT Press, 1990.

Superseded by [The Definition of Standard ML \(Revised\)](#). Accompanied by the [Commentary on Standard ML](#).

- [The Definition of Standard ML \(Revised\)](#). (addall) ISBN 0262631814. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. The MIT Press, 1997.

A terse and formal specification of Standard ML's syntax and semantics. Supersedes [The Definition of Standard ML](#).

- [Principles and a Preliminary Design for ML2000](#). The ML2000 working group, 1999.
- [Automatic Code Generation from Coloured Petri Nets for an Access Control System](#). Kjeld H. Mortensen. Workshop on Practical Use of Coloured Petri Nets and Design/CPN, 1999.
- [Procs and Locks: a Portable Multiprocessing Platform for Standard ML of New Jersey](#). J. Gregory Morrisett and Andrew Tolmach. PPOPP 1993.
- [ML Grid Programming with ConCert](#). Tom Murphy VII. ML 2006.

## 225.14 N

- [fxp - Processing Structured Documents in SML](#). Andreas Neumann. Scottish Functional Programming Workshop, 1999.

Describes [fxp](#), an XML parser implemented in Standard ML.

- [Parsing and Querying XML Documents in SML](#). Andreas Neumann. Doctoral Thesis, 1999.
- [Compiling ML Polymorphism with Explicit Layout Bitmap](#). Huu-Duc Nguyen and Atsushi Ohori. PPD 2006.

## 225.15 O

- [Purely Functional Data Structures](#). ISBN 0521663504. Chris Okasaki. Cambridge University Press, 1999.
- [A Simple Semantics for ML Polymorphism](#). Atsushi Ohori. FPCA 1989.
- [A Polymorphic Record Calculus and Its Compilation](#). Atsushi Ohori. TOPLAS 1995.
- [An Unboxed Operational Semantics for ML Polymorphism](#). Atsushi Ohori and Tomonobu Takamizawa. LASC 1997.
- [Type-Directed Specialization of Polymorphism](#). Atsushi Ohori. IC 1999.
- Regular-expression derivatives reexamined. Scott Owens, John Reppy, and Aaron Turon. JFP 2009.

## 225.16 P

- [ML For the Working Programmer](#) (addall) ISBN 052156543X. Larry C. Paulson. Cambridge University Press, 1996.
- [The HiPE/x86 Erlang Compiler: System Description and Performance Evaluation](#). Mikael Pettersson, Konstantinos Sagonas, and Erik Johansson. FLOPS 2002.

Describes a native x86 Erlang compiler and a comparison of many different native x86 compilers (including MLton) and their register usage and call stack implementations.

- [User's Guide to ML-Lex and ML-Yacc](#) Roger Price. 2009.
- [Reactive Programming in Standard ML](#). Riccardo R. Pucella. 1998. ICCL 1998.

**225.17 Q****225.18 R**

- [Concurrent Programming in ML](#). Norman Ramsey. Princeton University Technical Report CS-TR-262-90, 1990.
- [Embedding an Interpreted Language Using Higher-Order Functions and Types](#). Norman Ramsey. [IVME](#) 2003.
- [An Expressive Language of Signatures](#). Norman Ramsey, Kathleen Fisher, and Paul Govereau. [ICFP](#) 2005.
- [Widening Integer Arithmetic](#). Kevin Redwine and Norman Ramsey. [CC](#) 2004.

Describes a method to implement numeric types and operations (like `Int31` or `Word17`) for sizes smaller than that provided by the processor.

- Synchronous Operations as First-Class Values. John Reppy. [PLDI](#) 1988.
- [Concurrent Programming in ML \(addall\)](#). ISBN 0521480892. John Reppy. Cambridge University Press, 1999.

Describes [ConcurrentML](#).

- [Definitional Interpreters Revisited](#). John C. Reynolds. [HOSC](#) 1998.
- [Definitional Interpreters for Higher-Order Programming Languages](#) John C. Reynolds. [HOSC](#) 1998.
- [Defects in the Revised Definition of Standard ML](#). Andreas Rossberg. 2001.

**225.19 S**

- [Dual-Mode Garbage Collection](#). Patrick M. Sansom. Workshop on the Parallel Implementation of Functional Languages, 1991.
- [When Do Match-Compilation Heuristics Matter](#). Kevin Scott and Norman Ramsey. University of Virginia Technical Report CS-2000-13, 2000.

Modified SML/NJ to experimentally compare a number of match-compilation heuristics and showed that choice of heuristic usually does not significantly affect code size or run time.

- [ML Pattern Match Compilation and Partial Evaluation](#). Peter Sestoft. Partial Evaluation, 1996.

Describes the derivation of the match compiler used in [Moscow ML](#).

- [Space-Efficient Closure Representations](#). Zhong Shao and Andrew W. Appel. [LFP](#) 2006.
- [Unix System Programming with Standard ML](#). Anthony L. Shipman. 2002.

Includes a description of the [Swerve](#) HTTP server written in SML.

- [Calcul Statique des Applications de Modules Parametres](#). Julien Signoles. [JFLA](#) 2003.

Describes a defunctorizer for OCaml, and compares it to existing defunctorizers, including MLton.

- [Incremental Execution of Transformation Specifications](#). Ganesh Sittampalam, Oege de Moor, and Ken Friis Larsen. [POPL](#) 2004.

Mentions a port from Moscow ML to MLton of [MuDDY](#), an SML wrapper around the [BuDDY](#) BDD package.

- [A Separate Compilation Extension to Standard ML](#). David Swasey, Tom Murphy VII, Karl Cray and Robert Harper. [ML](#) 2006.

**225.20 T**

- [ML-Yacc User's Manual, Version 2.4](#) David R. Tarditi and Andrew W. Appel. 2000.
- [No Assembly Required: Compiling Standard ML to C](#). David Tarditi, Peter Lee, and Anurag Acharya. 1990.
- [Object-oriented programming and Standard ML](#). Lars Thorup and Mads Tofte. [ML](#), 1994.
- Type Inference for Polymorphic References. Mads Tofte. [IC](#) 1990.
- [A Debugger for Standard ML](#). Andrew Tolmach and Andrew W. Appel. [JFP](#) 1995.
- [Combining Closure Conversion with Closure Analysis using Algebraic Types](#). Andrew Tolmach. [TIC](#) 1997.

Describes a closure-conversion algorithm for a monomorphic IL. The algorithm uses a unification-based flow analysis followed by defunctionalization and is similar to the approach used in MLton ([CejtinEtAl100](#)).

- [From ML to Ada: Strongly-typed Language Interoperability via Source Translation](#). Andrew Tolmach and Dino Oliva. [JFP](#) 1998.

Describes a compiler for RML, a core SML-like language. The compiler is similar in structure to MLton, using monomorphisation, defunctionalization, and optimization on a first-order IL.

**225.21 U**

- [Elements of ML Programming \(addall\)](#). ISBN 0137903871. Jeffrey D. Ullman. Prentice-Hall, 1998.

**225.22 V****225.23 W**

- [A Types-as-Sets Semantics for Milner-Style Polymorphism](#). Mitchell Wand. [POPL](#) 1984.
- [Managing Memory with Types](#). Daniel C. Wang. PhD Thesis.

Chapter 6 describes an implementation of a type-preserving garbage collector for MLton.

- [Type-Preserving Garbage Collectors](#). Daniel C. Wang and Andrew W. Appel. [POPL](#) 2001.

Shows how to modify MLton to generate a strongly-typed garbage collector as part of a program.

- [Programming With Recursion Schemes](#). Daniel C. Wang and Tom Murphy VII.

Describes a programming technique for data abstraction, along with benchmarks of MLton and other SML compilers.

- [Whole-Program Compilation in MLton](#). Stephen Weeks. [ML](#) 2006.
- [Simple Imperative Polymorphism](#). Andrew Wright. [LASC](#), 8(4):343-355, 1995.

The origin of the [ValueRestriction](#).

**225.24 X****225.25 Y**

- [Encoding Types in ML-like Languages](#). Zhe Yang. [ICFP](#) 1998.

**225.26 Z**

- **Stabilizers: A Modular Checkpointing Abstraction for Concurrent Functional Programs.** Lukasz Ziarek, Philip Schatz, and Suresh Jagannathan. [ICFP 2006](#).
- **Flattening tuples in an SSA intermediate representation.** Lukasz Ziarek, Stephen Weeks, and Suresh Jagannathan. [HOSC 2008](#).

**225.27 Abbreviations**

- ACSD = International Conference on Application of Concurrency to System Design
  - BABEL = Workshop on multi-language infrastructure and interoperability
  - CC = International Conference on Compiler Construction
  - DPCOOL = Workshop on Declarative Programming in the Context of OO Languages
  - ESOP = European Symposium on Programming
  - FLOPS = Symposium on Functional and Logic Programming
  - FPCA = Conference on Functional Programming Languages and Computer Architecture
  - HOSC = Higher-Order and Symbolic Computation
  - IC = Information and Computation
  - ICCL = IEEE International Conference on Computer Languages
  - ICFP = International Conference on Functional Programming
  - IFL = International Workshop on Implementation and Application of Functional Languages
  - IVME = Workshop on Interpreters, Virtual Machines and Emulators
  - JFLA = Journées Francophones des Langues Applicatives
  - JFP = Journal of Functional Programming
  - LASC = Lisp and Symbolic Computation
  - LFP = Lisp and Functional Programming
  - ML = Workshop on ML
  - PLDI = Conference on Programming Language Design and Implementation
  - POPL = Symposium on Principles of Programming Languages
  - PPDP = International Conference on Principles and Practice of Declarative Programming
  - PPOPP = Principles and Practice of Parallel Programming
  - TCS = IFIP International Conference on Theoretical Computer Science
  - TIC = Types in Compilation
  - TLDI = Workshop on Types in Language Design and Implementation
  - TOPLAS = Transactions on Programming Languages and Systems
  - TPHOLs = International Conference on Theorem Proving in Higher Order Logics
-

## 226 RefFlatten

[RefFlatten](#) is an optimization pass for the [SSA2 IntermediateLanguage](#), invoked from [SSA2Simplify](#).

### 226.1 Description

This pass flattens a `ref` cell into its containing object. The idea is to replace, where possible, a type like

```
(int ref * real)
```

with a type like

```
(int [m] * real)
```

where the `[m]` indicates a mutable field of a tuple.

### 226.2 Implementation

- `ref-flatten.fun`

### 226.3 Details and Notes

The savings is obvious, I hope. We avoid an extra heap-allocated object for the `ref`, which in the above case saves two words. We also save the time and code for the extra indirection at each get and set. There are lots of useful data structures (singly-linked and doubly-linked lists, union-find, Fibonacci heaps, ...) that I believe we are paying through the nose right now because of the absence of ref flattening.

The idea is to compute for each occurrence of a `ref` type in the program whether or not that `ref` can be represented as an offset of some object (constructor or tuple). As before, a unification-based whole-program with deep abstract values makes sure the analysis is consistent.

The only syntactic part of the analysis that remains is the part that checks that for a variable bound to a value constructed by `Ref_ref`:

- the object allocation is in the same block. This is pretty draconian, and it would be nice to generalize it some day to allow flattening as long as the `ref` allocation and object allocation "line up one-to-one" in the same loop-free chunk of code.
- updates occur in the same block (and hence it is safe-for-space because the containing object is still alive). It would be nice to relax this to allow updates as long as it can be proved that the container is live.

Prevent flattening of `unit ref`-s.

[RefFlatten](#) is safe for space. The idea is to prevent a `ref` being flattened into an object that has a component of unbounded size (other than possibly the `ref` itself) unless we can prove that at each point the `ref` is live, then the containing object is live too. I used a pretty simple approximation to liveness.

## 227 Regions

In region-based memory management, the heap is divided into a collection of regions into which objects are allocated. At compile time, either in the source program or through automatic inference, allocation points are annotated with the region in which the allocation will occur. Typically, although not always, the regions are allocated and deallocated according to a stack discipline.

MLton does not use region-based memory management; it uses traditional [GarbageCollection](#). We have considered integrating regions with MLton, but in our opinion it is far from clear that regions would provide MLton with improved performance, while they would certainly add a lot of complexity to the compiler and complicate reasoning about and achieving [SpaceSafety](#). Region-based memory management and garbage collection have different strengths and weaknesses; it's pretty easy to come up with programs that do significantly better under regions than under GC, and vice versa. We believe that it is the case that common SML idioms tend to work better under GC than under regions.

One common argument for regions is that the region operations can all be done in (approximately) constant time; therefore, you eliminate GC pause times, leading to a real-time GC. However, because of space safety concerns (see below), we believe that region-based memory management for SML must also include a traditional garbage collector. Hence, to achieve real-time memory management for MLton/SML, we believe that it would be both easier and more efficient to implement a traditional real-time garbage collector than it would be to implement a region system.

### 227.1 Regions, the ML Kit, and space safety

The [ML Kit](#) pioneered the use of regions for compiling Standard ML. The ML Kit maintains a stack of regions at run time. At compile time, it uses region inference to decide when data can be allocated in a stack-like manner, assigning it to an appropriate region. The ML Kit has put a lot of effort into improving the supporting analyses and representations of regions, which are all necessary to improve the performance.

Unfortunately, under a pure stack-based region system, space leaks are inevitable in theory, and costly in practice. Data for which region inference can not determine the lifetime is moved into the "global region" whose lifetime is the entire program. There are two ways in which region inference will place an object to the global region.

- When the inference is too conservative, that is, when the data is used in a stack-like manner but the region inference can't figure it out.
- When data is not used in a stack-like manner. In this case, correctness requires region inference to place the object

This global region is a source of space leaks. No matter what region system you use, there are some programs such that the global region must exist, and its size will grow to an unbounded multiple of the live data size. For these programs one must have a GC to achieve space safety.

To solve this problem, the ML Kit has undergone work to combine garbage collection with region-based memory management. [HallenbergEtAl02](#) and [Elsman03](#) describe the addition of a garbage collector to the ML Kit's region-based system. These papers provide convincing evidence for space leaks in the global region. They show a number of benchmarks where the memory usage of the program running with just regions is a large multiple (2, 10, 50, even 150) of the program running with regions plus GC.

These papers also give some numbers to show the ML Kit with just regions does better than either a system with just GC or a combined system. Unfortunately, a pure region system isn't practical because of the lack of space safety. And the other performance numbers are not so convincing, because they compare to an old version of SML/NJ and not at all with MLton. It would be interesting to see a comparison with a more serious collector.

### 227.2 Regions, Garbage Collection, and Cyclone

One possibility is to take Cyclone's approach, and provide both region-based memory management and garbage collection, but at the programmer's option ([GrossmanEtAl02](#), [HicksEtAl03](#)).

One might ask whether we might do the same thing — i.e., provide a `MLton.Regions` structure with explicit region based memory management operations, so that the programmer could use them when appropriate. [MatthewFluet](#) has thought about this question

- <http://www.cs.cornell.edu/People/fluet/rgn-monad/index.html>

Unfortunately, his conclusion is that the SML type system is too weak to support this option, although there might be a "poor-man's" version with dynamic checks.

---

## 228 Release20041109

This is an archived public release of MLton, version 20041109.

### 228.1 Changes since the last public release

- New platforms:
  - x86: FreeBSD 5.x, OpenBSD
  - PowerPC: Darwin (MacOSX)
- Support for the [ML Basis system](#), a new mechanism supporting programming in the very large, separate delivery of library sources, and more.
- Support for dynamic libraries.
- Support for [ConcurrentML](#) (CML).
- New structures: `Int2`, `Int3`, ..., `Int31` and `Word2`, `Word3`, ..., `Word31`.
- Front-end bug fixes and improvements.
- A new form of profiling with `-profile count`, which can be used to test code coverage.
- A bytecode generator, available via `-codegen bytecode`.
- Representation improvements:
  - Tuples and datatypes are packed to decrease space usage.
  - Ref cells may be unboxed into their containing object.
  - Arrays of tuples may represent the tuples unboxed.

For a complete list of changes and bug fixes since 20040227, see the [changelog](#).

### 228.2 Also see

- [Bugs20041109](#)
-

## 229 Release20051202

This is an archived public release of MLton, version 20051202.

### 229.1 Changes since the last public release

- The [MLton license](#) is now BSD-style instead of the GPL.
- New platforms: [X86/MinGW](#) and HPPA/Linux.
- Improved and expanded documentation, based on the MLton wiki.
- Compiler.
  - improved exception history.
  - [Command-line switches](#).
    - \* Added: `-as-opt`, `-mlb-path-map`, `-target-as-opt`, `-target-cc-opt`.
    - \* Removed: `-native`, `-sequence-unit`, `-warn-match`, `-warn-unused`.
- Language.
  - [FFI](#) syntax changes and extensions.
    - \* Added: `_symbol`.
    - \* Changed: `_export`, `_import`.
    - \* Removed: `_ffi`.
  - [ML Basis annotations](#).
    - \* Added: `allowFFI`, `nonexhaustiveExnMatch`, `nonexhaustiveMatch`, `redundantMatch`, `sequenceNonUnit`.
    - \* Deprecated: `allowExport`, `allowImport`, `sequenceUnit`, `warnMatch`.
- Libraries.
  - Basis Library.
    - \* Added: `Int1`, `Word1`.
  - [MLton structure](#).
    - \* Added: `Process.create`, `ProcEnv.setgroups`, `Rusage.measureGC`, `Socket.fdToSock`, `Socket.Ctl.getError`.
    - \* Changed: `MLton.Platform.Arch`.
  - Other libraries.
    - \* Added: [ckit](#), [ML-NLFFI library](#), [SML/NJ library](#).
- Tools.
  - Updates of `mlllex` and `mlyacc` from SML/NJ.
  - Added [mnlffigen](#).
  - [Profiling](#) supports better inclusion/exclusion of code.

For a complete list of changes and bug fixes since [Release20041109](#), see the [change log](#) and [Bugs20041109](#).

---

## 229.2 20051202 binary packages

- x86
  - [Cygwin 1.5.18-1](#)
  - [FreeBSD 5.4](#)
  - Linux
    - \* [Debian sid](#)
    - \* [Debian stable \(Sarge\)](#)
    - \* [RedHat 7.1-9.3 FC1-FC4](#)
    - \* [tgz](#) for other distributions (glibc 2.3)
  - [MinGW](#)
  - [NetBSD 2.0.2](#)
  - [OpenBSD 3.7](#)
- PowerPC
  - [Darwin 7.9.0 \(Mac OS X\)](#)
- Sparc
  - [Solaris 8](#)

## 229.3 20051202 source packages

- [source tgz](#)
- Debian [dsc](#), [diff.gz](#), [orig.tar.gz](#)
- RedHat [source rpm](#)

## 229.4 Packages available at other sites

- [Debian](#)
- [FreeBSD](#)
- Fedora Core [4](#) [5](#)
- [Ubuntu](#)

## 229.5 Also see

- [Bugs20051202](#)
- [MLton Guide \(20051202\)](#).

A snapshot of the MLton wiki at the time of release.

---

## 230 Release20070826

This is an archived public release of MLton, version 20070826.

### 230.1 Changes since the last public release

- New platforms:
  - [AMD64/Linux](#), [AMD64/FreeBSD](#)
  - [HPPA/HPUX](#)
  - [PowerPC/AIX](#)
  - [X86/Darwin \(Mac OS X\)](#)
- Compiler.
  - Support for 64-bit platforms.
    - \* Native amd64 codegen.
  - [Compile-time options](#).
    - \* Added: `-codegen amd64`, `-codegen x86`, `-default-type type`, `-profile-val {false|true}`.
    - \* Changed: `-stop f` (file listing now includes `.mlb` files).
  - Bytecode codegen.
    - \* Support for exception history.
    - \* Support for profiling.
- Language.
  - [ML Basis annotations](#).
    - \* Removed: `allowExport`, `allowImport`, `sequenceUnit`, `warnMatch`.
- Libraries.
  - [Basis Library](#).
    - \* Added: `PackWord16Big`, `PackWord16Little`, `PackWord64Big`, `PackWord64Little`.
    - \* Bug fixes: see [change log](#).
  - [MLton structure](#).
    - \* Added: `MLTON_MONO_ARRAY`, `MLTON_MONO_VECTOR`, `MLTON_REAL`, `MLton.BinIO.tempPrefix`, `MLton.CharArray`, `MLton.CharVector`, `MLton.Exn.defaultTopLevelHandler`, `MLton.Exn.getTopLevelHandler`, `MLton.Exn.setTopLevelHandler`, `MLton.IntInf.BigWord`, `MLton.IntInf.SmallInt`, `MLton.LargeReal`, `MLton.LargeWord`, `MLton.Real`, `MLton.Real32`, `MLton.Real64`, `MLton.Rlimit.Rlim`, `MLton.TextIO.tempPrefix`, `MLton.Vector.create`, `MLton.Word.bswap`, `MLton.Word8.bswap`, `MLton.Word16`, `MLton.Word32`, `MLton.Word64`, `MLton.Word8Array`, `MLton.Word8Vector`.
    - \* Changed: `MLton.Array.unfoldi`, `MLton.IntInf.rep`, `MLton.Rlimit`, `MLton.Vector.unfoldi`.
    - \* Deprecated: `MLton.Socket`.
  - Other libraries.
    - \* Added: [MLRISC library](#).
    - \* Updated: [ckit library](#), [SML/NJ library](#).
- Tools.

For a complete list of changes and bug fixes since [Release20051202](#), see the [change log](#) and [Bugs20051202](#).

## 230.2 20070826 binary packages

- AMD64
  - [Linux](#), glibc 2.3
- HPPA
  - [HPUX](#) 11.00 and above, statically linked against [GnuMP](#)
- PowerPC
  - [AIX](#) 5.1 and above, statically linked against [GnuMP](#)
  - [Darwin](#) 8.10 (Mac OS X), statically linked against [GnuMP](#)
  - [Darwin](#) 8.10 (Mac OS X), dynamically linked against [GnuMP](#) in `/opt/local/lib` (suitable for [MacPorts](#) install of [GnuMP](#))
- Sparc
  - [Solaris](#) 8 and above, statically linked against [GnuMP](#)
- X86
  - [Cygwin](#) 1.5.24-2
  - [Darwin \(.tgz\)](#) 8.10 (Mac OS X), dynamically linked against [GnuMP](#) in `/opt/local/lib` (suitable for [MacPorts](#) install of [GnuMP](#))
  - [Darwin \(.dmg\)](#) 8.10 (Mac OS X), dynamically linked against [GnuMP](#) in `/opt/local/lib` (suitable for [MacPorts](#) install of [GnuMP](#))
  - [Darwin \(.tgz\)](#) 8.10 (Mac OS X), statically linked against [GnuMP](#)
  - [Darwin \(.dmg\)](#) 8.10 (Mac OS X), statically linked against [GnuMP](#)
  - [FreeBSD](#)
  - [Linux](#), glibc 2.3
  - [Linux](#), glibc 2.1, statically linked against [GnuMP](#)
  - [MinGW](#), dynamically linked against [GnuMP](#) (requires `libgmp-3.dll`)
  - [MinGW](#), statically linked against [GnuMP](#)

## 230.3 20070826 source packages

- [source tgz](#)
- Debian [dsc](#), [diff.gz](#), [orig.tar.gz](#)

## 230.4 Packages available at other sites

- [Debian](#)
- [FreeBSD](#)
- [Fedora](#)
- [Ubuntu](#)

## 230.5 Also see

- [Bugs20070826](#)
- [MLton Guide \(20070826\)](#).  
A snapshot of the MLton wiki at the time of release.

## 231 Release20100608

This is an archived public release of MLton, version 20100608.

### 231.1 Changes since the last public release

- New platforms.
    - [AMD64/Darwin](#) (Mac OS X Snow Leopard)
    - [IA64/HPUX](#)
    - [PowerPC64/AIX](#)
  - Compiler.
    - [Command-line switches](#).
      - \* Added: `-mlb-path-var <name> <value>`
      - \* Removed: `-keep sml, -stop sml`
    - Improved constant folding of floating-point operations.
    - Experimental: Support for compiling to a C library; see [documentation](#).
    - Extended `-show-def-use output` to include types of variable definitions.
    - Deprecated features (to be removed in a future release)
      - \* Bytecode codegen: The bytecode codegen has not seen significant use and it is not well understood by any of the active developers.
      - \* Support for `.cm` files as input: The ML Basis system provides much better infrastructure for "programming in the very large" than the (very) limited support for CM. The `cm2mlb` tool (available in the source distribution) can be used to convert CM projects to MLB projects, preserving the CM scoping of module identifiers.
    - Bug fixes: see [changelog](#)
  - Runtime.
    - [@MLton switches](#).
      - \* Added: `may-page-heap {false|true}`
    - `may-page-heap`: By default, MLton will not page the heap to disk when unable to grow the heap to accommodate an allocation. (Previously, this behavior was the default, with no means to disable, with security an least-surprise issues.)
    - Bug fixes: see [changelog](#)
  - Language.
    - Allow numeric characters in [ML Basis](#) path variables.
  - Libraries.
    - [Basis Library](#).
      - \* Bug fixes: see [changelog](#)
    - [MLton structure](#).
      - \* Added: `MLton.equal`, `MLton.hash`, `MLton.Cont.isolate`, `MLton.GC.Statistics`, `MLton.Pointer.sizeofPointer`, `MLton.Socket.Address.toVector`
      - \* Changed:
      - \* Deprecated: `MLton.Socket`
    - [Unsafe structure](#).
-

- \* Added versions of all of the monomorphic array and vector structures.
- Other libraries.
  - \* Updated: [ckit library](#), [MLRISC library](#), [SML/NJ library](#).
- Tools.
  - mllex
    - \* Eliminated top-level `type int = Int.int` in output.
    - \* Include `(*#line line:col "file.lex" *)` directives in output.
    - \* Added `%posint` command, to set the `yypos` type and allow the lexing of multi-gigabyte files.
  - mlnlffigen
    - \* Added command-line switches `-linkage archive` and `-linkage shared`.
    - \* Deprecated command-line switch `-linkage static`.
    - \* Added support for [IA64](#) and [HPPA](#) targets.
  - mlyacc
    - \* Eliminated top-level `type int = Int.int` in output.
    - \* Include `(*#line line:col "file.grm" *)` directives in output.

For a complete list of changes and bug fixes since [Release20070826](#), see the [change log](#) and [Bugs20070826](#).

## 231.2 20100608 binary packages

- AMD64 (aka "x86-64" or "x64")
  - [Darwin \(.tgz\)](#) 10.3 (Mac OS X Snow Leopard), dynamically linked against [GnuMP](#) in `/opt/local/lib` (suitable for [MacPorts](#) install of [GnuMP](#))
  - [Darwin \(.tgz\)](#) 10.3 (Mac OS X Snow Leopard), statically linked against [GnuMP](#) (but requires [GnuMP](#) for generated executables)
  - [Linux](#), glibc 2.11
  - [Linux](#), statically linked
  - Windows MinGW 32/64 [self-extracting](#) (28MB) or [MSI](#) (61MB) installer
- X86
  - [Cygwin](#) 1.7.5
  - [Darwin \(.tgz\)](#) 9.8 (Mac OS X Leopard), dynamically linked against [GnuMP](#) in `/opt/local/lib` (suitable for [MacPorts](#) install of [GnuMP](#))
  - [Darwin \(.tgz\)](#) 9.8 (Mac OS X Leopard), statically linked against [GnuMP](#) (but requires [GnuMP](#) for generated executables)
  - [Linux](#), glibc 2.11
  - [Linux](#), statically linked
  - Windows MinGW 32/64 [self-extracting](#) (28MB) or [MSI](#) (61MB) installer

## 231.3 20100608 source packages

- [mlton-20100608.src.tgz](#)

### 231.4 Packages available at other sites

- [Debian](#)
- [FreeBSD](#)
- [Fedora](#)
- [Ubuntu](#)

### 231.5 Also see

- [Bugs20100608](#)
- [MLton Guide \(20100608\)](#).

A snapshot of the MLton wiki at the time of release.

---

## 232 Release20130715

Here you can download the latest public release of MLton, version 20130715. Elsewhere you can download newer, [Experimental releases](#).

### 232.1 Changes since the last public release

- Compiler.
  - Cosmetic improvements to type-error messages.
  - Removed features:
    - \* Bytecode codegen: The bytecode codegen had not seen significant use and it was not well understood by any of the active developers.
    - \* Support for `.cm` files as input: The [ML Basis system](#) provides much better infrastructure for "programming in the very large" than the (very) limited support for CM. The `cm2mlb` tool (available in the source distribution) can be used to convert CM projects to MLB projects, preserving the CM scoping of module identifiers.
  - Bug fixes: see [changelog](#)
- Runtime.
  - Bug fixes: see [changelog](#)
- Language.
  - Interpret `(*#line line:col "file" *)` directives as relative file names.
  - [ML Basis annotations](#).
    - \* Added: `resolveScope`
- Libraries.
  - [Basis Library](#).
    - \* Improved performance of `String.concatWith`.
    - \* Use bit operations for `REAL.class` and other low-level operations.
    - \* Support additional variables with `Posix.ProcEnv.sysconf`.
    - \* Bug fixes: see [changelog](#)
  - [MLton structure](#).
    - \* Removed: `MLton.Socket`
  - Other libraries.
    - \* Updated: [ckit library](#), [MLRISC library](#), [SML/NJ library](#)
    - \* Added: [MLLPT library](#)
- Tools.
  - `mllex`
    - \* Generate `(*#line line:col "file.lex" *)` directives with simple (relative) file names, rather than absolute paths.
  - `mlyacc`
    - \* Generate `(*#line line:col "file.grm" *)` directives with simple (relative) file names, rather than absolute paths.
    - \* Fixed bug in comment-handling in lexer.

For a complete list of changes and bug fixes since [Release20100608](#), see the [changelog](#) and [Bugs20100608](#).

---

## 232.2 20130715 binary packages

- AMD64 (aka "x86-64" or "x64")
  - [Darwin \(.tgz\)](#) 11.4 (Mac OS X Lion), dynamically linked against [GnuMP](#) in `/opt/local/lib` (suitable for [MacPorts](#) install of [GnuMP](#))
  - [Darwin \(.tgz\)](#) 11.4 (Mac OS X Lion), statically linked against [GnuMP](#) (but requires [GnuMP](#) for generated executables)
  - [Linux](#), glibc 2.15
- X86
  - [Linux](#), glibc 2.15

## 232.3 20130715 source packages

- [mlton-20130715.src.tgz](#)

## 232.4 Downstream packages

- [Debian](#)
- [FreeBSD](#)
- [Fedora](#)
- [Ubuntu](#)

## 232.5 Also see

- [Bugs20130715](#)
- [MLton Guide \(20130715\)](#).

A snapshot of the MLton website at the time of release.

---

## 233 ReleaseChecklist

### 233.1 Advance preparation for release

- Update `doc/changelog`.
  - Write entries for missing notable commits.
  - Write summary of changes from previous release.
  - Update with estimated release date.
- Update `doc/README`.
  - Check features and description.
- Update `man/{mlton,mlprof}.1`.
  - Check compile-time and run-time options in `man/mlton.1`.
  - Check options in `man/mlprof.1`.
  - Update with estimated release date.
- Update `doc/guide`.
  - Synchronize [Features](#) page with `doc/README`.
  - Update [Credits](#) page with acknowledgements.
  - Create **ReleaseYYYYMM??** page (i.e., forthcoming release) based on **ReleaseXXXXLLCC** (i.e., previous release).
    - \* Update summary from `doc/changelog`.
    - \* Update links to estimated release date.
  - Create **BugsYYYYMM??** page based on **BugsXXXXLLCC**.
    - \* Update links to estimated release date.
  - Spell check pages.
- Ensure that all updates are pushed to master branch of `mlton`.

### 233.2 Prepare sources for tagging

- Update `doc/changelog`.
  - Update with proper release date.
- Update `man/{mlton,mlprof}.1`.
  - Update with proper release date.
- Update `doc/guide`.
  - Rename **ReleaseYYYYMM??** to **ReleaseYYYYMMDD** with proper release date.
    - \* Update links with proper release date.
  - Rename **BugsYYYYMM??** to **BugsYYYYMMDD** with proper release date.
    - \* Update links with proper release date.
  - Update **ReleaseXXXXLLCC**.
    - \* Change intro to "This is an archived public release of MLton, version XXXXLLCC."
  - Update [Home](#) with note of new release.
    - \* Change `What's new?` text to `Please try out our new release, <:ReleaseYYYYMMDD:MLton YYYYMMDD>`.
  - Update [Releases](#) with new release.
  - Clear [Experimental](#).
- Ensure that all updates are pushed to master branch of `mlton`.

### 233.3 Tag sources

- Shell commands:

```
git clone http://github.com/MLton/mlton mlton.git
cd mlton.git
git checkout master
git tag -a -m "Tagging YYYYMMDD release" on-YYYYMMDD-release master
git push origin on-YYYYMMDD-release
```

### 233.4 Packaging

#### 233.4.1 SourceForge FRS

- Create YYYYMMDD directory:

```
sftp user@frs.sourceforge.net:/home/frs/project/mlton/mlton
sftp> mkdir YYYYMMDD
sftp> quit
```

#### 233.4.2 Source release

- Create mlton-YYYYMMDD.src.tgz:

```
git clone http://github.com/MLton/mlton mlton
cd mlton
git checkout on-YYYYMMDD-release
make version VERSION=YYYYMMDD
(cd mllex ; latexmk -pdf lexgen ; latexmk -c lexgen ; make mllex.pdf)
(cd mlyacc ; (cd doc; latexmk -pdf mlyaccc ; latexmk -c mlyacc); make mlyacc.pdf)
make -C doc/guide
make release VERSION=YYYYMMDD
cd ..
```

or

```
wget https://github.com/MLton/mlton/archive/on-YYYYMMDD-release.tar.gz
tar xzvf on-YYYYMMDD-release.tar.gz
cd mlton-on-YYYYMMDD-release
make version VERSION=YYYYMMDD
(cd mllex ; latexmk -pdf lexgen ; latexmk -c lexgen ; make mllex.pdf)
(cd mlyacc ; (cd doc; latexmk -pdf mlyaccc ; latexmk -c mlyacc); make mlyacc.pdf)
make -C doc/guide
make release VERSION=YYYYMMDD
cd ..
```

- Upload mlton-YYYYMMDD.src.tgz:

```
scp mlton-YYYYMMDD.src.tgz user@frs.sourceforge.net:/home/frs/project/mlton/mlton/YYYYMMDD ↔
/
```

- Update **ReleaseYYYYMMDD** with mlton-YYYYMMDD.src.tgz link.

### 233.4.3 Binary releases

- Build and create `mlton-YYYYMMDD-1.ARCH-OS.tgz`:

```
wget http://sourceforge.net/projects/mlton/files/mlton/YYYYMMDD/mlton-YYYYMMDD.src.tgz
tar xzvf mlton-YYYYMMDD.src.tgz
cd mlton-YYYYMMDD
make all
make install
cd install
tar czvf ../mlton-YYYYMMDD-1.ARCH-OS.tgz *
cd ../../
```

- Upload `mlton-YYYYMMDD-1.ARCH-OS.tgz`:

```
scp mlton-YYYYMMDD-1.ARCH-OS.tgz user@frs.sourceforge.net:/home/frs/project/mlton/mlton/ ↔
YYYYMMDD/
```

- Update **Release YYYYMMDD** with `mlton-YYYYMMDD-1.ARCH-OS.tgz` link.

### 233.5 Website

- `guide/YYYYMMDD` gets a copy of `doc/guide/localhost`.

- Shell commands:

```
wget http://sourceforge.net/projects/mlton/files/mlton/YYYYMMDD/mlton-YYYYMMDD.src.tgz
tar xzvf mlton-YYYYMMDD.src.tgz
cd mlton-YYYYMMDD
cd doc/guide
cp -prf localhost YYYYMMDD
tar czvf guide-YYYYMMDD.tgz YYYYMMDD
rsync -avzP --delete -e ssh YYYYMMDD user@web.sourceforge.net:/home/project-web/mlton/ ↔
htdocs/guide/
rsync -avzP --delete -e ssh guide-YYYYMMDD.tgz user@web.sourceforge.net:/home/project-web/ ↔
mlton/htdocs/guide/
```

### 233.6 Announce release

- Mail announcement to:

- [MLton-devel@mlton.org](mailto:MLton-devel@mlton.org)
- [MLton-user@mlton.org](mailto:MLton-user@mlton.org)

- Update [OtherSites](#) that have MLton pages.

### 233.7 Misc.

- dupload Debian package.
  - Generate new [Performance](#) numbers.
-

## 234 Releases

Public releases of MLton:

- [Release20130715](#)
  - [Release20100608](#)
  - [Release20070826](#)
  - [Release20051202](#)
  - [Release20041109](#)
  - [Release20040227](#)
  - [Release20030716](#)
  - [Release20030711](#)
  - [Release20030312](#)
  - [Release20020923](#)
  - [Release20020410](#)
  - [Release20011006](#)
  - [Release20010806](#)
  - [Release20010706](#)
  - [Release20000906](#)
  - [Release20000712](#)
  - [Release19990712](#)
  - [Release19990319](#)
  - [Release19980826](#)
-

## 235 RemoveUnused

[RemoveUnused](#) is an optimization pass for both the [SSA](#) and [SSA2 IntermediateLanguages](#), invoked from [SSASimplify](#) and [SSA2Simplify](#).

### 235.1 Description

This pass aggressively removes unused:

- datatypes
- datatype constructors
- datatype constructor arguments
- functions
- function arguments
- function returns
- blocks
- block arguments
- statements (variable bindings)
- handlers from non-tail calls (mayRaise analysis)
- continuations from non-tail calls (mayReturn analysis)

### 235.2 Implementation

- [remove-unused.fun](#)
- [remove-unused2.fun](#)

### 235.3 Details and Notes

---

## 236 Restore

`Restore` is a rewrite pass for the [SSA](#) and [SSA2 IntermediateLanguages](#), invoked from [KnownCase](#) and [LocalRef](#).

### 236.1 Description

This pass restores the SSA condition for a violating [SSA](#) or [SSA2](#) program; the program must satisfy:

Every path from the root to a use of a variable (excluding globals) passes through a def of that variable.

### 236.2 Implementation

- [restore.sig](#)
- [restore.fun](#)
- [restore2.sig](#)
- [restore2.fun](#)

### 236.3 Details and Notes

Based primarily on Section 19.1 of [Modern Compiler Implementation in ML](#).

The main deviation is the calculation of liveness of the violating variables, which is used to predicate the insertion of phi arguments. This is due to the algorithm's bias towards imperative languages, for which it makes the assumption that all variables are defined in the start block and all variables are "used" at exit.

This is "optimized" for restoration of functions with small numbers of violating variables — use bool vectors to represent sets of violating variables.

Also, we use a `Promise.t` to suspend part of the dominance frontier computation.

---

## 237 ReturnStatement

Programmers coming from languages that have a `return` statement, such as C, Java, and Python, often ask how one can translate functions that return early into SML. This page briefly describes a number of ways to translate uses of `return` to SML.

### 237.1 Conditional iterator function

A conditional iterator function, such as `List.find`, `List.exists`, or `List.all` is probably what you want in most cases. Unfortunately, it might be the case that the particular conditional iteration pattern that you want isn't provided for your data structure. Usually the best alternative in such a case is to implement the desired iteration pattern as a higher-order function. For example, to implement a `find` function for arrays (which already exists as `Array.find`) one could write

```
fun find predicate array = let
 fun loop i =
 if i = Array.length array then
 NONE
 else if predicate (Array.sub (array, i)) then
 SOME (Array.sub (array, i))
 else
 loop (i+1)
in
 loop 0
end
```

Of course, this technique, while probably the most common case in practice, applies only if you are essentially iterating over some data structure.

### 237.2 Escape handler

Probably the most direct way to translate code using `return` statements is to basically implement `return` using exception handling. The mechanism can be packaged into a reusable module with the signature (`exit.sig`):

```
(**
 * Signature for exit (or escape) handlers.
 *
 * Note that the implementation necessarily uses exception handling. This
 * is to make proper resource handling possible. Exceptions raised by the
 * implementation can be caught by wildcard exception handlers. Wildcard
 * exception handlers should generally reraise exceptions after performing
 * their effects.
 *)
signature EXIT = sig
 type 'a t
 (** The type of exits. *)

 val within : ('a t, 'a) CPS.t
 (**
 * Sets up an exit and passes it to the given function. The function
 * may then return normally or by calling {to} with the exit and a
 * return value. For example,
 *
 * > Exit.within
 * > (fn l =>
 * > if condition then
 * > Exit.to l l
 * > else
 * > 2)
 *
 *)
```

```

* evaluates either to {1} or to {2} depending on the {condition}.
*
* Note that the function receiving the exit is called from a non-tail
* position.
*)

val to : 'a t -> 'a -> 'b
(**
 * {to l v} returns from the {within} invocation that introduced the
 * exit {l} with the value {v}. Evaluating {to l v} outside of the
 * {within} invocation that introduced {l} is a programming error and
 * raises an exception.
 *
 * Note that the type variable {'b} only appears as the return type.
 * This means that {to} doesn't return normally to the caller and can
 * be called from a context of any type.
 *)

val call : ('a -> 'b, 'a) CPS.t
(**
 * Simpler, but less flexibly typed, interface to {within} and {to}.
 * Specifically, {call f} is equivalent to {within (f o to)}.
 *)
end

```

([Typing First-Class Continuations in ML](#) discusses the typing of a related construct.) The implementation (`exit.sml`) is straightforward:

```

structure Exit :> EXIT = struct
 type 'a t = 'a -> exn

 fun within block = let
 exception EscapedExit of 'a
 in
 block EscapedExit
 handle EscapedExit value => value
 end

 fun to exit value = raise exit value

 fun call block = within (block o to)
end

```

Here is an example of how one could implement a `find` function given an `app` function:

```

fun appToFind (app : ('a -> unit) -> 'b -> unit)
 (predicate : 'a -> bool)
 (data : 'b) =
 Exit.call
 (fn return =>
 (app (fn x =>
 if predicate x then
 return (SOME x)
 else
 ()))
 data
 ; NONE)

```

In the above, as soon as the expression `predicate x` evaluates to `true` the `app` invocation is terminated.

### 237.3 Continuation-passing Style (CPS)

A general way to implement complex control patterns is to use **CPS**. In CPS, instead of returning normally, functions invoke a function passed as an argument. In general, multiple continuation functions may be passed as arguments and the ordinary return continuation may also be used. As an example, here is a function that finds the leftmost element of a binary tree satisfying a given predicate:

```
datatype 'a tree = LEAF | BRANCH of 'a tree * 'a * 'a tree

fun find predicate = let
 fun recurse continue =
 fn LEAF =>
 continue ()
 | BRANCH (lhs, elem, rhs) =>
 recurse
 (fn () =>
 if predicate elem then
 SOME elem
 else
 recurse continue rhs)
 lhs
in
 recurse (fn () => NONE)
end
```

Note that the above function returns as soon as the leftmost element satisfying the predicate is found.

## 238 RSSA

RSSA is an [IntermediateLanguage](#), translated from SSA2 by [ToRSSA](#), optimized by [RSSASimplify](#), and translated by [ToMachine](#) to [Machine](#).

### 238.1 Description

RSSA is a [IntermediateLanguage](#) that makes representation decisions explicit.

### 238.2 Implementation

- [rssa.sig](#)
- [rssa.fun](#)

### 238.3 Type Checking

The new type language is aimed at expressing bit-level control over layout and associated packing of data representations. There are singleton types that denote constants, other atomic types for things like integers and reals, and arbitrary sum types and sequence (tuple) types. The big change to the type system is that type checking is now based on subtyping, not type equality. So, for example, the singleton type `0xFFFFFEEBB` whose only inhabitant is the eponymous constant is a subtype of the type `Word32`.

### 238.4 Details and Notes

SSA is an abbreviation for Static Single Assignment. The [RSSA IntermediateLanguage](#) is a variant of SSA.

---

## 239 RSSAShrink

[RSSAShrink](#) is an optimization pass for the [RSSA IntermediateLanguage](#).

### 239.1 Description

This pass implements a whole family of compile-time reductions, like:

- constant folding, copy propagation
- inline the `Goto` to a block with a unique predecessor

### 239.2 Implementation

- `rssa.fun`

### 239.3 Details and Notes

---

## 240 RSSASimplify

The optimization passes for the [RSSA IntermediateLanguage](#) are collected and controlled by the `Backend` functor (`backend.sig`, `backend.fun`).

The following optimization pass is implemented:

- [RSSAShrink](#)

The following implementation passes are implemented:

- [ImplementHandlers](#)
- [ImplementProfiling](#)
- [InsertLimitChecks](#)
- [InsertSignalChecks](#)

The optimization passes can be controlled from the command-line by the options

- `-diag-pass <pass>` — keep diagnostic info for pass
  - `-drop-pass <pass>` — omit optimization pass
  - `-keep-pass <pass>` — keep the results of pass
-

## 241 RunningOnAIX

MLton runs fine on AIX.

### 241.1 Also see

- [RunningOnPowerPC](#)
  - [RunningOnPowerPC64](#)
-

## 242 RunningOnAlpha

MLton runs fine on the Alpha architecture.

### 242.1 Notes

- When compiling for Alpha, MLton doesn't support native code generation (`-codegen native`). Hence, performance is not as good as it might be and compile times are longer. Also, the quality of code generated by `gcc` is important. By default, MLton calls `gcc -O1`. You can change this by calling MLton with `-cc-opt -O2`.
  - When compiling for Alpha, MLton uses `-align 8` by default.
-

## 243 RunningOnAMD64

MLton runs fine on the AMD64 (aka "x86-64" or "x64") architecture.

### 243.1 Notes

- When compiling for AMD64, MLton targets the 64-bit ABI.
  - On AMD64, MLton supports native code generation (`-codegen native` or `-codegen amd64`).
  - When compiling for AMD64, MLton uses `-align 8` by default. Using `-align 4` may be incompatible with optimized builds of the [GnuMP](#) library, which assume 8-byte alignment. (See the thread at <http://www.mlton.org/pipermail/mlton/2009-October/030674.html> for more details.)
-

## 244 RunningOnARM

MLton runs fine on the ARM architecture.

### 244.1 Notes

- When compiling for ARM, MLton doesn't support native code generation (`-codegen native`). Hence, performance is not as good as it might be and compile times are longer. Also, the quality of code generated by `gcc` is important. By default, MLton calls `gcc -O1`. You can change this by calling MLton with `-cc-opt -O2`.
-

## 245 RunningOnCygwin

MLton runs on the [Cygwin](#) emulation layer, which provides a Posix-like environment while running on Windows. To run MLton with Cygwin, you must first install Cygwin on your Windows machine. To do this, visit the [Cygwin site](#) from your Windows machine and run their `setup.exe` script. Then, you can unpack the MLton binary `tgz` in your Cygwin environment.

To run MLton cross-compiled executables on Windows, you must install the `Cygwin.dll` on the Windows machine.

### 245.1 Known issues

- Time profiling is disabled.
- Cygwin's `mmap` emulation is less than perfect. Sometimes it interacts badly with `Posix.Process.fork`.
- The `socket.sml` regression test fails. We suspect this is not a bug and is simply due to our test relying on a certain behavior when connecting to a socket that has not yet accepted, which is handled differently on Cygwin than other platforms. Any help in understanding and resolving this issue is appreciated.

### 245.2 Also see

- [RunningOnMinGW](#)
-

## 246 RunningOnDarwin

MLton runs fine on Darwin (and on Mac OS X).

### 246.1 Notes

- MLton requires the [GnuMP](#) library, which is available via [Fink](#), [MacPorts](#), [Homebrew](#).
- For Intel-based Macs, MLton targets the [AMD64 architecture](#) on Darwin 10 (Mac OS X Snow Leopard) and higher and targets the [x86 architecture](#) on Darwin 8 (Mac OS X Tiger) and Darwin 9 (Mac OS X Leopard).

### 246.2 Known issues

- Executables that save and load worlds on Darwin 11 (Mac OS X Lion) and higher should be compiled with `-link-opt -fno-PIE` ; see [MLtonWorld](#) for more details.
- [ProfilingTime](#) may give inaccurate results on multi-processor machines. The SIGPROF signal, used to sample the profiled program, is supposed to be delivered 100 times a second (i.e., at 10000us intervals), but there can be delays of over 1 minute between the delivery of consecutive SIGPROF signals. A more complete description may be found [here](#) and [here](#).

### 246.3 Also see

- [RunningOnAMD64](#)
  - [RunningOnPowerPC](#)
  - [RunningOnX86](#)
-

## 247 RunningOnFreeBSD

MLton runs fine on [FreeBSD](#).

### 247.1 Notes

- MLton is available as a [FreeBSD port](#).

### 247.2 Known issues

- Executables often run more slowly than on a comparable Linux machine. We conjecture that part of this is due to costs due to heap resizing and kernel zeroing of pages. Any help in solving the problem would be appreciated.
- FreeBSD defaults to a datasize limit of 512M, even if you have more than that amount of memory in the computer. Hence, your MLton process will be limited in the amount of memory it has. To fix this problem, turn up the datasize and the default datasize available to a process: Edit `/boot/loader.conf` to set the limits. For example, the setting

```
kern.maxdsiz="671088640"
kern.dfldsiz="671088640"
kern.maxssiz="134217728"
```

will give a process 640M of datasize memory, default to 640M available and set 128M of stack size memory.

## 248 RunningOnHPPA

MLton runs fine on the HPPA architecture.

### 248.1 Notes

- When compiling for HPPA, MLton targets the 32-bit HPPA architecture.
  - When compiling for HPPA, MLton doesn't support native code generation (`-codegen native`). Hence, performance is not as good as it might be and compile times are longer. Also, the quality of code generated by `gcc` is important. By default, MLton calls `gcc -O1`. You can change this by calling MLton with `-cc-opt -O2`.
  - When compiling for HPPA, MLton uses `-align 8` by default. While this speeds up reals, it also may increase object sizes. If your program does not make significant use of reals, you might see a speedup with `-align 4`.
-

## 249 RunningOnHPUX

MLton runs fine on HPUX.

### 249.1 Also see

- [RunningOnHPPA](#)
-

## 250 RunningOnIA64

MLton runs fine on the IA64 architecture.

### 250.1 Notes

- When compiling for IA64, MLton targets the 64-bit ABI.
  - When compiling for IA64, MLton doesn't support native code generation (`-codegen native`). Hence, performance is not as good as it might be and compile times are longer. Also, the quality of code generated by `gcc` is important. By default, MLton calls `gcc -O1`. You can change this by calling MLton with `-cc-opt -O2`.
  - When compiling for IA64, MLton uses `-align 8` by default.
  - On the IA64, the [GnuMP](#) library supports multiple ABIs. See the [GnuMP](#) page for more details.
-

## 251 RunningOnLinux

MLton runs fine on Linux.

---

## 252 RunningOnMinGW

MLton runs on [MinGW](#), a library for porting Unix applications to Windows. Some library functionality is missing or changed.

### 252.1 Notes

- To compile MLton on MinGW:
  - The [GnuMP](#) library is required.
  - The Bash shell is required. If you are using a prebuilt MSYS, you probably want to symlink `bash` to `sh`.

### 252.2 Known issues

- Many functions are unimplemented and will raise `SysErr`.
  - `MLton.Itimer.set`
  - `MLton.ProcEnv.setgroups`
  - `MLton.Process.kill`
  - `MLton.Process.reap`
  - `MLton.World.load`
  - `OS.FileSys.readLink`
  - `OS.IO.poll`
  - `OS.Process.terminate`
  - `Posix.FileSys.chown`
  - `Posix.FileSys.fchown`
  - `Posix.FileSys.fpathconf`
  - `Posix.FileSys.link`
  - `Posix.FileSys.mkfifo`
  - `Posix.FileSys.pathconf`
  - `Posix.FileSys.readlink`
  - `Posix.FileSys.symlink`
  - `Posix.IO.dupfd`
  - `Posix.IO.getfd`
  - `Posix.IO.getfl`
  - `Posix.IO.getlk`
  - `Posix.IO.setfd`
  - `Posix.IO.setfl`
  - `Posix.IO.setlkw`
  - `Posix.IO.setlk`
  - `Posix.ProcEnv.ctermid`
  - `Posix.ProcEnv.getegid`
  - `Posix.ProcEnv.geteuid`
  - `Posix.ProcEnv.getgid`
  - `Posix.ProcEnv.getgroups`
  - `Posix.ProcEnv.getlogin`

- 
- Posix.ProcEnv.getpgrp
  - Posix.ProcEnv.getpid
  - Posix.ProcEnv.getppid
  - Posix.ProcEnv.getuid
  - Posix.ProcEnv.setgid
  - Posix.ProcEnv.setpgid
  - Posix.ProcEnv.setsid
  - Posix.ProcEnv.setuid
  - Posix.ProcEnv.sysconf
  - Posix.ProcEnv.times
  - Posix.ProcEnv.ttyname
  - Posix.Process.exece
  - Posix.Process.execp
  - Posix.Process.exit
  - Posix.Process.fork
  - Posix.Process.kill
  - Posix.Process.pause
  - Posix.Process.waitpid\_nh
  - Posix.Process.waitpid
  - Posix.SysDB.getgrgid
  - Posix.SysDB.getgrnam
  - Posix.SysDB.getpwuid
  - Posix.TTY.TC.drain
  - Posix.TTY.TC.flow
  - Posix.TTY.TC.flush
  - Posix.TTY.TC.getattr
  - Posix.TTY.TC.getpgrp
  - Posix.TTY.TC.sendbreak
  - Posix.TTY.TC.setattr
  - Posix.TTY.TC.setpgrp
  - Unix.kill
  - Unix.reap
  - UnixSock.fromAddr
  - UnixSock.toAddr
-

## 253 RunningOnNetBSD

MLton runs fine on [NetBSD](#).

### 253.1 Installing the correct packages for NetBSD

The NetBSD system installs 3rd party packages by a mechanism known as pkgsrc. This is a tree of Makefiles which when invoked downloads the source code, builds a package and installs it on the system. In order to run MLton on NetBSD, you will have to install several packages for it to work:

- shells/bash
- devel/gmp
- devel/gmake

In order to get graphical call-graphs of profiling information, you will need the additional package

- graphics/graphviz

To build the documentation for MLton, you will need the additional package

- htmldoc.

### 253.2 Tips for compiling and using MLton on NetBSD

MLton can be a memory-hog on computers with little memory. While 640Mb of RAM ought to be enough to self-compile MLton one might want to do some tuning to the NetBSD VM subsystem in order to succeed. The notes presented here is what [JesperLouisAndersen](#) uses for compiling MLton on his laptop.

#### 253.2.1 The NetBSD VM subsystem

NetBSD uses a VM subsystem named [UVM](#). [Tuning the VM system](#) can be done via the `sysctl(8)`-interface with the "VM" MIB set.

#### 253.2.2 Tuning the NetBSD VM subsystem for MLton

MLton uses a lot of anonymous pages when it is running. Thus, we will need to tune up the default of 80 for anonymous pages. Setting

```
sysctl -w vm.anonmax=95
sysctl -w vm.anonmin=50
sysctl -w vm.filemin=2
sysctl -w vm.execmin=2
sysctl -w vm.filemax=4
sysctl -w vm.execmax=4
```

makes it less likely for the VM system to swap out anonymous pages. For a full explanation of the above flags, see the documentation.

The result is that my laptop goes from a MLton compile where it swaps a lot to a MLton compile with no swapping.

## 254 RunningOnOpenBSD

MLton runs fine on [OpenBSD](#).

### 254.1 Known issues

- The `socket.sml` regression test fails. We suspect this is not a bug and is simply due to our test relying on a certain behavior when connecting to a socket that has not yet accepted, which is handled differently on OpenBSD than other platforms. Any help in understanding and resolving this issue is appreciated.
-

## 255 RunningOnPowerPC

MLton runs fine on the PowerPC architecture.

### 255.1 Notes

- When compiling for PowerPC, MLton targets the 32-bit PowerPC architecture.
  - When compiling for PowerPC, MLton doesn't support native code generation (`-codegen native`). Hence, performance is not as good as it might be and compile times are longer. Also, the quality of code generated by `gcc` is important. By default, MLton calls `gcc -O1`. You can change this by calling MLton with `-cc-opt -O2`.
  - On the PowerPC, the [GnuMP](#) library supports multiple ABIs. See the [GnuMP](#) page for more details.
-

## 256 RunningOnPowerPC64

MLton runs fine on the PowerPC64 architecture.

### 256.1 Notes

- When compiling for PowerPC64, MLton targets the 64-bit PowerPC architecture.
  - When compiling for PowerPC64, MLton doesn't support native code generation (`-codegen native`). Hence, performance is not as good as it might be and compile times are longer. Also, the quality of code generated by `gcc` is important. By default, MLton calls `gcc -O1`. You can change this by calling MLton with `-cc-opt -O2`.
  - On the PowerPC64, the [GnuMP](#) library supports multiple ABIs. See the [GnuMP](#) page for more details.
-

## 257 RunningOnS390

MLton runs fine on the S390 architecture.

### 257.1 Notes

- When compiling for S390, MLton doesn't support native code generation (`-codegen native`). Hence, performance is not as good as it might be and compile times are longer. Also, the quality of code generated by `gcc` is important. By default, MLton calls `gcc -O1`. You can change this by calling MLton with `-cc-opt -O2`.
-

## 258 RunningOnSolaris

MLton runs fine on Solaris.

### 258.1 Notes

- You must install the `binutils`, `gcc`, and `make` packages. You can find out how to get these at [sunfreeware.com](http://sunfreeware.com).
- Making the documentation requires that you install `latex` and `dvips`, which are available in the `tetex` package.

### 258.2 Known issues

- Bootstrapping on the [Sparc architecture](#) is so slow as to be impractical (many hours on a 500MHz UltraSparc). For this reason, we strongly recommend building with a [cross compiler](#).

### 258.3 Also see

- [RunningOnAMD64](#)
  - [RunningOnSparc](#)
  - [RunningOnX86](#)
-

## 259 RunningOnSparc

MLton runs fine on the Sparc architecture.

### 259.1 Notes

- When compiling for Sparc, MLton targets the 32-bit Sparc architecture (i.e., Sparc V8).
- When compiling for Sparc, MLton doesn't support native code generation (`-codegen native`). Hence, performance is not as good as it might be and compile times are longer. Also, the quality of code generated by `gcc` is important. By default, MLton calls `gcc -O1`. You can change this by calling MLton with `-cc-opt -O2`. We have seen this speed up some programs by as much as 30%, especially those involving floating point; however, it can also more than double compile times.
- When compiling for Sparc, MLton uses `-align 8` by default. While this speeds up reals, it also may increase object sizes. If your program does not make significant use of reals, you might see a speedup with `-align 4`.

### 259.2 Known issues

- Bootstrapping on the [Sparc architecture](#) is so slow as to be impractical (many hours on a 500MHz UltraSparc). For this reason, we strongly recommend building with a [cross compiler](#).

### 259.3 Also see

- [RunningOnSolaris](#)
-

## 260 RunningOnX86

MLton runs fine on the x86 architecture.

### 260.1 Notes

- On x86, MLton supports native code generation (`-codegen native` or `-codegen x86`).
-

## 261 RunTimeOptions

Executables produced by MLton take command line arguments that control the runtime system. These arguments are optional, and occur before the executable's usual arguments. To use these options, the first argument to the executable must be `@MLton`. The optional arguments then follow, must be terminated by `--`, and are followed by any arguments to the program. The optional arguments are *not* made available to the SML program via `CommandLine.arguments`. For example, a valid call to `hello-world` is:

```
hello-world @MLton gc-summary fixed-heap 10k -- a b c
```

In the above example, `CommandLine.arguments () = ["a", "b", "c"]`.

It is allowed to have a sequence of `@MLton` arguments, as in:

```
hello-world @MLton gc-summary -- @MLton fixed-heap 10k -- a b c
```

Run-time options can also control MLton, as in

```
mlton @MLton fixed-heap 0.5g -- foo.sml
```

### 261.1 Options

- `fixed-heap x{k|K|m|M|g|G}`

Use a fixed size heap of size  $x$ , where  $x$  is a real number and the trailing letter indicates its units.

k or K	1024
m or M	1,048,576
g or G	1,073,741,824

A value of 0 means to use almost all the RAM present on the machine.

The heap size used by `fixed-heap` includes all memory allocated by SML code, including memory for the stack (or stacks, if there are multiple threads). It does not, however, include any memory used for code itself or memory used by C globals, the C stack, or `malloc`.

- `gc-messages`

Print a message at the start and end of every garbage collection.

- `gc-summary`

Print a summary of garbage collection statistics upon program termination.

- `load-world world`

Restart the computation with the file specified by `world`, which must have been created by a call to `MLton.World.save` by the same executable. See [MLtonWorld](#).

- `max-heap x{k|K|m|M|g|G}`

Run the computation with an automatically resized heap that is never larger than  $x$ , where  $x$  is a real number and the trailing letter indicates the units as with `fixed-heap`. The heap size for `max-heap` is accounted for as with `fixed-heap`.

- `may-page-heap {false|true}`

Enable paging the heap to disk when unable to grow the heap to a desired size.

- `no-load-world`

Disable `load-world`. This can be used as an argument to the compiler via `-runtime no-load-world` to create executables that will not load a world. This may be useful to ensure that set-uid executables do not load some strange world.

- `ram-slop x`

Multiply  $x$  by the amount of RAM on the machine to obtain what the runtime views as the amount of RAM it can use. Typically  $x$  is less than 1, and is used to account for space used by other programs running on the same machine.

- `stop`

Causes the runtime to stop processing @MLton arguments once the next `--` is reached. This can be used as an argument to the compiler via `-runtime stop` to create executables that don't process any @MLton arguments.

---

## 262 ScopeInference

Scope inference is an analysis/rewrite pass for the [AST IntermediateLanguage](#), invoked from [Elaborate](#).

### 262.1 Description

This pass adds free type variables to the `val` or `fun` declaration where they are implicitly scoped.

### 262.2 Implementation

```
scope.sig scope.fun
```

### 262.3 Details and Notes

Scope inference determines for each type variable, the declaration where it is bound. Scope inference is a direct implementation of the specification given in section 4.6 of the [Definition](#). Recall that a free occurrence of a type variable `'a` in a declaration `d` is *unguarded* in `d` if `'a` is not part of a smaller declaration. A type variable `'a` is implicitly scoped at `d` if `'a` is unguarded in `d` and `'a` does not occur unguarded in any declaration containing `d`.

The first pass of scope inference walks down the tree and renames all explicitly bound type variables in order to avoid name collisions. It then walks up the tree and adds to each declaration the set of unguarded type variables occurring in that declaration. At this point, if declaration `d` contains an unguarded type variable `'a` and the immediately containing declaration does not contain `'a`, then `'a` is implicitly scoped at `d`. The final pass walks down the tree leaving a `'a` at the a declaration where it is scoped and removing it from all enclosed declarations.

## 263 SelfCompiling

If you want to compile MLton, you must first get the [Sources](#). You can compile with either MLton or SML/NJ, but we strongly recommend using MLton, since it generates a much faster and more robust executable.

### 263.1 Compiling with MLton

To compile with MLton, you need the binary versions of `mlton`, `mllex`, and `mlyacc` that come with the MLton binary package. To be safe, you should use the same version of MLton that you are building. However, older versions may work, as long as they don't go back too far. To build MLton, run `make` from within the root directory of the sources. This will build MLton first with the already installed binary version of MLton and will then rebuild MLton with itself.

First, the Makefile calls `mllex` and `mlyacc` to build the lexer and parser, and then calls `mlton` to compile itself. When making MLton using another version the Makefile automatically uses `mlton-stubs.cm`, which will put in enough stubs to emulate the MLton structure. Once MLton is built, the Makefile will rebuild MLton with itself, this time using `mlton.cm` and the real MLton structure from the [Basis Library](#). This second round of compilation is essential in order to achieve a fast and robust MLton.

Compiling MLton requires at least 512M of actual RAM, and 1G is preferable. If your machine has less than 512M, self-compilation will likely fail, or at least take a very long time due to paging. Even if you have enough memory, there simply may not be enough available, due to memory consumed by other processes. In this case, you may see an `Out of memory` message, or self-compilation may become extremely slow. The only fix is to make sure that enough memory is available.

#### 263.1.1 Possible Errors

- If you have errors running `latex`, you can skip building the documentation by using `make all-no-docs`.
- The C compiler may not be able to find the [GnuMP](#) header file, `gmp.h` leading to an error like the following.

```
platform/darwin.h:26:36: /usr/local/include/gmp.h: No such file or directory
```

The solution is to install (or build) the GnuMP on your machine. If you install it at a different location, put the new path in `runtime/platform/<os>.h`.

- The following error indicates that a binary version of MLton could not be found in your path.

```
../upgrade-basis: mlton: command not found
Error: cannot upgrade basis because the compiler doesn't work
make[3]: *** [upgrade-basis.sml] Error 1
```

You need to have `mlton` in your path to build MLton from source.

During the build process, there are various times that the Makefile's look for a `mlton` in your path and in `src/build/bin`. It is OK if the latter doesn't exist when the build starts; it is the target being built. While not finding `build/bin/mlton` also results in `mlton: command not found` error messages, such errors are benign and will not abort the build. Failure to find a `mlton` in your path will abort the build.

- Mac OS X executables do not seem to like static libraries to have a different path location at runtime compared to when the executable was built. For example, the binary package for Mac OS X unpacks to `/usr`. If you try to install it in `/usr/local` you may get the following errors:

```
/usr/bin/ld: table of contents for archive:
/usr/local/lib/mlton/self/libmlton.a is out of date;
rerun ranlib(1) (can't load from it)
```

Although running `ranlib` seems like the right thing to do, it doesn't actually resolve the problem. Best bet is to install in `/usr` and then either live with this location, or build MLton yourself and install in `/usr/local`.

## 263.2 Compiling with SML/NJ

To compile with SML/NJ, run `make nj-mlton` from within the root directory of the sources. You must use a recent version of SML/NJ. First, the `Makefile` calls `mlex` and `mlyacc` to build the lexer and parser. Then, it calls SML/NJ with the appropriate `sources.cm` file. Building with SML/NJ takes some time (roughly 10 minutes on a 1.6GHz machine). Unless you are doing compiler development and need rapid recompilation, we recommend compiling with MLton.

---

## 264 Serialization

Standard ML does not have built-in support for serialization. Here are papers that describe user-level approaches:

- [Elsman04](#)
- [Kennedy04](#)

The MLton repository also contains an experimental generic programming library (see [README](#)) that includes a pickling (serialization) generic (see [pickle.sig](#)).

---

## 265 ShowBasis

MLton has a flag, `-show-basis <file>`, that causes MLton to pretty print to *file* the basis defined by the input program. For example, if `foo.sml` contains

```
fun f x = x + 1
```

then `mlton -show-basis foo.basis foo.sml` will create `foo.basis` with the following contents.

```
val f: int -> int
```

If you only want to see the basis and do not wish to compile the program, you can call MLton with `-stop tc`.

### 265.1 Displaying signatures

When displaying signatures, MLton prefixes types defined in the signature them with `?.` to distinguish them from types defined in the environment. For example,

```
signature SIG =
 sig
 type t
 val x: t * int -> unit
 end
```

is displayed as

```
signature SIG =
 sig
 type t = ?.t
 val x: (?.t * int) -> unit
 end
```

Notice that `int` occurs without the `?.` prefix.

MLton also uses a canonical name for each type in the signature, and that name is used everywhere for that type, no matter what the input signature looked like. For example:

```
signature SIG =
 sig
 type t
 type u = t
 val x: t
 val y: u
 end
```

is displayed as

```
signature SIG =
 sig
 type t = ?.t
 type u = ?.t
 val x: ?.t
 val y: ?.t
 end
```

Canonical names are always relative to the "top" of the signature, even when used in nested substructures. For example:

```
signature S =
 sig
 type t
 val w: t
 structure U:
 sig
 type u
 val x: t
 val y: u
 end
 val z: U.u
 end
```

is displayed as

```
signature S =
 sig
 type t = ?.t
 val w: ?.t
 val z: ?.U.u
 structure U:
 sig
 type u = ?.U.u
 val x: ?.t
 val y: ?.U.u
 end
 end
```

## 265.2 Displaying structures

When displaying structures, MLton uses signature constraints wherever possible, combined with `where type` clauses to specify the meanings of the types defined within the signature. For example:

```
signature SIG =
 sig
 type t
 val x: t
 end
structure S: SIG =
 struct
 type t = int
 val x = 13
 end
structure S2:> SIG = S
```

is displayed as

```
structure S: SIG
 where type t = int
structure S2: SIG
 where type t = S2.t
signature SIG =
 sig
 type t = ?.t
 val x: ?.t
 end
```

## 266 ShowProf

If an executable is compiled for [profiling](#), then it accepts a special command-line runtime system argument, `show-prof`, that outputs information about the source functions that are profiled. Normally, this information is used by `mlprof`. This page documents the `show-prof` output format, and is intended for those working on the profiler internals.

The `show-prof` output is ASCII, and consists of a sequence of lines.

- The magic number of the executable.
- The number of source names in the executable.
- A line for each source name giving the name of the function, a tab, the filename of the file containing the function, a colon, a space, and the line number that the function starts on in that file.
- The number of (split) source functions.
- A line for each (split) source function, where each line consists of a source-name index (into the array of source names) and a successors index (into the array of split-source sequences, defined below).
- The number of split-source sequences.
- A line for each split-source sequence, where each line is a space separated list of (split) source functions.

The latter two arrays, split sources and split-source sequences, define a directed graph, which is the call-graph of the program.

---

## 267 Shrink

[Shrink](#) is a rewrite pass for the [SSA](#) and [SSA2 IntermediateLanguages](#), invoked from every optimization pass (see [SSASimplify](#) and [SSA2Simplify](#)).

### 267.1 Description

This pass implements a whole family of compile-time reductions, like:

- $\#1(a, b) \Rightarrow a$
- $\text{case } C \ x \ \text{of } C \ y \Rightarrow e \Rightarrow \text{let } y = x \ \text{in } e$
- constant folding, copy propagation
- eta blocks
- tuple reconstruction elimination

### 267.2 Implementation

- [shrink.sig](#)
- [shrink.fun](#)
- [shrink2.sig](#)
- [shrink2.fun](#)

### 267.3 Details and Notes

The [Shrink](#) pass is run after every [SSA](#) and [SSA2](#) optimization pass.

The [Shrink](#) implementation also includes functions to eliminate unreachable blocks from a [SSA](#) or [SSA2](#) program or function. The [Shrink](#) pass does not guarantee to eliminate all unreachable blocks. Doing so would unduly complicate the implementation, and it is almost always the case that all unreachable blocks are eliminated. However, a small number of optimization passes require that the input have no unreachable blocks (essentially, when the analysis works on the control flow graph and the rewrite iterates on the vector of blocks). These passes explicitly call `eliminateDeadBlocks`.

The [Shrink](#) pass has a special case to turn a non-tail call where the continuation and handler only do `Profile` statements into a tail call where the `Profile` statements precede the tail call.

## 268 SimplifyTypes

`SimplifyTypes` is an optimization pass for the [SSA IntermediateLanguage](#), invoked from `SSASimplify`.

### 268.1 Description

This pass computes a "cardinality" of each datatype, which is an abstraction of the number of values of the datatype.

- `Zero` means the datatype has no values (except for bottom).
- `One` means the datatype has one value (except for bottom).
- `Many` means the datatype has many values.

This pass removes all datatypes whose cardinality is `Zero` or `One` and removes:

- components of tuples
- function args
- constructor args

which are such datatypes.

This pass marks constructors as one of:

- `Useless`: it never appears in a `ConApp`.
- `Transparent`: it is the only variant in its datatype and its argument type does not contain any uses of `array` or `vector`.
- `Useful`: otherwise

This pass also removes `Useless` and `Transparent` constructors.

### 268.2 Implementation

- `simplify-types.fun`

### 268.3 Details and Notes

This pass must happen before polymorphic equality is implemented because

- it will make polymorphic equality faster because some types are simpler
- it removes uses of polymorphic equality that must return true

We must keep track of `Transparent` constructors whose argument type uses `array` because of datatypes like the following:

```
datatype t = T of t array
```

Such a datatype has `Cardinality.Many`, but we cannot eliminate the datatype and replace the lhs by the rhs, i.e. we must keep the circularity around.

Must do similar things for `vectors`.

Also, to eliminate as many `Transparent` constructors as possible, for something like the following,

```
datatype t = T of u array
and u = U of t vector
```

we (arbitrarily) expand one of the datatypes first. The result will be something like

```
datatype u = U of u array array
```

where all uses of `t` are replaced by `u array`.

## 269 SML3d

The [SML3d Project](#) is a collection of libraries to support 3D graphics programming using Standard ML and the [OpenGL](#) graphics API. It currently requires the MLton implementation of SML and is supported on Linux, Mac OS X, and Microsoft Windows. There is also support for [OpenCL](#).

---

## 270 SMLNET

**SML.NET** is a [Standard ML implementation](#) that targets the .NET Common Language Runtime.

SML.NET is based on the [MLj](#) compiler.

### 270.1 Also see

- [BentonEtAl04](#)
-

## 271 SMLNJ

[SML/NJ](#) is a [Standard ML implementation](#). It is a native code compiler that runs on a variety of platforms and has a number of libraries and tools.

We maintain a list of SML/NJ's [deviations](#) from [The Definition of Standard ML](#).

MLton has support for some features of SML/NJ in order to ease porting between MLton and SML/NJ.

- [CompilationManager](#) (CM)
  - [LineDirectives](#)
  - [SMLofNJStructure](#)
  - [UnsafeStructure](#)
-

## 272 SMLNJDeviations

Here are some deviations of SML/NJ from [The Definition of Standard ML \(Revised\)](#). Some of these are documented in the [SML '97 Conversion Guide](#). Since MLton does not deviate from the Definition, you should look here if you are having trouble porting a program from MLton to SML/NJ or vice versa. If you discover other deviations of SML/NJ that aren't listed here, please send mail to [MLton-devel@mlton.org](mailto:MLton-devel@mlton.org).

- SML/NJ allows spaces in long identifiers, as in `S . x`. Section 2.5 of the Definition implies that `S . x` should be treated as three separate lexical items.
- SML/NJ allows `op` to appear in `val` specifications:

```
signature FOO = sig
 val op + : int * int -> int
end
```

The grammar on page 14 of the Definition does not allow it. Recent versions of SML/NJ do give a warning.

- SML/NJ rejects

```
(op *)
```

as an unmatched close comment.

- SML/NJ allows `=` to be rebound by the declaration:

```
val op = = 13
```

This is explicitly forbidden on page 5 of the Definition. Recent versions of SML/NJ do give a warning.

- SML/NJ allows rebinding `true`, `false`, `nil`, `::`, and `ref` by the declarations:

```
fun true () = ()
fun false () = ()
fun nil () = ()
fun op :: () = ()
fun ref () = ()
```

This is explicitly forbidden on page 9 of the Definition.

- SML/NJ extends the syntax of the language to allow vector expressions and patterns like the following:

```
val v = #[1,2,3]
val #[x,y,z] = v
```

- SML/NJ extends the syntax of the language to allow *or patterns* like the following:

```
datatype foo = Foo of int | Bar of int
val (Foo x | Bar x) = Foo 13
```

- SML/NJ allows higher-order functors, that is, functors can be components of structures and can be passed as functor arguments and returned as functor results. As a consequence, SML/NJ allows abbreviated functor definitions, as in the following:

```
signature S =
 sig
 type t
 val x: t
 end
functor F (structure A: S): S =
 struct
 type t = A.t * A.t
 val x = (A.x, A.x)
 end
functor G = F
```

- SML/NJ extends the syntax of the language to allow `functor` and `signature` declarations to occur within the scope of `local` and `structure` declarations.
- SML/NJ allows duplicate type specifications in signatures when the duplicates are introduced by `include`, as in the following:

```
signature SIG1 =
 sig
 type t
 type u
 end
signature SIG2 =
 sig
 type t
 type v
 end
signature SIG =
 sig
 include SIG1
 include SIG2
 end
```

This is disallowed by rule 77 of the Definition.

- SML/NJ allows sharing constraints between type abbreviations in signatures, as in the following:

```
signature SIG =
 sig
 type t = int * int
 type u = int * int
 sharing type t = u
 end
```

These are disallowed by rule 78 of the Definition. Recent versions of SML/NJ correctly disallow sharing constraints between type abbreviations in signatures.

- SML/NJ disallows multiple `where` type specifications of the same type name, as in the following

```
signature S =
 sig
 type t
 type u = t
 end
where type u = int
```

This is allowed by rule 64 of the Definition.

- SML/NJ allows `and in` sharing specs in signatures, as in

```
signature S =
 sig
 type t
 type u
 type v
 sharing type t = u
 and type u = v
 end
```

- SML/NJ does not expand the `withtype` derived form as described by the Definition. According to page 55 of the Definition, the type bindings of a `withtype` declaration are substituted simultaneously in the connected datatype. Consider the following program.

```

type u = real ;
datatype a =
 A of t
 | B of u
withtype u = int
and t = u

```

According to the Definition, it should be expanded to the following.

```

type u = real ;
datatype a =
 A of u
 | B of int ;
type u = int
and t = u

```

However, SML/NJ expands `withtype` bindings sequentially, meaning that earlier bindings are expanded within later ones. Hence, the above program is expanded to the following.

```

type u = real ;
datatype a =
 A of int
 | B of int ;
type u = int
type t = int

```

- SML/NJ allows `withtype` specifications in signatures.
- SML/NJ allows a `where` structure specification that is similar to a `where` type specification. For example:

```

structure S = struct type t = int end
signature SIG =
 sig
 structure T : sig type t end
 end where T = S

```

This is equivalent to:

```

structure S = struct type t = int end
signature SIG =
 sig
 structure T : sig type t end
 end where type T.t = S.t

```

SML/NJ also allows a definitional structure specification that is similar to a definitional type specification. For example:

```

structure S = struct type t = int end
signature SIG =
 sig
 structure T : sig type t end = S
 end

```

This is equivalent to the previous examples and to:

```

structure S = struct type t = int end
signature SIG =
 sig
 structure T : sig type t end where type t = S.t
 end

```

- SML/NJ disallows binding non-datatypes with datatype replication. For example, it rejects the following program that should be allowed according to the Definition.

```
type ('a, 'b) t = 'a * 'b
datatype u = datatype t
```

This idiom can be useful when one wants to rename a type without rewriting all the type arguments. For example, the above would have to be written in SML/NJ as follows.

```
type ('a, 'b) t = 'a * 'b
type ('a, 'b) u = ('a, 'b) t
```

- SML/NJ disallows sharing a structure with one of its substructures. For example, SML/NJ disallows the following.

```
signature SIG =
 sig
 structure S:
 sig
 type t
 structure T: sig type t end
 end
 sharing S = S.T
 end
```

This signature is allowed by the Definition.

- SML/NJ disallows polymorphic generalization of refutable patterns. For example, SML/NJ disallows the following.

```
val [x] = [[]]
val _ = (1 :: x, "one" :: x)
```

Recent versions of SML/NJ correctly allow polymorphic generalization of refutable patterns.

- SML/NJ uses an overly restrictive context for type inference. For example, SML/NJ rejects both of the following.

```
structure S =
 struct
 val z = (fn x => x) []
 val y = z :: [true] :: nil
 end
```

```
structure S : sig val z : bool list end =
 struct
 val z = (fn x => x) []
 end
```

These structures are allowed by the Definition.

## 272.1 Deviations from the Basis Library Specification

Here are some deviations of SML/NJ from the [Basis Library specification](#).

- SML/NJ exposes the equality of the `vector` type in structures such as `Word8Vector` that abstractly match `MONO_VECTOR`, which says `type vector`, not `eqtype vector`. So, for example, SML/NJ accepts the following program:

```
fun f (v: Word8Vector.vector) = v = v
```

- SML/NJ exposes the equality property of the type `status` in `OS.Process`. This means that programs which directly compare two values of type `status` will work with SML/NJ but not MLton.
- Under SML/NJ on Windows, `OS.Path.validVolume` incorrectly considers absolute empty volumes to be valid. In other words, when the expression

```
OS.Path.validVolume { isAbs = true, vol = "" }
```

is evaluated by SML/NJ on Windows, the result is `true`. MLton, on the other hand, correctly follows the Basis Library Specification, which states that on Windows, `OS.Path.validVolume` should return `false` whenever `isAbs = true` and `vol = ""`.

This incorrect behavior causes other `OS.Path` functions to behave differently. For example, when the expression

```
OS.Path.toString (OS.Path.fromString "\\usr\\local")
```

is evaluated by SML/NJ on Windows, the result is `"\\usr\\local"`, whereas under MLton on Windows, evaluating this expression (correctly) causes an `OS.Path.Path` exception to be raised.

---

## 273 SMLNJLibrary

The **SML/NJ Library** is a collection of libraries that are distributed with SML/NJ. Due to differences between SML/NJ and MLton, these libraries will not work out-of-the box with MLton.

As of 20130706, MLton includes a port of the SML/NJ Library synchronized with SML/NJ version 110.76.

### 273.1 Usage

- You can import a sub-library of the SML/NJ Library into an MLB file with:

MLB file	Description
<code>\$(SML_LIB)/smlnj-lib/Util/smlnj-lib.mlb</code>	Various utility modules, included collections, simple formatting, ...
<code>\$(SML_LIB)/smlnj-lib/Controls/controls-lib.mlb</code>	A library for managing control flags in an application.
<code>\$(SML_LIB)/smlnj-lib/HashCons/hash-cons-lib.mlb</code>	Support for implementing hash-consed data structures.
<code>\$(SML_LIB)/smlnj-lib/HTML/html-lib.mlb</code>	HTML 3.2 parsing and pretty-printing library.
<code>\$(SML_LIB)/smlnj-lib/HTML4/html4-lib.mlb</code>	HTML 4.01 parsing and pretty-printing library.
<code>\$(SML_LIB)/smlnj-lib/INet/inet-lib.mlb</code>	Networking utilities; supported on both Unix and Windows systems.
<code>\$(SML_LIB)/smlnj-lib/JSON/json-lib.mlb</code>	JavaScript Object Notation (JSON) reading and writing library.
<code>\$(SML_LIB)/smlnj-lib/PP/pp-lib.mlb</code>	Pretty-printing library.
<code>\$(SML_LIB)/smlnj-lib/Reactive/reactive-lib.mlb</code>	Reactive scripting library.
<code>\$(SML_LIB)/smlnj-lib/RegExp/regexp-lib.mlb</code>	Regular expression library.
<code>\$(SML_LIB)/smlnj-lib/SExp/sexp-lib.mlb</code>	S-expression library.
<code>\$(SML_LIB)/smlnj-lib/Unix/unix-lib.mlb</code>	Utilities for Unix-based operating systems.

- If you are porting a project from SML/NJ's **CompilationManager** to MLton's **ML Basis system** using `cm2mlb`, note that the following maps are included by default:

```
SMLNJ Library
$SMLNJ-LIB $(SML_LIB)/smlnj-lib
$smlnj-lib.cm $(SML_LIB)/smlnj-lib/Util
$controls-lib.cm $(SML_LIB)/smlnj-lib/Controls
$hash-cons-lib.cm $(SML_LIB)/smlnj-lib/HashCons
$html-lib.cm $(SML_LIB)/smlnj-lib/HTML
$html4-lib.cm $(SML_LIB)/smlnj-lib/HTML4
$inet-lib.cm $(SML_LIB)/smlnj-lib/INet
$json-lib.cm $(SML_LIB)/smlnj-lib/JSON
$pp-lib.cm $(SML_LIB)/smlnj-lib/PP
$reactive-lib.cm $(SML_LIB)/smlnj-lib/Reactive
$regexp-lib.cm $(SML_LIB)/smlnj-lib/RegExp
$sexp-lib.cm $(SML_LIB)/smlnj-lib/SExp
$unix-lib.cm $(SML_LIB)/smlnj-lib/Unix
```

This will automatically convert a `$/smlnj-lib.cm` import in an input `.cm` file into a `$(SML_LIB)/smlnj-lib/Util/smlnj-lib.mlb` import in the output `.mlb` file.

### 273.2 Details

The following changes were made to the SML/NJ Library, in addition to deriving the `.mlb` files from the `.cm` files:

- HTML/html-attrs-fn.sml (modified): Rewrote use of or-patterns.
- HTML/html-elements-fn.sml (modified): Rewrote use of or-patterns.

- `HTML4/pp-init.sml` (added): Implements structure `PrettyPrint` using the SML/NJ PP Library. This implementation is taken from the SML/NJ compiler source, since the SML/NJ HTML4 Library used the structure `PrettyPrint` provided by the SML/NJ compiler itself.
- `Util/base64.sml` (modified): Rewrote use of `Unsafe.CharVector.create` and `Unsafe.CharVector.update`; MLton assumes that vectors are immutable.
- `Util/bit-array.sml` (modified): The computation of the `maxLen` is given by:

```
val maxLen = 8*Word8Array.maxLen
```

This is fine in SML/NJ where `Word8Array.maxLen` is 16777215, but in MLton, `Word8Array.maxLen` is equal to `valOf(Int.maxInt)`, so the computation overflows. To accommodate both SML/NJ and MLton, the computation is replaced by

```
val maxLen = (8*Word8Array.maxLen) handle Overflow => Word8Array.maxLen
```

- `Util/engine.mlton.sml` (added, not exported): Implements structure `Engine`, providing time-limited, resumable computations using [MLtonThread](#), [MLtonSignal](#), and [MLtonItimer](#).
- `Util/graph-scc-fn.sml` (modified): Rewrote use of `where` structure specification.
- `Util/redblack-map-fn.sml` (modified): Rewrote use of `where` structure specification.
- `Util/redblack-set-fn.sml` (modified): Rewrote use of `where` structure specification.
- `Util/time-limit.mlb` (added): Exports structure `TimeLimit`, which is *not* exported by `smlnj-lib.mlb`. Since MLton is very conservative in the presence of threads and signals, program performance may be adversely affected by unnecessarily including structure `TimeLimit`.
- `Util/time-limit.mlton.sml` (added): Implements structure `TimeLimit` using structure `Engine`. The SML/NJ implementation of structure `TimeLimit` uses SML/NJ's first-class continuations, signals, and interval timer.

### 273.3 Patch

- [smlnj-lib.patch](#)

## 274 SMLofNJStructure

```
signature SML_OF_NJ =
 sig
 structure Cont:
 sig
 type 'a cont
 val callcc: ('a cont -> 'a) -> 'a
 val isolate: ('a -> unit) -> 'a cont
 val throw: 'a cont -> 'a -> 'b
 end
 structure SysInfo:
 sig
 exception UNKNOWN
 datatype os_kind = BEOS | MACOS | OS2 | UNIX | WIN32

 val getHostArch: unit -> string
 val getOSKind: unit -> os_kind
 val getOSName: unit -> string
 end

 val exnHistory: exn -> string list
 val exportFn: string * (string * string list -> OS.Process.status) -> unit
 val exportML: string -> bool
 val getAllArgs: unit -> string list
 val getArgs: unit -> string list
 val getCmdName: unit -> string
 end
```

SMLofNJ implements a subset of the structure of the same name provided in [Standard ML of New Jersey](#). It is included to make it easier to port programs between the two systems. The semantics of these functions may be different than in SML/NJ.

- `structure Cont`  
implements continuations.
- `SysInfo.getHostArch ()`  
returns the string for the architecture.
- `SysInfo.getOSKind`  
returns the OS kind.
- `SysInfo.getOSName ()`  
returns the string for the host.
- `exnHistory`  
the same as `MLton.Exn.history`.
- `getCmdName ()`  
the same as `CommandLine.name ()`.
- `getArgs ()`  
the same as `CommandLine.arguments ()`.
- `getAllArgs ()`  
the same as `getCmdName () :: getArgs ()`.
- `exportFn f`  
saves the state of the computation to a file that will apply `f` to the command-line arguments upon restart.

- `exportML f`  
saves the state of the computation to file `f` and continue. Returns `true` in the restarted computation and `false` in the continuing computation.
-

## 275 SMLSharp

SML# is an [implementation](#) of an extension of SML.

It includes some [generally useful SML tools](#) including a pretty printer generator, a document generator, and a regression testing framework, and [scripting library](#).

---

## 276 Sources

We maintain our sources with [Git](#). You can [view them on the web](#) or access them with a git client.

Anonymous read-only access is available via

```
https://github.com/MLton/mlton.git
```

or

```
git://github.com/MLton/mlton.git
```

### 276.1 Commit email

All commits are sent to [MLton-commit@mlton.org](mailto:MLton-commit@mlton.org) ([subscribe](#), [archive](#), [archive](#)) which is a read-only mailing list for commit emails. Discussion should go to [MLton-devel@mlton.org](mailto:MLton-devel@mlton.org).

### 276.2 Changelog

See the [changelog](#) for a list of changes and bug fixes.

### 276.3 Subversion

Prior to 20130308, we used [Subversion](#).

### 276.4 CVS

Prior to 20050730, we used [CVS](#).

---

## 277 SpaceSafety

Informally, space safety is a property of a language implementation that asymptotically bounds the space used by a running program.

### 277.1 Also see

- Chapter 12 of [Appel92](#)
  - [Clinger98](#)
-

## 278 SSA

SSA is an [IntermediateLanguage](#), translated from [SXML](#) by [ClosureConvert](#), optimized by [SSASimplify](#), and translated by [ToSSA2](#) to [SSA2](#).

### 278.1 Description

SSA is a [FirstOrder, SimplyTyped IntermediateLanguage](#). It is the main [IntermediateLanguage](#) used for optimizations.

An SSA program consists of a collection of datatype declarations, a sequence of global statements, and a collection of functions, along with a distinguished "main" function. Each function consists of a collection of basic blocks, where each basic block is a sequence of statements ending with some control transfer.

### 278.2 Implementation

- [ssa.sig](#)
- [ssa.fun](#)
- [ssa-tree.sig](#)
- [ssa-tree.fun](#)

### 278.3 Type Checking

Type checking ([type-check.sig](#), [type-check.fun](#)) of a SSA program verifies the following:

- no duplicate definitions (tycons, cons, vars, labels, funcs)
- no out of scope references (tycons, cons, vars, labels, funcs)
- variable definitions dominate variable uses
- case transfers are exhaustive and irredundant
- `Enter/Leave` profile statements match
- "traditional" well-typedness

### 278.4 Details and Notes

SSA is an abbreviation for Static Single Assignment.

For some initial design discussion, see the thread at:

- <http://mlton.org/pipermail/mlton/2001-August/019689.html>

For a retrospective, see the thread at:

- <http://mlton.org/pipermail/mlton/2007-February/029597.html>

## 279 SSA2

SSA2 is an [IntermediateLanguage](#), translated from SSA by [ToSSA2](#), optimized by [SSA2Simplify](#), and translated by [ToRSSA](#) to [RSSA](#).

### 279.1 Description

SSA2 is a [FirstOrder](#), [SimplyTyped IntermediateLanguage](#), a slight variant of the [SSA IntermediateLanguage](#),

Like SSA, a SSA program consists of a collection of datatype declarations, a sequence of global statements, and a collection of functions, along with a distinguished "main" function. Each function consists of a collection of basic blocks, where each basic block is a sequence of statements ending with some control transfer.

Unlike SSA, SSA2 includes mutable fields in objects and makes the vector type constructor n-ary instead of unary. This allows optimizations like [RefFlatten](#) and [DeepFlatten](#) to be expressed.

### 279.2 Implementation

- [ssa2.sig](#)
- [ssa2.fun](#)
- [ssa-tree2.sig](#)
- [ssa-tree2.fun](#)

### 279.3 Type Checking

Type checking ([type-check2.sig](#), [type-check2.fun](#)) of a SSA2 program verifies the following:

- no duplicate definitions (tycons, cons, vars, labels, funcs)
- no out of scope references (tycons, cons, vars, labels, funcs)
- variable definitions dominate variable uses
- case transfers are exhaustive and irredundant
- *Enter/Leave* profile statements match
- "traditional" well-typedness

### 279.4 Details and Notes

SSA is an abbreviation for Static Single Assignment.

---

## 280 SSA2Simplify

The optimization passes for the [SSA2 IntermediateLanguage](#) are collected and controlled by the `Simplify2` functor (`simplify2.sig`, `simplify2.fun`).

The following optimization passes are implemented:

- [DeepFlatten](#)
- [RefFlatten](#)
- [RemoveUnused](#)
- [Zone](#)

There are additional analysis and rewrite passes that augment many of the other optimization passes:

- [Restore](#)
- [Shrink](#)

The optimization passes can be controlled from the command-line by the options

- `-diag-pass <pass>` — keep diagnostic info for pass
  - `-drop-pass <pass>` — omit optimization pass
  - `-keep-pass <pass>` — keep the results of pass
  - `-loop-passes <n>` — loop optimization passes
  - `-ssa2-passes <passes>` — ssa optimization passes
-

## 281 SSASimplify

The optimization passes for the [SSA IntermediateLanguage](#) are collected and controlled by the `Simplify` functor (`simplify.sig`, `simplify.fun`).

The following optimization passes are implemented:

- [CombineConversions](#)
- [CommonArg](#)
- [CommonBlock](#)
- [CommonSubexp](#)
- [ConstantPropagation](#)
- [Contify](#)
- [Flatten](#)
- [Inline](#)
- [IntroduceLoops](#)
- [KnownCase](#)
- [LocalFlatten](#)
- [LocalRef](#)
- [LoopInvariant](#)
- [Redundant](#)
- [RedundantTests](#)
- [RemoveUnused](#)
- [SimplifyTypes](#)
- [Useless](#)

The following implementation passes are implemented:

- [PolyEqual](#)
- [PolyHash](#)

There are additional analysis and rewrite passes that augment many of the other optimization passes:

- [Multi](#)
- [Restore](#)
- [Shrink](#)

The optimization passes can be controlled from the command-line by the options:

- `-diag-pass <pass>` — keep diagnostic info for pass
  - `-drop-pass <pass>` — omit optimization pass
  - `-keep-pass <pass>` — keep the results of pass
  - `-loop-passes <n>` — loop optimization passes
  - `-ssa-passes <passes>` — ssa optimization passes
-

## 282 Stabilizers

### 282.1 Installation

- Stabilizers currently require the MLton sources, this should be fixed by the next release

### 282.2 License

- Stabilizers are released under the MLton License

### 282.3 Instructions

- Download and build a source copy of MLton
- Extract the tar.gz file attached to this page
- Some examples are provided in the "examples/" sub directory, more examples will be added to this page in the following week

### 282.4 Bug reports / Suggestions

- Please send any errors you encounter to schatzp and lziarek at cs.purdue.edu
- We are looking to expand the usability of stabilizers
- Please send any suggestions and desired functionality to the above email addresses

### 282.5 Note

- This is an alpha release. We expect to have another release shortly with added functionality soon
- More documentation, such as signatures and descriptions of functionality, will be forthcoming

### 282.6 Documentation

```
signature STABLE =
 sig
 type checkpoint

 val stable: ('a -> 'b) -> ('a -> 'b)
 val stabilize: unit -> 'a

 val stableCP: (('a -> 'b) * (unit -> unit)) ->
 (('a -> 'b) * checkpoint)
 val stabilizeCP: checkpoint -> unit

 val unmonitoredAssign: ('a ref * 'a) -> unit
 val monitoredAssign: ('a ref * 'a) -> unit
 end
```

Stable provides functions to manage stable sections.

- type checkpoint  
handle used to stabilize contexts other than the current one.

- `stable f`  
returns a function identical to `f` that will execute within a stable section.
- `stabilize ()`  
unrolls the effects made up to the current context to at least the nearest enclosing *stable* section. These effects may have propagated to other threads, so all affected threads are returned to a globally consistent previous state. The return is undefined because control cannot resume after `stabilize` is called.
- `stableCP (f, comp)`  
returns a function `f'` and checkpoint tag `cp`. Function `f'` is identical to `f` but when applied will execute within a stable section. `comp` will be executed if `f'` is later stabilized. `cp` is used by `stabilizeCP` to stabilize a given checkpoint.
- `stabilizeCP cp`  
same as `stabilize` except that the (possibly current) checkpoint to stabilize is provided.
- `unmonitoredAssign (r, v)`  
standard assignment (`:=`). The version of CML distributed rebinds `:=` to a monitored version so interesting effects can be recorded.
- `monitoredAssign (r, v)`  
the assignment operator that should be used in programs that use stabilizers. `:=` is rebound to this by including CML.

## 282.7 Download

- [stabilizers\\_alpha\\_2006-10-09.tar.gz](#)

## 282.8 Also see

- [ZiarekEtAl06](#)
-

## 283 StandardML

Standard ML (SML) is a programming language that combines excellent support for rapid prototyping, modularity, and development of large programs, with performance approaching that of C.

### 283.1 SML Resources

- [Tutorials](#)
- [Books](#)
- [Implementations](#)

### 283.2 Aspects of SML

- [DefineTypeBeforeUse](#)
- [EqualityType](#)
- [EqualityTypeVariable](#)
- [GenerativeDatatype](#)
- [GenerativeException](#)
- [Identifier](#)
- [OperatorPrecedence](#)
- [Overloading](#)
- [PolymorphicEquality](#)
- [TypeVariableScope](#)
- [ValueRestriction](#)

### 283.3 Using SML

- [Fixpoints](#)
  - [ForLoops](#)
  - [FunctionalRecordUpdate](#)
  - [InfixingOperators](#)
  - [Lazy](#)
  - [ObjectOrientedProgramming](#)
  - [OptionalArguments](#)
  - [Printf](#)
  - [PropertyList](#)
  - [ReturnStatement](#)
  - [Serialization](#)
  - [StandardMLGotchas](#)
  - [StyleGuide](#)
  - [TipsForWritingConciseSML](#)
  - [UniversalType](#)
-

## 283.4 Programming in SML

- [Emacs](#)
- [Enscript](#)
- [Pygments](#)

## 283.5 Notes

- [History of SML](#)
- [Regions](#)

## 283.6 Related Languages

- [Alice](#)
  - [F#](#)
  - [OCaml](#)
-

## **284 StandardMLBooks**

### **284.1 Introductory Books**

- [Elements of ML Programming](#)
- [ML For the Working Programmer](#)
- [Introduction to Programming using SML](#)
- [The Little MLer](#)

### **284.2 Applications**

- [Unix System Programming with Standard ML](#)

### **284.3 Reference Books**

- [The Standard ML Basis Library](#)
- [The Definition of Standard ML \(Revised\)](#)

### **284.4 Related Topics**

- [Concurrent Programming in ML](#)
  - [Purely Functional Data Structures](#)
-

## 285 StandardMLGotchas

This page contains brief explanations of some recurring sources of confusion and problems that SML newbies encounter.

Many confusions about the syntax of SML seem to arise from the use of an interactive REPL (Read-Eval Print Loop) while trying to learn the basics of the language. While writing your first SML programs, you should keep the source code of your programs in a form that is accepted by an SML compiler as a whole.

### 285.1 The and keyword

It is a common mistake to misuse the `and` keyword or to not know how to introduce mutually recursive definitions. The purpose of the `and` keyword is to introduce mutually recursive definitions of functions and datatypes. For example,

```
fun isEven 0w0 = true
 | isEven 0w1 = false
 | isEven n = isOdd (n-0w1)
and isOdd 0w0 = false
 | isOdd 0w1 = true
 | isOdd n = isEven (n-0w1)
```

and

```
datatype decl = VAL of id * pat * expr
 (* | ... *)
 and expr = LET of decl * expr
 (* | ... *)
```

You can also use `and` as a shorthand in a couple of other places, but it is not necessary.

### 285.2 Constructed patterns

It is a common mistake to forget to parenthesize constructed patterns in `fun` bindings. Consider the following invalid definition:

```
fun length nil = 0
 | length h :: t = 1 + length t
```

#### The pattern `h`

`t`` needs to be parenthesized:

```
fun length nil = 0
 | length (h :: t) = 1 + length t
```

The parentheses are needed, because a `fun` definition may have multiple consecutive constructed patterns through currying.

The same applies to nonfix constructors. For example, the parentheses in

```
fun valOf NONE = raise Option
 | valOf (SOME x) = x
```

are required. However, the outermost constructed pattern in a `fn` or `case` expression need not be parenthesized, because in those cases there is always just one constructed pattern. So, both

```
val valOf = fn NONE => raise Option
 | SOME x => x
```

and

```
fun valOf x = case x of
 NONE => raise Option
 | SOME x => x
```

are fine.

### 285.3 Declarations and expressions

It is a common mistake to confuse expressions and declarations. Normally an SML source file should only contain declarations. The following are declarations:

```
datatype dt = ...
fun f ... = ...
functor Fn (...) = ...
infix ...
infixr ...
local ... in ... end
nonfix ...
open ...
signature SIG = ...
structure Struct = ...
type t = ...
val v = ...
```

Note that

```
let ... in ... end
```

isn't a declaration.

To specify a side-effecting computation in a source file, you can write:

```
val () = ...
```

### 285.4 Equality types

SML has a fairly intricate built-in notion of equality. See [EqualityType](#) and [EqualityTypeVariable](#) for a thorough discussion.

### 285.5 Nested cases

It is a common mistake to write nested case expressions without the necessary parentheses. See [UnresolvedBugs](#) for a discussion.

### 285.6 (op \*)

It used to be a common mistake to parenthesize `op *` as `(op *)`. Before SML'97, `*` was considered a comment terminator in SML and caused a syntax error. At the time of writing, [SML/NJ](#) still rejects the code. An extra space may be used for portability: `(op * )`. However, parenthesizing `op` is redundant, even though it is a widely used convention.

### 285.7 Overloading

A number of standard operators (`+`, `-`, `~`, `*`, `<`, `>`, ...) and numeric constants are overloaded for some of the numeric types (`int`, `real`, `word`). It is a common surprise that definitions using overloaded operators such as

```
fun min (x, y) = if y < x then y else x
```

are not overloaded themselves. SML doesn't really support (user-defined) overloading or other forms of ad hoc polymorphism. In cases such as the above where the context doesn't resolve the overloading, expressions using overloaded operators or constants get assigned a default type. The above definition gets the type

```
val min : int * int -> int
```

See [Overloading](#) and [TypeIndexedValues](#) for further discussion.

## 285.8 Semicolons

It is a common mistake to use redundant semicolons in SML code. This is probably caused by the fact that in an SML REPL, a semicolon (and enter) is used to signal the REPL that it should evaluate the preceding chunk of code as a unit. In SML source files, semicolons are really needed in only two places. Namely, in expressions of the form

```
(exp ; ... ; exp)
```

and

```
let ... in exp ; ... ; exp end
```

Note that semicolons act as expression (or declaration) separators rather than as terminators.

## 285.9 Stale bindings

### 285.10 Unresolved records

### 285.11 Value restriction

See [ValueRestriction](#).

### 285.12 Type Variable Scope

See [TypeVariableScope](#).

---

## 286 StandardMLHistory

[Standard ML](#) grew out of [ML](#) in the early 1980s.

For an excellent overview of SML's history, see Appendix F of the [Definition](#).

For an overview if its history before 1982, see [How ML Evolved](#).

---

## 287 StandardMLImplementations

There are a number of implementations of [Standard ML](#), from interpreters, to byte-code compilers, to incremental compilers, to whole-program compilers.

- [Alice ML](#)
- [HaMLet](#)
- [ML Kit](#)
- [MLton](#)
- [Moscow ML](#)
- [Poly/ML](#)
- [SML#](#)
- [SML/NJ](#)
- [SML.NET](#)
- [TILT](#)

### 287.1 Not Actively Maintained

- [Edinburgh ML](#)
  - [MLj](#)
  - [MLWorks](#)
  - [Poplog](#)
  - [TIL](#)
-

## 288 StandardMLPortability

Technically, SML'97 as defined in the [Definition](#) requires only a minimal initial basis, which, while including the types `int`, `real`, `char`, and `string`, need have no operations on those base types. Hence, the only observable output of an SML'97 program is termination or raising an exception. Most SML compilers should agree there, to the degree each agrees with the Definition. See [UnresolvedBugs](#) for MLton's very few corner cases.

Realistically, a program needs to make use of the [Basis Library](#). Within the Basis Library, there are numerous places where the behavior is implementation dependent. For a trivial example:

```
val _ = valOf (Int.maxInt)
```

may either raise the `Option` exception (if `Int.maxInt == NONE`) or may terminate normally. The default `Int/Real/Word` sizes are the biggest implementation dependent aspect; so, one implementation may raise `Overflow` while another can accommodate the result. Also, maximum array and vector lengths are implementation dependent. Interfacing with the operating system is a bit murky, and implementations surely differ in handling of errors there.

## 289 StandardMLTutorials

- [A Gentle Introduction to ML](#). Andrew Cummings.
  - [Programming in Standard ML '97: An Online Tutorial](#). Stephen Gilmore.
  - [Programming in Standard ML](#). Robert Harper.
  - [Essentials of Standard ML Modules](#). Mads Tofte.
-

## 290 StaticSum

While SML makes it impossible to write functions whose types would depend on the values of their arguments, or so called dependently typed functions, it is possible, and arguably commonplace, to write functions whose types depend on the types of their arguments. Indeed, the types of parametrically polymorphic functions like `map` and `foldl` can be said to depend on the types of their arguments. What is less commonplace, however, is to write functions whose behavior would depend on the types of their arguments. Nevertheless, there are several techniques for writing such functions. [Type-indexed values](#) and [fold](#) are two such techniques. This page presents another such technique dubbed static sums.

### 290.1 Ordinary Sums

Consider the sum type as defined below:

```
structure Sum = struct
 datatype ('a, 'b) t = INL of 'a | INR of 'b
end
```

While a generic sum type such as defined above is very useful, it has a number of limitations. As an example, we could write the function `out` to extract the value from a sum as follows:

```
fun out (s : ('a, 'a) Sum.t) : 'a =
 case s
 of Sum.INL a => a
 | Sum.INR a => a
```

As can be seen from the type of `out`, it is limited in the sense that it requires both variants of the sum to have the same type. So, `out` cannot be used to extract the value of a sum of two different types, such as the type `(int, real) Sum.t`. As another example of a limitation, consider the following attempt at a `succ` function:

```
fun succ (s : (int, real) Sum.t) : ??? =
 case s
 of Sum.INL i => i + 1
 | Sum.INR r => Real.nextAfter (r, Real.posInf)
```

The above definition of `succ` cannot be typed, because there is no type for the codomain within SML.

### 290.2 Static Sums

Interestingly, it is possible to define values `inL`, `inR`, and `match` that satisfy the laws

```
match (inL x) (f, g) = f x
match (inR x) (f, g) = g x
```

and do not suffer from the same limitations. The definitions are actually quite trivial:

```
structure StaticSum = struct
 fun inL x (f, _) = f x
 fun inR x (_, g) = g x
 fun match x = x
end
```

Now, given the `succ` function defined as

```
fun succ s =
 StaticSum.match s
 (fn i => i + 1,
 fn r => Real.nextAfter (r, Real.posInf))
```

we get

```
succ (StaticSum.inL 1) = 2
succ (StaticSum.inR Real.maxFinite) = Real.posInf
```

To better understand how this works, consider the following signature for static sums:

```
structure StaticSum :> sig
 type ('dL, 'cL, 'dR, 'cR, 'c) t
 val inL : 'dL -> ('dL, 'cL, 'dR, 'cR, 'cL) t
 val inR : 'dR -> ('dL, 'cL, 'dR, 'cR, 'cR) t
 val match : ('dL, 'cL, 'dR, 'cR, 'c) t -> ('dL -> 'cL) * ('dR -> 'cR) -> 'c
end = struct
 type ('dL, 'cL, 'dR, 'cR, 'c) t = ('dL -> 'cL) * ('dR -> 'cR) -> 'c
 open StaticSum
end
```

Above, 'd stands for domain and 'c for codomain. The key difference between an ordinary sum type, like `(int, real) Sum.t`, and a static sum type, like `(int, real, real, int, real) StaticSum.t`, is that the ordinary sum type says nothing about the type of the result of deconstructing a sum while the static sum type specifies the type.

With the sealed static sum module, we get the type

```
val succ : (int, int, real, real, 'a) StaticSum.t -> 'a
```

for the previously defined `succ` function. The type specifies that `succ` maps a left `int` to an `int` and a right `real` to a `real`. For example, the type of `StaticSum.inL 1` is `(int, 'cL, 'dR, 'cR, 'cL) StaticSum.t`. Unifying this with the argument type of `succ` gives the type `(int, int, real, real, int) StaticSum.t -> int`.

The `out` function is quite useful on its own. Here is how it can be defined:

```
structure StaticSum = struct
 open StaticSum
 val out : ('a, 'a, 'b, 'b, 'c) t -> 'c =
 fn s => match s (fn x => x, fn x => x)
end
```

Due to the value restriction, lack of first class polymorphism and polymorphic recursion, the usefulness and convenience of static sums is somewhat limited in SML. So, don't throw away the ordinary sum type just yet. Static sums can nevertheless be quite useful.

### 290.2.1 Example: Send and Receive with Argument Type Dependent Result Types

In some situations it would seem useful to define functions whose result type would depend on some of the arguments. Traditionally such functions have been thought to be impossible in SML and the solution has been to define multiple functions. For example, the `Socket structure` of the Basis library defines 16 `send` and 16 `recv` functions. In contrast, the `Net structure` (`net.sig`) of the Basic library designed by Stephen Weeks defines only a single `send` and a single `receive` and the result types of the functions depend on their arguments. The implementation (`net.sml`) uses static sums (with a slightly different signature: `static-sum.sig`).

### 290.2.2 Example: Picking Monad Results

Suppose that we need to write a parser that accepts a pair of integers and returns their sum given a monadic parsing combinator library. A part of the signature of such library could look like this

```
signature PARSING = sig
 include MONAD
 val int : int t
 val lparen : unit t
 val rparen : unit t
```

```

 val comma : unit t
 (* ... *)
end

```

where the MONAD signature could be defined as

```

signature MONAD = sig
 type 'a t
 val return : 'a -> 'a t
 val >>= : 'a t * ('a -> 'b t) -> 'b t
end
infix >>=

```

The straightforward, but tedious, way to write the desired parser is:

```

val p = lparen >>= (fn _ =>
 int >>= (fn x =>
 comma >>= (fn _ =>
 int >>= (fn y =>
 rparen >>= (fn _ =>
 return (x + y))))))

```

In Haskell, the parser could be written using the `do` notation considerably less verbosely as:

```

p = do { lparen ; x <- int ; comma ; y <- int ; rparen ; return $ x + y }

```

SML doesn't provide a `do` notation, so we need another solution.

Suppose we would have a "pick" notation for monads that would allows us to write the parser as

```

val p = `lparen ^ `int ^ `comma ^ `int ^ `rparen @ (fn x & y => x + y)

```

using four auxiliary combinators: ```, `\`, `^`, and `@`.

Roughly speaking

- ``p` means that the result of `p` is dropped,
- `\p` means that the result of `p` is taken,
- `p ^ q` means that results of `p` and `q` are taken as a product, and
- `p @ a` means that the results of `p` are passed to the function `a` and that result is returned.

The difficulty is in implementing the concatenation combinator `^`. The type of the result of the concatenation depends on the types of the arguments.

Using static sums and the [product type](#), the pick notation for monads can be implemented as follows:

```

functor MkMonadPick (include MONAD) = let
 open StaticSum
in
 struct
 fun `a = inL (a >>= (fn _ => return ()))
 val \ = inR
 fun a @ f = out a >>= (return o f)
 fun a ^ b =
 (match b o match a)
 (fn a =>
 (fn b => inL (a >>= (fn _ => b)),
 fn b => inR (a >>= (fn _ => b))),
 fn a =>
 (fn b => inR (a >>= (fn a => b >>= (fn _ => return a))),
 fn b => inR (a >>= (fn a => b >>= (fn b => return (a & b))))))
 end
end

```

The above implementation is inefficient, however. It uses many more bind operations, `>>=`, than necessary. That can be solved with an additional level of abstraction:

```

functor MkMonadPick (include MONAD) = let
 open StaticSum
in
 struct
 fun `a = inL (fn b => a >>= (fn _ => b ()))
 fun `a = inR (fn b => a >>= b)
 fun a @ f = out a (return o f)
 fun a ^ b =
 (match b o match a)
 (fn a => (fn b => inL (fn c => a (fn () => b c)),
 fn b => inR (fn c => a (fn () => b c))),
 fn a => (fn b => inR (fn c => a (fn a => b (fn () => c a))),
 fn b => inR (fn c => a (fn a => b (fn b => c (a & b))))))
 end
end

```

After instantiating and opening either of the above monad pick implementations, the previously given definition of `p` can be compiled and results in a parser whose result is of type `int`. Here is a functor to test the theory:

```

functor Test (Arg : PARSING) = struct
 local
 structure Pick = MkMonadPick (Arg)
 open Pick Arg
 in
 val p : int t =
 `lparen ^ `int ^ `comma ^ `int ^ `rparen @ (fn x & y => x + y)
 end
end

```

### 290.3 Also see

There are a number of related techniques. Here are some of them.

- [Fold](#)
- [TypeIndexedValues](#)

## 291 StephenWeeks

I live in the New York City area and work at [Jane Street Capital](#).

My [home page](#).

You can email me at [sweeks@sweeks.com](mailto:sweeks@sweeks.com).

---

## 292 StyleGuide

These conventions are chosen so that inertia is towards modularity, code reuse and finding bugs early, *not* to save typing.

- [SyntacticConventions](#)
-

## 293 Subversion

**Subversion** is a version control system. The MLton project used Subversion to maintain its [source code](#), but switched to [Git](#) on 20130308.

Here are some online Subversion resources.

- [Version Control with Subversion](#)
-

## 294 SuccessorML

The purpose of **successor ML**, or sML for short, is to provide a vehicle for the continued evolution of ML, using Standard ML as a starting point. The intention is for successor ML to be a living, evolving dialect of ML that is responsive to community needs and advances in language design, implementation, and semantics.

---

## 295 SureshJagannathan

I am an Associate Professor at the [Department of Computer Science](#) at Purdue University. My research focus is in programming language design and implementation, concurrency, and distributed systems. I am interested in various aspects of MLton, mostly related to (in no particular order): (1) control-flow analysis (2) representation strategies (e.g., flattening), (3) IR formats, and (4) extensions for distributed programming.

Please see my [Home page](#) for more details.

---

## 296 Swerve

**Swerve** is an HTTP server written in SML, originally developed with SML/NJ. [RayRacine](#) ported Swerve to MLton in January 2005.

[Download](#) the port.

Excerpt from the included README:

Total testing of this port consisted of a successful compile, startup, and serving one html page with one gif image. Given that the original code was thoroughly designed and implemented in a thoughtful manner and I expect it is quite usable modulo a few minor bugs introduced by my porting effort.

Swerve is described in [Shipman02](#).

---

## 297 SXML

SXML is an [IntermediateLanguage](#), translated from [XML](#) by [Monomorphise](#), optimized by [SXMLSimplify](#), and translated by [ClosureConvert](#) to [SSA](#).

### 297.1 Description

SXML is a simply-typed version of [XML](#).

### 297.2 Implementation

- `sxml.sig`
- `sxml.fun`
- `sxml-tree.sig`

### 297.3 Type Checking

[SXML](#) shares the type checker for [XML](#).

### 297.4 Details and Notes

There are only two differences between [XML](#) and [SXML](#). First, [SXML](#) `val`, `fun`, and `datatype` declarations always have an empty list of type variables. Second, [SXML](#) variable references always have an empty list of type arguments. Constructors uses can only have a nonempty list of type arguments if the constructor is a primitive.

Although we could rely on the type system to enforce these constraints by parameterizing the [XML](#) signature, [StephenWeeks](#) did so in a previous version of the compiler, and the software engineering gains were not worth the effort.

---

## 298 SXMLShrink

SXMLShrink is an optimization pass for the [SXML IntermediateLanguage](#), invoked from [SXMLSimplify](#).

### 298.1 Description

This pass performs optimizations based on a reduction system.

### 298.2 Implementation

- `shrink.sig`
- `shrink.fun`

### 298.3 Details and Notes

[SXML](#) shares the [XMLShrink](#) simplifier.

---

## 299 SXMLSimplify

The optimization passes for the [SXML IntermediateLanguage](#) are collected and controlled by the `SxmlSimplify` functor (`sxml-simplify.sig`, `sxml-simplify.fun`).

The following optimization passes are implemented:

- [Polyvariance](#)
- [SXMLShrink](#)

The following implementation passes are implemented:

- [ImplementExceptions](#)
- [ImplementSuffix](#)

The following optimization passes are not implemented, but might prove useful:

- [Uncurry](#)
- [LambdaLift](#)

The optimization passes can be controlled from the command-line by the options

- `-diag-pass <pass>` — keep diagnostic info for pass
  - `-drop-pass <pass>` — omit optimization pass
  - `-keep-pass <pass>` — keep the results of pass
  - `-sxml-passes <passes>` — sxml optimization passes
-

## 300 SyntacticConventions

Here are a number of syntactic conventions useful for programming in SML.

### 300.1 General

- A line of code never exceeds 80 columns.
- Only split a syntactic entity across multiple lines if it doesn't fit on one line within 80 columns.
- Use alphabetical order wherever possible.
- Avoid redundant parentheses.
- When using `:`, there is no space before the colon, and a single space after it.

### 300.2 Identifiers

- Variables, record labels and type constructors begin with and use small letters, using capital letters to separate words.

```
cost
maxValue
```

- Variables that represent collections of objects (lists, arrays, vectors, ...) are often suffixed with an `s`.

```
xs
employees
```

- Constructors, structure identifiers, and functor identifiers begin with a capital letter.

```
Queue
LinkedList
```

- Signature identifiers are in all capitals, using `_` to separate words.

```
LIST
BINARY_HEAP
```

### 300.3 Types

- Alphabetize record labels. In a record type, there are spaces after colons and commas, but not before colons or commas, or at the delimiters `{` and `}`.

```
{bar: int, foo: int}
```

- Only split a record type across multiple lines if it doesn't fit on one line. If a record type must be split over multiple lines, put one field per line.

```
{bar: int,
 foo: real * real,
 zoo: bool}
```

- In a tuple type, there are spaces before and after each `*`.

```
int * bool * real
```

- Only split a tuple type across multiple lines if it doesn't fit on one line. In a tuple type split over multiple lines, there is one type per line, and the \*-s go at the beginning of the lines.

```
int
* bool
* real
```

It may also be useful to parenthesize to make the grouping more apparent.

```
(int
 * bool
 * real)
```

- In an arrow type split over multiple lines, put the arrow at the beginning of its line.

```
int * real
-> bool
```

It may also be useful to parenthesize to make the grouping more apparent.

```
(int * real
-> bool)
```

- Avoid redundant parentheses.
- Arrow types associate to the right, so write

```
a -> b -> c
```

not

```
a -> (b -> c)
```

- Type constructor application associates to the left, so write

```
int ref list
```

not

```
(int ref) list
```

- Type constructor application binds more tightly than a tuple type, so write

```
int list * bool list
```

not

```
(int list) * (bool list)
```

- Tuple types bind more tightly than arrow types, so write

```
int * bool -> real
```

not

```
(int * bool) -> real
```

### 300.4 Core

- A core expression or declaration split over multiple lines does not contain any blank lines.

- A record field selector has no space between the # and the record label. So, write

```
#foo
```

```
not
```

```
foo
```

- A tuple has a space after each comma, but not before, and not at the delimiters ( and ).

```
(e1, e2, e3)
```

- A tuple split over multiple lines has one element per line, and the commas go at the end of the lines.

```
(e1,
e2,
e3)
```

- A list has a space after each comma, but not before, and not at the delimiters [ and ].

```
[e1, e2, e3]
```

- A list split over multiple lines has one element per line, and the commas at the end of the lines.

```
[e1,
e2,
e3]
```

- A record has spaces before and after =, a space after each comma, but not before, and not at the delimiters { and }. Field names appear in alphabetical order.

```
{bar = 13, foo = true}
```

- A sequence expression has a space after each semicolon, but not before.

```
(e1; e2; e3)
```

- A sequence expression split over multiple lines has one expression per line, and the semicolons at the beginning of lines. Lisp and Scheme programmers may find this hard to read at first.

```
(e1
; e2
; e3)
```

*Rationale:* this makes it easy to visually spot the beginning of each expression, which becomes more valuable as the expressions themselves are split across multiple lines.

- An application expression has a space between the function and the argument. There are no parens unless the argument is a tuple (in which case the parens are really part of the tuple, not the application).

```
f a
f (a1, a2, a3)
```

- Avoid redundant parentheses. Application associates to left, so write

```
f a1 a2 a3
```

not

```
((f a1) a2) a3
```

- Infix operators have a space before and after the operator.

```
x + y
x * y - z
```

- Avoid redundant parentheses. Use [OperatorPrecedence](#). So, write

```
x + y * z
```

not

```
x + (y * z)
```

- An `andalso` expression split over multiple lines has the `andalso` at the beginning of subsequent lines.

```
e1
andalso e2
andalso e3
```

- A case expression is indented as follows

```
case e1 of
 p1 => e1
| p2 => e2
| p3 => e3
```

- A datatype's constructors are alphabetized.

```
datatype t = A | B | C
```

- A datatype declaration has a space before and after each `|`.

```
datatype t = A | B of int | C
```

- A datatype split over multiple lines has one constructor per line, with the `|` at the beginning of lines and the constructors beginning 3 columns to the right of the datatype.

```
datatype t =
 A
| B
| C
```

- A fun declaration may start its body on the subsequent line, indented 3 spaces.

```
fun f x y =
 let
 val z = x + y + z
 in
 z
 end
```

- An `if` expression is indented as follows.

```
if e1
 then e2
 else e3
```

- A sequence of if-then-else-s is indented as follows.

```
if e1
 then e2
else if e3
 then e4
else if e5
 then e6
else e7
```

- A let expression has the let, in, and end on their own lines, starting in the same column. Declarations and the body are indented 3 spaces.

```
let
 val x = 13
 val y = 14
in
 x + y
end
```

- A local declaration has the local, in, and end on their own lines, starting in the same column. Declarations are indented 3 spaces.

```
local
 val x = 13
in
 val y = x
end
```

- An orelse expression split over multiple lines has the orelse at the beginning of subsequent lines.

```
e1
orelse e2
orelse e3
```

- A val declaration has a space before and after the =.

```
val p = e
```

- A val declaration can start the expression on the subsequent line, indented 3 spaces.

```
val p =
 if e1 then e2 else e3
```

### 300.5 Signatures

- A signature declaration is indented as follows.

```
signature FOO =
 sig
 val x: int
 end
```

*Exception:* a signature declaration in a file to itself can omit the indentation to save horizontal space.

```
signature FOO =
sig

val x: int

end
```

In this case, there should be a blank line after the `sig` and before the `end`.

- A `val` specification has a space after the colon, but not before.

```
val x: int
```

*Exception:* in the case of operators (like `+`), there is a space before the colon to avoid lexing the colon as part of the operator.

```
val + : t * t -> t
```

- Alphabetize specifications in signatures.

```
sig
 val x: int
 val y: bool
end
```

## 300.6 Structures

- A structure declaration has a space on both sides of the `=`.

```
structure Foo = Bar
```

- A structure declaration split over multiple lines is indented as follows.

```
structure S =
 struct
 val x = 13
 end
```

*Exception:* a structure declaration in a file to itself can omit the indentation to save horizontal space.

```
structure S =
struct

val x = 13

end
```

In this case, there should be a blank line after the `struct` and before the `end`.

- Declarations in a `struct` are separated by blank lines.

```
struct
 val x =
 let
 y = 13
 in
 y + 1
 end

 val z = 14
end
```

### 300.7 Functors

- A `functor` declaration has spaces after each `:` (or `:>`) but not before, and a space before and after the `=`. It is indented as follows.

```
functor Foo (S: FOO_ARG) : FOO =
 struct
 val x = S.x
 end
```

*Exception:* a functor declaration in a file to itself can omit the indentation to save horizontal space.

```
functor Foo (S: FOO_ARG) : FOO =
struct

val x = S.x

end
```

In this case, there should be a blank line after the `struct` and before the `end`.

## 301 Talk

### 301.1 The MLton Standard ML Compiler

Henry Cejtin, Matthew Fluet, Suresh Jagannathan, Stephen Weeks

---

	<a href="#">Next</a>
--	----------------------

---

## 302 TalkDiveIn

### 302.1 Dive In

- to [Development](#)
- to [Documentation](#)
- to [Download](#)

---

<a href="#">Prev</a>	
----------------------	--

## 303 TalkFolkLore

### 303.1 Folk Lore

- Defunctorization and monomorphisation are feasible
- Global control-flow analysis is feasible
- Early closure conversion is feasible

---

<a href="#">Prev</a>	<a href="#">Next</a>
----------------------	----------------------

## 304 TalkFromSMLTo

### 304.1 From Standard ML to S-T F-O IL

- What issues arise when translating from Standard ML into an intermediate language?

---

<a href="#">Prev</a>	<a href="#">Next</a>
----------------------	----------------------

## 305 TalkHowHigherOrder

### 305.1 Higher-order Functions

- How does one represent SML's higher-order functions?
- MLton's answer: defunctionalize

See [ClosureConvert](#).

---

<a href="#">Prev</a>	<a href="#">Next</a>
----------------------	----------------------

## 306 TalkHowModules

### 306.1 Modules

- How does one represent SML's modules?
- MLton's answer: defunctorize

See [Elaborate](#).

---

<a href="#">Prev</a>	<a href="#">Next</a>
----------------------	----------------------

## 307 TalkHowPolymorphism

### 307.1 Polymorphism

- How does one represent SML's polymorphism?
- MLton's answer: monomorphise

See [Monomorphise](#).

---

<a href="#">Prev</a>	<a href="#">Next</a>
----------------------	----------------------

## 308 TalkMLtonApproach

### 308.1 MLton's Approach

- whole-program optimization using a simply-typed, first-order intermediate language
- ensures programs are not penalized for exploiting abstraction and modularity

---

<a href="#">Prev</a>	<a href="#">Next</a>
----------------------	----------------------

## 309 TalkMLtonFeatures

### 309.1 MLton Features

- Supports full Standard ML language and Basis Library
- Generates standalone executables
- Extensions
  - Foreign function interface (SML to C, C to SML)
  - ML Basis system for programming in the very large
  - Extension libraries

See [Features](#).

---

<a href="#">Prev</a>	<a href="#">Next</a>
----------------------	----------------------

## 310 TalkMLtonHistory

### 310.1 MLton History

April 1997	Stephen Weeks wrote a defunctorizer for SML/NJ
Aug. 1997	Begin independent compiler ( <code>smlc</code> )
Oct. 1997	Monomorphiser
Nov. 1997	Polyvariant higher-order control-flow analysis (10,000 lines)
March 1999	First release of MLton (48,006 lines)
Jan. 2002	MLton at 102,541 lines
Jan. 2003	MLton at 112,204 lines
Jan. 2004	MLton at 122,299 lines
Nov. 2004	MLton at 141,311 lines

See [History](#).

---

[Prev](#)

[Next](#)

## 311 TalkStandardML

### 311.1 Standard ML

- a high-level language makes
  - a programmer's life easier
  - a compiler writer's life harder
- perceived overheads of features discourage their use
  - higher-order functions
  - polymorphic datatypes
  - separate modules

Also see [Standard ML](#).

---

<a href="#">Prev</a>	<a href="#">Next</a>
----------------------	----------------------

## 312 TalkTemplate

### 312.1 Title

- Bullet
- Bullet

---

<a href="#">Prev</a>	<a href="#">Next</a>
----------------------	----------------------

## 313 TalkWholeProgram

### 313.1 Whole Program Compiler

- Each of these techniques requires whole-program analysis
- But, additional benefits:
  - eliminate (some) variability in programming styles
  - specialize representations
  - simplifies and improves runtime system

---

<a href="#">Prev</a>	<a href="#">Next</a>
----------------------	----------------------

## 314 TILT

TILT is a [Standard ML](#) implementation.

## 315 TipsForWritingConciseSML

SML is a rich enough language that there are often several ways to express things. This page contains miscellaneous tips (ideas not rules) for writing concise SML. The metric that we are interested in here is the number of tokens or words (rather than the number of lines, for example).

### 315.1 Datatypes in Signatures

A seemingly frequent source of repetition in SML is that of datatype definitions in signatures and structures. Actually, it isn't repetition at all. A datatype specification in a signature, such as,

```
signature EXP = sig
 datatype exp = Fn of id * exp | App of exp * exp | Var of id
end
```

is just a specification of a datatype that may be matched by multiple (albeit identical) datatype declarations. For example, in

```
structure AnExp : EXP = struct
 datatype exp = Fn of id * exp | App of exp * exp | Var of id
end

structure AnotherExp : EXP = struct
 datatype exp = Fn of id * exp | App of exp * exp | Var of id
end
```

the types `AnExp.exp` and `AnotherExp.exp` are two distinct types. If such [generativity](#) isn't desired or needed, you can avoid the repetition:

```
structure Exp = struct
 datatype exp = Fn of id * exp | App of exp * exp | Var of id
end

signature EXP = sig
 datatype exp = datatype Exp.exp
end

structure Exp : EXP = struct
 open Exp
end
```

Keep in mind that this isn't semantically equivalent to the original.

### 315.2 Clausal Function Definitions

The syntax of clausal function definitions is rather repetitive. For example,

```
fun isSome NONE = false
 | isSome (SOME _) = true
```

is more verbose than

```
val isSome =
 fn NONE => false
 | SOME _ => true
```

For recursive functions the break-even point is one clause higher. For example,

```
fun fib 0 = 0
 | fib 1 = 1
 | fib n = fib (n-1) + fib (n-2)
```

isn't less verbose than

```
val rec fib =
 fn 0 => 0
 | 1 => 1
 | n => fib (n-1) + fib (n-2)
```

It is quite often the case that a curried function primarily examines just one of its arguments. Such functions can be written particularly concisely by making the examined argument last. For example, instead of

```
fun eval (Fn (v, b)) env => ...
 | eval (App (f, a) env) => ...
 | eval (Var v) env => ...
```

consider writing

```
fun eval env =
 fn Fn (v, b) => ...
 | App (f, a) => ...
 | Var v => ...
```

### 315.3 Parentheses

It is a good idea to avoid using lots of irritating superfluous parentheses. An important rule to know is that prefix function application in SML has higher precedence than any infix operator. For example, the outer parentheses in

```
(square (5 + 1)) + (square (5 * 2))
```

are superfluous.

People trained in other languages often use superfluous parentheses in a number of places. In particular, the parentheses in the following examples are practically always superfluous and are best avoided:

```
if (condition) then ... else ...
while (condition) do ...
```

The same basically applies to case expressions:

```
case (expression) of ...
```

It is not uncommon to match a tuple of two or more values:

```
case (a, b) of
 (A1, B1) => ...
 | (A2, B2) => ...
```

Such case expressions can be written more concisely with an [infix product constructor](#):

```
case a & b of
 A1 & B1 => ...
 | A2 & B2 => ...
```

### 315.4 Conditionals

Repeated sequences of conditionals such as

```
if x < y then ...
else if x = y then ...
else ...
```

can often be written more concisely as case expressions such as

```
case Int.compare (x, y) of
 LESS => ...
| EQUAL => ...
| GREATER => ...
```

For a custom comparison, you would then define an appropriate datatype and a reification function. An alternative to using datatypes is to use dispatch functions

```
comparing (x, y)
{lt = fn () => ...,
 eq = fn () => ...,
 gt = fn () => ...}
```

where

```
fun comparing (x, y) {lt, eq, gt} =
 (case Int.compare (x, y) of
 LESS => lt
 | EQUAL => eq
 | GREATER => gt) ()
```

An advantage is that no datatype definition is needed. A disadvantage is that you can't combine multiple dispatch results easily.

### 315.5 Command-Query Fusion

Many are familiar with the [Command-Query Separation Principle](#). Adhering to the principle, a signature for an imperative stack might contain specifications

```
val isEmpty : 'a t -> bool
val pop : 'a t -> 'a
```

and use of a stack would look like

```
if isEmpty stack
then ... pop stack ...
else ...
```

or, when the element needs to be named,

```
if isEmpty stack
then let val elem = pop stack in ... end
else ...
```

For efficiency, correctness, and conciseness, it is often better to combine the query and command and return the result as an option:

```
val pop : 'a t -> 'a option
```

A use of a stack would then look like this:

```
case pop stack of
 NONE => ...
| SOME elem => ...
```

## 316 ToMachine

[ToMachine](#) is a translation pass from the [RSSA IntermediateLanguage](#) to the [Machine IntermediateLanguage](#).

### 316.1 Description

This pass converts from a [RSSA](#) program into a [Machine](#) program.

It uses [AllocateRegisters](#), [Chunkify](#), and [ParallelMove](#).

### 316.2 Implementation

- [backend.sig](#)
- [backend.fun](#)

### 316.3 Details and Notes

Because the MLton runtime system is shared by all codegens, it is most convenient to decide on stack layout *before* any codegen takes over. In particular, we compute all the stack frame info for each [RSSA](#) function, including stack size, [garbage collector](#) masks for each frame, etc. To do so, the [Machine IntermediateLanguage](#) imagines an abstract machine with an infinite number of (pseudo-)registers of every size. A liveness analysis determines, for each variable, whether or not it is live across a point where the runtime system might take over (for example, any garbage collection point) or a non-tail call to another [RSSA](#) function. Those that are live go on the stack, while those that aren't live go into psuedo-registers. From this information, we know all we need to about each stack frame. On the downside, nothing further on is allowed to change this stack info; it is set in stone.

---

## 317 TomMurphy

Tom Murphy VII is a long time MLton user and occasional contributor. He works on programming languages for his PhD work at Carnegie Mellon in Pittsburgh, USA. [AdamGoode](#) lives on the same floor of Wean Hall.

[Home page](#)

---

## 318 ToRSSA

ToRSSA is a translation pass from the [SSA2 IntermediateLanguage](#) to the [RSSA IntermediateLanguage](#).

### 318.1 Description

This pass converts a [SSA2](#) program into a [RSSA](#) program.

It uses [PackedRepresentation](#).

### 318.2 Implementation

- [ssa-to-rssa.sig](#)
- [ssa-to-rssa.fun](#)

### 318.3 Details and Notes

---

## 319 ToSSA2

ToSSA2 is a translation pass from the [SSA IntermediateLanguage](#) to the [SSA2 IntermediateLanguage](#).

### 319.1 Description

This pass is a simple conversion from a [SSA](#) program into a [SSA2](#) program.

The only interesting portions of the translation are:

- an [SSA](#) `ref` type becomes an object with a single mutable field
- `array`, `vector`, and `ref` are eliminated in favor of `select` and `updates`
- `Case` transfers separate discrimination and constructor argument selects

### 319.2 Implementation

- `ssa-to-ssa2.sig`
- `ssa-to-ssa2.fun`

### 319.3 Details and Notes

---

## 320 TypeChecking

MLton's type checker follows the [Definition](#) closely, so you may find differences between MLton and other SML compilers that do not follow the Definition so closely. In particular, SML/NJ has many deviations from the Definition — please see [SMLNJDeviations](#) for those that we are aware of.

In some respects MLton's type checker is more powerful than other SML compilers, so there are programs that MLton accepts that are rejected by some other SML compilers. These kinds of programs fall into a few simple categories.

- MLton resolves flexible record patterns using a larger context than many other SML compilers. For example, MLton accepts the following.

```
fun f {x, ...} = x
val _ = f {x = 13, y = "foo"}
```

- MLton uses as large a context as possible to resolve the type of variables constrained by the value restriction to be monotypes. For example, MLton accepts the following.

```
structure S:
 sig
 val f: int -> int
 end =
 struct
 val f = (fn x => x) (fn y => y)
 end
```

### 320.1 Type error messages

To aid in the understanding of type errors, MLton's type checker displays type errors differently than other SML compilers. In particular, when two types are different, it is important for the programmer to easily understand why they are different. So, MLton displays only the differences between two types that don't match, using underscores for the parts that match. For example, if a function expects `real * int` but gets `real * real`, the type error message would look like

```
expects: _ * [int]
but got: _ * [real]
```

As another aid to spotting differences, MLton places brackets `[]` around the parts of the types that don't match. A common situation is when a function receives a different number of arguments than it expects, in which case you might see an error like

```
expects: [int * real]
but got: [int * real * string]
```

The brackets make it easy to see that the problem is that the tuples have different numbers of components — not that the components don't match. Contrast that with a case where a function receives the right number of arguments, but in the wrong order, in which case you might see an error like

```
expects: [int] * [real]
but got: [real] * [int]
```

Here the brackets make it easy to see that the components do not match.

We appreciate feedback on any type error messages that you find confusing, or suggestions you may have for improvements to error messages.

### 320.2 The shortest/most-recent rule for type names

In a type error message, MLton often has a number of choices in deciding what name to use for a type. For example, in the following type-incorrect program

```
type t = int
fun f (x: t) = x
val _ = f "foo"
```

MLton reports the error message

```
Error: z.sml 3.9.
 Function applied to incorrect argument.
 expects: [t]
 but got: [string]
 in: f "foo"
```

MLton could have reported `expects: [int]` instead of `expects: [t]`. However, MLton uses the shortest/most-recent rule in order to decide what type name to display. This rule means that, at the point of the error, MLton first looks for the shortest name for a type in terms of number of structure identifiers (e.g. `foobar` is shorter than `A.t`). Next, if there are multiple names of the same length, then MLton uses the most recently defined name. It is this tiebreaker that causes MLton to prefer `t` to `int` in the above example.

In signature matching, most recently defined is taken to include all of the definitions introduced by the structure. For example

```
structure S:
 sig
 val x: int
 end =
 struct
 type t = int
 val x = "foo"
 end
```

MLton reports the error message

```
Error: z.sml 2.4.
 Variable type in structure disagrees with signature.
 variable: x
 structure: [string]
 signature: [t]
```

in which the `[t]` refers to the type defined in the structure, since that is more recent than the definition of `int`.

In signatures with type equations, this can be somewhat confusing. For example.

```
structure S:
 sig
 type t
 type u = t
 end =
 struct
 type t = int
 type u = char
 end
```

MLton reports the error message

```
Error: z.sml 2.4.
 Type definition in structure disagrees with signature.
 type: u
 structure: [u]
 signature: [t]
```

This error reflects the fact that the signature requires type `u` to equal `t`, but that in the structure, `u` is defined to be `char`, whose most-recent name is `u`, while the signature requires `u` to be `int`, whose most-recent name is `t`.

---

## 321 TypeConstructor

In [Standard ML](#), a type constructor is a function from types to types. Type constructors can be *nullary*, meaning that they take no arguments, as in `char`, `int`, and `real`. Type constructors can be *unary*, meaning that they take one argument, as in `array`, `list`, and `vector`. A program can define a new type constructor in two ways: a `type` definition or a `datatype` declaration. User-defined type constructors can take any number of arguments.

```
datatype t = T of int * real (* 0 arguments *)
type 'a t = 'a * int (* 1 argument *)
datatype ('a, 'b) t = A | B of 'a * 'b (* 2 arguments *)
type ('a, 'b, 'c) t = 'a * ('b -> 'c) (* 3 arguments *)
```

Here are the syntax rules for type constructor application.

- Type constructor application is written in postfix. So, one writes `int list`, not `list int`.
- Unary type constructors drop the parens, so one writes `int list`, not `(int) list`.
- Nullary type constructors drop the argument entirely, so one writes `int`, not `() int`.
- N-ary type constructors use tuple notation; for example, `(int, real) t`.
- Type constructor application associates to the left. So, `int ref list` is the same as `(int ref) list`.

## 322 TypeIndexedValues

Standard ML does not support ad hoc polymorphism. This presents a challenge to programmers. The problem is that at first glance there seems to be no practical way to implement something like a function for converting a value of any type to a string or a function for computing a hash value for a value of any type. Fortunately there are ways to implement type-indexed values in SML as discussed in Yang98. Various articles such as Danvy98, Ramsey03, Elsmann04, Kennedy04, and Benton05 also contain examples of type-indexed values.

**NOTE:** The technique used in the following example uses an early (and somewhat broken) variation of the basic technique used in an experimental generic programming library (see README) that can be found from the MLton repository. The generic programming library also includes a more advanced generic pretty printing function (see pretty.sig).

### 322.1 Example: Converting any SML value to (roughly) SML syntax

Consider the problem of converting any SML value to a textual presentation that matches the syntax of SML as closely as possible. One solution is a type-indexed function that maps a given type to a function that maps any value (of the type) to its textual presentation. A type-indexed function like this can be useful for a variety of purposes. For example, one could use it to show debugging information. We'll call this function "show".

We'll do a fairly complete implementation of show. We do not distinguish infix and nonfix constructors, but that is not an intrinsic property of SML datatypes. We also don't reconstruct a type name for the value, although it would be particularly useful for functional values. To reconstruct type names, some changes would be needed and the reader is encouraged to consider how to do that. A more realistic implementation would use some pretty printing combinators to compute a layout for the result. This should be a relatively easy change (given a suitable pretty printing library). Cyclic values (through references and arrays) do not have a standard textual presentation and it is impossible to convert arbitrary functional values (within SML) to a meaningful textual presentation. Finally, it would also make sense to show sharing of references and arrays. We'll leave these improvements to an actual library implementation.

The following code uses the [fixpoint framework](#) and other utilities from an Extended Basis library (see [README](#)).

#### 322.1.1 Signature

Let's consider the design of the SHOW signature:

```
infixr -->

signature SHOW = sig
 type 'a t (* complete type-index *)
 type 'a s (* incomplete sum *)
 type ('a, 'k) p (* incomplete product *)
 type u (* tuple or unlabelled product *)
 type l (* record or labelled product *)

 val show : 'a t -> 'a -> string

 (* user-defined types *)
 val inj : ('a -> 'b) -> 'b t -> 'a t

 (* tuples and records *)
 val * : ('a, 'k) p * ('b, 'k) p -> (('a, 'b) product, 'k) p

 val U : 'a t -> ('a, u) p
 val L : string -> 'a t -> ('a, l) p

 val tuple : ('a, u) p -> 'a t
 val record : ('a, l) p -> 'a t

 (* datatypes *)
 val + : 'a s * 'b s -> (('a, 'b) sum) s
```

```

val C0 : string -> unit s
val C1 : string -> 'a t -> 'a s

val data : 'a s -> 'a t

val Y : 'a t Tie.t

(* exceptions *)
val exn : exn t
val regExn : (exn -> ('a * 'a s) option) -> unit

(* some built-in type constructors *)
val refc : 'a t -> 'a ref t
val array : 'a t -> 'a array t
val list : 'a t -> 'a list t
val vector : 'a t -> 'a vector t
val --> : 'a t * 'b t -> ('a -> 'b) t

(* some built-in base types *)
val string : string t
val unit : unit t
val bool : bool t
val char : char t
val int : int t
val word : word t
val real : real t
end

```

While some details are shaped by the specific requirements of `show`, there are a number of (design) patterns that translate to other type-indexed values. The former kind of details are mostly shaped by the syntax of SML values that `show` is designed to produce. To this end, abstract types and phantom types are used to distinguish incomplete record, tuple, and datatype type-indices from each other and from complete type-indices. Also, names of record labels and datatype constructors need to be provided by the user.

### Arbitrary user-defined datatypes

Perhaps the most important pattern is how the design supports arbitrary user-defined datatypes. A number of combinators together conspire to provide the functionality. First of all, to support new user-defined types, a combinator taking a conversion function to a previously supported type is provided:

```
val inj : ('a -> 'b) -> 'b t -> 'a t
```

An injection function is sufficient in this case, but in the general case, an embedding with injection and projection functions may be needed.

To support products (tuples and records) a product combinator is provided:

```
val * : ('a, 'k) p * ('b, 'k) p -> (('a, 'b) product, 'k) p
```

The second (phantom) type variable `'k` is there to distinguish between labelled and unlabelled products and the type `p` distinguishes incomplete products from complete type-indices of type `t`. Most type-indexed values do not need to make such distinctions.

To support sums (datatypes) a sum combinator is provided:

```
val + : 'a s * 'b s -> (('a, 'b) sum) s
```

Again, the purpose of the type `s` is to distinguish incomplete sums from complete type-indices of type `t`, which usually isn't necessary.

Finally, to support recursive datatypes, including sets of mutually recursive datatypes, a [fixpoint tier](#) is provided:

```
val Y : 'a t Tie.t
```

Together these combinators (with the more domain specific combinators `U`, `L`, `tuple`, `record`, `C0`, `C1`, and `data`) enable one to encode a type-index for any user-defined datatype.

### Exceptions

The `exn` type in SML is a [universal type](#) into which all types can be embedded. SML also allows a program to generate new exception variants at run-time. Thus a mechanism is required to register handlers for particular variants:

```
val exn : exn t
val regExn : (exn -> ('a * 'a s) option) -> unit
```

The universal `exn` type-index then makes use of the registered handlers. The above particular form of handler, which converts an exception value to a value of some type and a type-index for that type (essentially an existential type) is designed to make it convenient to write handlers. To write a handler, one can conveniently reuse existing type-indices:

```
exception Int of int

local
 open Show
in
 val () = regExn (fn Int v => SOME (v, C1"Int" int)
 | _ => NONE)
end
```

Note that a single handler may actually handle an arbitrary number of different exceptions.

### Other types

Some built-in and standard types typically require special treatment due to their special nature. The most important of these are arrays and references, because cyclic data (ignoring closures) and observable sharing can only be constructed through them.

When arrow types are really supported, unlike in this case, they usually need special treatment due to the contravariance of arguments.

Lists and vectors require special treatment in the case of `show`, because of their special syntax. This isn't usually the case.

The set of base types to support also needs to be considered unless one exports an interface for constructing type-indices for entirely new base types.

## 322.2 Usage

Before going to the implementation, let's look at some examples. For the following examples, we'll assume a structure binding `Show :> SHOW`. If you want to try the examples immediately, just skip forward to the implementation.

To use `show`, one first needs a type-index, which is then given to `show`. To show a list of integers, one would use the type-index `list int`, which has the type `int list Show.t`:

```
val "[3, 1, 4]" =
 let open Show in show (list int) end
 [3, 1, 4]
```

Likewise, to show a list of lists of characters, one would use the type-index `list (list char)`, which has the type `char list list Show.t`:

```
val "[[#\"a\", #\"b\", #\"c\"], []]" =
 let open Show in show (list (list char)) end
 [[#"a", #"b", #"c"], []]
```

Handling standard types is not particularly interesting. It is more interesting to see how user-defined types can be handled. Although the `option` datatype is a standard type, it requires no special support, so we can treat it as a user-defined type. Options can be encoded easily using a sum:

```
fun option t = let
 open Show
in
 inj (fn NONE => INL ()
 | SOME v => INR v)
 (data (C0"NONE" + C1"SOME" t))
end

val "SOME 5" =
 let open Show in show (option int) end
 (SOME 5)
```

Readers new to type-indexed values might want to type annotate each subexpression of the above example as an exercise. (Use a compiler to check your annotations.)

Using a product, user specified records can be also be encoded easily:

```
val abc = let
 open Show
in
 inj (fn {a, b, c} => a & b & c)
 (record (L"a" (option int) *
 L"b" real *
 L"c" bool))
end

val "{a = SOME 1, b = 3.0, c = false}" =
 let open Show in show abc end
 {a = SOME 1, b = 3.0, c = false}
```

As you can see, both of the above use `inj` to inject user-defined types to the general purpose sum and product types.

Of particular interest is whether recursive datatypes and cyclic data can be handled. For example, how does one write a type-index for a recursive datatype such as a cyclic graph?

```
datatype 'a graph = VTX of 'a * 'a graph list ref
fun arcs (VTX (_, r)) = r
```

Using the `Show` combinators, we could first write a new type-index combinator for `graph`:

```
fun graph a = let
 open Tie Show
in
 fix Y (fn graph_a =>
 inj (fn VTX (x, y) => x & y)
 (data (C1"VTX"
 (tuple (U a *
 U (refc (list graph_a)))))))
end
```

To show a graph with integer labels

```
val a_graph = let
 val a = VTX (1, ref [])
 val b = VTX (2, ref [])
 val c = VTX (3, ref [])
 val d = VTX (4, ref [])
 val e = VTX (5, ref [])
 val f = VTX (6, ref [])
```



```

val "[|ExnArray %0|] as %0" =
 let open Show in show array_exn end
 a_cycle

```

Cyclic data (excluding closures) in Standard ML can only be constructed imperatively through arrays and references (combined with exceptions or recursive datatypes). Before recursing to a reference or an array, one needs to check whether that reference or array has already been seen before. When `ref` or `array` is called with a type-index, a new cyclicity checker is instantiated.

### 322.3 Implementation

```

structure SmlSyntax = struct
 local
 structure CV = CharVector and C = Char
 in
 val isSym = Char.contains "!%&$#+-/:<=>?@\~\^|*"

 fun isSymId s = 0 < size s andalso CV.all isSym s

 fun isAlphaNumId s =
 0 < size s
 andalso C.isAlpha (CV.sub (s, 0))
 andalso CV.all (fn c => C.isAlphaNum c
 orelse #"'" = c
 orelse #"_" = c) s

 fun isNumLabel s =
 0 < size s
 andalso #"0" <> CV.sub (s, 0)
 andalso CV.all C.isDigit s

 fun isId s = isAlphaNumId s orelse isSymId s

 fun isLongId s = List.all isId (String.fields (#"." <\ op =) s)

 fun isLabel s = isId s orelse isNumLabel s
 end
end

structure Show :> SHOW = struct
 datatype 'a t = IN of exn list * 'a -> bool * string
 type 'a s = 'a t
 type ('a, 'k) p = 'a t
 type u = unit
 type l = unit

 fun show (IN t) x = #2 (t ([], x))

 (* user-defined types *)
 fun inj inj (IN b) = IN (b o Pair.map (id, inj))

 local
 fun surround pre suf (_, s) = (false, concat [pre, s, suf])
 fun parenthesize x = if #1 x then surround "(" ")" x else x
 fun construct tag =
 (fn (_, s) => (true, concat [tag, " ", s])) o parenthesize
 in
 fun check p m s = if p s then () else raise Fail (m^s)

 (* tuples and records *)
 fun (IN l) * (IN r) =

```

```

 IN (fn (rs, a & b) =>
 (false, concat [#2 (l (rs, a)),
 ", ",
 #2 (r (rs, b))]))

val U = id
fun L l = (check SmlSyntax.isLabel "Invalid label: " l
 ; fn IN t => IN (surround (l^" = ") "" o t))

fun tuple (IN t) = IN (surround "(" o t)
fun record (IN t) = IN (surround "{" o t)

(* datatypes *)
fun (IN l) + (IN r) = IN (fn (rs, INL a) => l (rs, a)
 | (rs, INR b) => r (rs, b))

fun C0 c = (check SmlSyntax.isId "Invalid constructor: " c
 ; IN (const (false, c)))
fun C1 c (IN t) = (check SmlSyntax.isId "Invalid constructor: " c
 ; IN (construct c o t))

val data = id

fun Y ? = Tie.iso Tie.function (fn IN x => x, IN) ?

(* exceptions *)
local
 val handlers = ref ([] : (exn -> unit t option) list)
in
 val exn = IN (fn (rs, e) => let
 fun lp [] =
 C0(concat ["<exn:",
 General.exnName e,
 ">"])
 | lp (f::fs) =
 case f e
 of NONE => lp fs
 | SOME t => t
 val IN f = lp (!handlers)
 in
 f (rs, ())
 end)
 fun regExn f =
 handlers := (Option.map
 (fn (x, IN f) =>
 IN (fn (rs, ()) =>
 f (rs, x))) o f)
 :: !handlers
end

(* some built-in type constructors *)
local
 fun cyclic (IN t) = let
 exception E of ''a * bool ref
 in
 IN (fn (rs, v : ''a) => let
 val idx = Int.toString o length
 fun lp (E (v', c)::rs) =
 if v' <> v then lp rs
 else (c := false ; (false, "%"^idx rs))
 | lp (_::rs) = lp rs
 | lp [] = let

```

```

 val c = ref true
 val r = t (E (v, c)::rs, v)
 in
 if !c then r
 else surround " " (" as %" ^idx rs) r
 end
in
 lp rs
end)
end

fun aggregate pre suf toList (IN t) =
 IN (surround pre suf o
 (fn (rs, a) =>
 (false,
 String.concatWith
 ", "
 (map (#2 o curry t rs)
 (toList a))))))
in
 fun refc ? = (cyclic o inj ! o C1"ref") ?
 fun array ? = (cyclic o aggregate "[]" "[]" (Array.foldr op:: [])) ?
 fun list ? = aggregate "[" "]" id ?
 fun vector ? = aggregate "#[" "]" (Vector.foldr op:: [] ?
end

fun (IN _) --> (IN _) = IN (const (false, "<fn>"))

(* some built-in base types *)
local
 fun mk toS = (fn x => (false, x)) o toS o (fn (_, x) => x)
in
 val string =
 IN (surround "\"" "\"" o mk (String.translate Char.toString))
 val unit = IN (mk (fn () => "()"))
 val bool = IN (mk Bool.toString)
 val char = IN (surround "#\"" "\"" o mk Char.toString)
 val int = IN (mk Int.toString)
 val word = IN (surround "0wx" "" o mk Word.toString)
 val real = IN (mk Real.toString)
end
end
end

(* Handlers for standard top-level exceptions *)
val () = let
 open Show
 fun E0 name = SOME ((), C0 name)
in
 regExn (fn Bind => E0"Bind"
 | Chr => E0"Chr"
 | Div => E0"Div"
 | Domain => E0"Domain"
 | Empty => E0"Empty"
 | Match => E0"Match"
 | Option => E0"Option"
 | Overflow => E0"Overflow"
 | Size => E0"Size"
 | Span => E0"Span"
 | Subscript => E0"Subscript"
 | _ => NONE)
; regExn (fn Fail s => SOME (s, C1"Fail" string)

```

```
| _ => NONE)
end
```

### 322.4 Also see

There are a number of related techniques. Here are some of them.

- [Fold](#)
- [StaticSum](#)

## 323 TypeVariableScope

In [Standard ML](#), every type variable is *scoped* (or bound) at a particular point in the program. A type variable can be either implicitly scoped or explicitly scoped. For example, `'a` is implicitly scoped in

```
val id: 'a -> 'a = fn x => x
```

and is implicitly scoped in

```
val id = fn x: 'a => x
```

On the other hand, `'a` is explicitly scoped in

```
val 'a id: 'a -> 'a = fn x => x
```

and is explicitly scoped in

```
val 'a id = fn x: 'a => x
```

A type variable can be scoped at a `val` or `fun` declaration. An SML type checker performs scope inference on each top-level declaration to determine the scope of each implicitly scoped type variable. After scope inference, every type variable is scoped at exactly one enclosing `val` or `fun` declaration. Scope inference shows that the first and second example above are equivalent to the third and fourth example, respectively.

Section 4.6 of the [Definition](#) specifies precisely the scope of an implicitly scoped type variable. A free occurrence of a type variable `'a` in a declaration `d` is said to be *unguarded* in `d` if `'a` is not part of a smaller declaration. A type variable `'a` is implicitly scoped at `d` if `'a` is unguarded in `d` and `'a` does not occur unguarded in any declaration containing `d`.

### 323.1 Scope inference examples

- In this example,

```
val id: 'a -> 'a = fn x => x
```

`'a` is unguarded in `val id` and does not occur unguarded in any containing declaration. Hence, `'a` is scoped at `val id` and the declaration is equivalent to the following.

```
val 'a id: 'a -> 'a = fn x => x
```

- In this example,

```
val f = fn x => let exception E of 'a in E x end
```

`'a` is unguarded in `val f` and does not occur unguarded in any containing declaration. Hence, `'a` is scoped at `val f` and the declaration is equivalent to the following.

```
val 'a f = fn x => let exception E of 'a in E x end
```

- In this example (taken from the [Definition](#)),

```
val x: int -> int = let val id: 'a -> 'a = fn z => z in id id end
```

`'a` occurs unguarded in `val id`, but not in `val x`. Hence, `'a` is implicitly scoped at `val id`, and the declaration is equivalent to the following.

```
val x: int -> int = let val 'a id: 'a -> 'a = fn z => z in id id end
```

- In this example,

```
val f = (fn x: 'a => x) (fn y => y)
```

'a occurs unguarded in `val f` and does not occur unguarded in any containing declaration. Hence, 'a is implicitly scoped at `val f`, and the declaration is equivalent to the following.

```
val 'a f = (fn x: 'a => x) (fn y => y)
```

This does not type check due to the [ValueRestriction](#).

- In this example,

```
fun f x =
 let
 fun g (y: 'a) = if true then x else y
 in
 g x
 end
```

'a occurs unguarded in `fun g`, not in `fun f`. Hence, 'a is implicitly scoped at `fun g`, and the declaration is equivalent to

```
fun f x =
 let
 fun 'a g (y: 'a) = if true then x else y
 in
 g x
 end
```

This fails to type check because `x` and `y` must have the same type, and hence 'a can not be generalized at `fun g`. MLton reports the following error.

```
Error: scope.sml 3.7.
 Unable to generalize 'a.
 in: fun 'a g ((y): 'a) = (if true then x else y)
```

This problem could be fixed either by adding an explicit type constraint, as in `fun f (x: 'a)`, or by explicitly scoping 'a, as in `fun 'a f x`.

## 323.2 Restrictions on type variable scope

It is not allowed to scope a type variable within a declaration in which it is already in scope (see the last restriction listed on page 9 of the [Definition](#)). For example, the following program is invalid.

```
fun 'a f (x: 'a) =
 let
 fun 'a g (y: 'a) = y
 in
 ()
 end
```

MLton reports the following error.

```
Error: z.sml 3.11.
 Type variable 'a scoped at an outer declaration.
```

This is an error even if the scoping is implicit. That is, the following program is invalid as well.

```
fun f (x: 'a) =
 let
 fun 'a g (y: 'a) = y
 in
 ()
 end
```

## 324 Unicode

The current release of MLton does not support Unicode. We are working on adding support.

- `WideChar` structure.
- UTF-8 encoded source files.

There is no real support for Unicode in the [Definition](#); there are only a few throw-away sentences along the lines of "ASCII must be a subset of the character set in programs".

Neither is there real support for Unicode in the [Basis Library](#). The general consensus (which includes the opinions of the editors of the Basis Library) is that the `WideChar` structure is insufficient for the purposes of Unicode. There is no `LargeChar` structure, which in itself is a deficiency, since a programmer can not program against the largest supported character size.

MLton has some preliminary support for 16 and 32 bit characters and strings. It is even possible to include arbitrary Unicode characters in 32-bit strings using a `\Uxxxxxxxx` escape sequence. (This longer escape sequence is a minor extension over the [Definition](#) which only allows `\uxxxx`.) This is by no means completely satisfactory in terms of support for Unicode, but it is what is currently available.

There are periodic flurries of questions and discussion about Unicode in MLton/SML. In December 2004, there was a discussion that led to some seemingly sound design decisions. The discussion started at:

<http://www.mlton.org/pipermail/mlton/2004-December/026396.html>

There is a good summary of points at:

<http://www.mlton.org/pipermail/mlton/2004-December/026440.html>

In November 2005, there was a followup discussion and the beginning of some coding.

<http://www.mlton.org/pipermail/mlton/2005-November/028300.html>

We are optimistic that support will appear in the next MLton release.

### 324.1 Also see

The [fxp](#) XML parser has some support for dealing with Unicode documents.

---

## 325 UniversalType

A universal type is a type into which all other types can be embedded. Here's a [Standard ML](#) signature for a universal type.

```
signature UNIVERSAL_TYPE =
 sig
 type t

 val embed: unit -> ('a -> t) * (t -> 'a option)
 end
```

The idea is that type `t` is the universal type and that each call to `embed` returns a new pair of functions (`inject`, `project`), where `inject` embeds a value into the universal type and `project` extracts the value from the universal type. A pair (`inject`, `project`) returned by `embed` works together in that `project u` will return `SOME v` if and only if `u` was created by `inject v`. If `u` was created by a different function `inject'`, then `project` returns `NONE`.

Here's an example embedding integers and reals into a universal type.

```
functor Test (U: UNIVERSAL_TYPE): sig end =
 struct
 val (intIn: int -> U.t, intOut) = U.embed ()
 val r: U.t ref = ref (intIn 13)
 val s1 =
 case intOut (!r) of
 NONE => "NONE"
 | SOME i => Int.toString i
 val (realIn: real -> U.t, realOut) = U.embed ()
 val () = r := realIn 13.0
 val s2 =
 case intOut (!r) of
 NONE => "NONE"
 | SOME i => Int.toString i
 val s3 =
 case realOut (!r) of
 NONE => "NONE"
 | SOME x => Real.toString x
 val () = print (concat [s1, " ", s2, " ", s3, "\n"])
 end
```

Applying `Test` to an appropriate implementation will print

```
13 NONE 13.0
```

Note that two different calls to `embed` on the same type return different embeddings.

Standard ML does not have explicit support for universal types; however, there are at least two ways to implement them.

### 325.1 Implementation Using Exceptions

While the intended use of SML exceptions is for exception handling, an accidental feature of their design is that the `exn` type is a universal type. The implementation relies on being able to declare exceptions locally to a function and on the fact that exceptions are [generative](#).

```
structure U:> UNIVERSAL_TYPE =
 struct
 type t = exn

 fun 'a embed () =
 let
 exception E of 'a
```

```

 fun project (e: t): 'a option =
 case e of
 E a => SOME a
 | _ => NONE
 in
 (E, project)
 end
end

```

## 325.2 Implementation Using Functions and References

```

structure U:> UNIVERSAL_TYPE =
 struct
 datatype t = T of {clear: unit -> unit,
 store: unit -> unit}

 fun 'a embed () =
 let
 val r: 'a option ref = ref NONE
 fun inject (a: 'a): t =
 T {clear = fn () => r := NONE,
 store = fn () => r := SOME a}
 fun project (T {clear, store}): 'a option =
 let
 val () = store ()
 val res = !r
 val () = clear ()
 in
 res
 end
 in
 (inject, project)
 end
 end
 end

```

Note that due to the use of a shared ref cell, the above implementation is not thread safe.

One could try to simplify the above implementation by eliminating the `clear` function, making type `t = unit -> unit`.

```

structure U:> UNIVERSAL_TYPE =
 struct
 type t = unit -> unit

 fun 'a embed () =
 let
 val r: 'a option ref = ref NONE
 fun inject (a: 'a): t = fn () => r := SOME a
 fun project (f: t): 'a option = (r := NONE; f (); !r)
 in
 (inject, project)
 end
 end
 end

```

While correct, this approach keeps the contents of the ref cell alive longer than necessary, which could cause a space leak. The problem is in `project`, where the call to `f` stores some value in some ref cell `r'`. Perhaps `r'` is the same ref cell as `r`, but perhaps not. If we do not clear `r'` before returning from `project`, then `r'` will keep the value alive, even though it is useless.

## 325.3 Also see

- [PropertyList](#): Lisp-style property lists implemented with a universal type

## 326 UnresolvedBugs

Here are the places where MLton deviates from [The Definition of Standard ML \(Revised\)](#). In general, MLton complies with the [Definition](#) quite closely, typically much more closely than other SML compilers (see, e.g., our list of [SML/NJ's deviations](#)). In fact, the four deviations listed here are the only known deviations, and we have no plans to fix them. If you find a deviation not listed here, please report a [Bug](#).

We don't plan to fix these bugs because the first (parsing nested cases) has historically never been accepted by any SML compiler, the other three clearly indicate problems in the [Definition](#).

- MLton does not correctly parse case expressions nested within other matches. For example, the following fails.

```
fun f 0 y =
 case x of
 1 => 2
 | _ => 3
 | f _ y = 4
```

To do this in a program, simply parenthesize the case expression.

Allowing such expressions, although compliant with the Definition, would be a mistake, since using parentheses is clearer and no SML compiler has ever allowed them. Furthermore, implementing this would require serious yacc grammar rewriting followed by postprocessing.

- MLton rejects rebinding of constructors with `val rec` declarations, as in

```
val rec NONE = fn () => ()
```

The Definition (bizarrely) requires this program to type check, but to raise `Bind`.

We have no plans to change this behavior, as the Definition's behavior is clearly an error, a mismatch between the static semantics and the dynamic semantics.

- MLton does not hide the equality aspect of types declared in `abstype` declarations. So, MLton accepts programs like the following, while the Definition rejects them.

```
abstype t = T with end
val _ = fn (t1, t2 : t) => t1 = t2

abstype t = T with val a = T end
val _ = a = a
```

One consequence of this choice is that MLton accepts the following program, in accordance with the Definition.

```
abstype t = T with val eq = op = end
val _ = fn (t1, t2 : t) => eq (t1, t2)
```

Other implementations will typically reject this program, because they make an early choice for the type of `eq` to be `"a * "a -> bool` instead of `t * t -> bool`. The choice is understandable, since the Definition accepts the following program.

```
abstype t = T with val eq = op = end
val _ = eq (1, 2)
```

- MLton (re-)type checks each functor definition at every corresponding functor application (the compilation technique of defunctorization). One consequence of this implementation is that MLton accepts the following program, while the Definition rejects it.

```
functor F (X: sig type t end) = struct
 val f = id id
end
structure A = F (struct type t = int end)
structure B = F (struct type t = bool end)
val _ = A.f 10
val _ = B.f "dude"
```

On the other hand, other implementations will typically reject the following program, while MLton and the Definition accept it.

```
functor F (X: sig type t end) = struct
 val f = id id
end
structure A = F (struct type t = int end)
structure B = F (struct type t = bool end)
val _ = A.f 10
val _ = B.f false
```

See [DreyerBlume07](#) for more details.

## 327 UnsafeStructure

This module is a subset of the `Unsafe` module provided by SML/NJ, with a few extract operations for `PackWord` and `PackReal`.

```
signature UNSAFE_MONO_ARRAY =
 sig
 type array
 type elem

 val create: int -> array
 val sub: array * int -> elem
 val update: array * int * elem -> unit
 end

signature UNSAFE_MONO_VECTOR =
 sig
 type elem
 type vector

 val sub: vector * int -> elem
 end

signature UNSAFE =
 sig
 structure Array:
 sig
 val create: int * 'a -> 'a array
 val sub: 'a array * int -> 'a
 val update: 'a array * int * 'a -> unit
 end
 structure CharArray: UNSAFE_MONO_ARRAY
 structure CharVector: UNSAFE_MONO_VECTOR
 structure IntArray: UNSAFE_MONO_ARRAY
 structure IntVector: UNSAFE_MONO_VECTOR
 structure Int8Array: UNSAFE_MONO_ARRAY
 structure Int8Vector: UNSAFE_MONO_VECTOR
 structure Int16Array: UNSAFE_MONO_ARRAY
 structure Int16Vector: UNSAFE_MONO_VECTOR
 structure Int32Array: UNSAFE_MONO_ARRAY
 structure Int32Vector: UNSAFE_MONO_VECTOR
 structure Int64Array: UNSAFE_MONO_ARRAY
 structure Int64Vector: UNSAFE_MONO_VECTOR
 structure IntInfArray: UNSAFE_MONO_ARRAY
 structure IntInfVector: UNSAFE_MONO_VECTOR
 structure LargeIntArray: UNSAFE_MONO_ARRAY
 structure LargeIntVector: UNSAFE_MONO_VECTOR
 structure LargeRealArray: UNSAFE_MONO_ARRAY
 structure LargeRealVector: UNSAFE_MONO_VECTOR
 structure LargeWordArray: UNSAFE_MONO_ARRAY
 structure LargeWordVector: UNSAFE_MONO_VECTOR
 structure RealArray: UNSAFE_MONO_ARRAY
 structure RealVector: UNSAFE_MONO_VECTOR
 structure Real32Array: UNSAFE_MONO_ARRAY
 structure Real32Vector: UNSAFE_MONO_VECTOR
 structure Real64Array: UNSAFE_MONO_ARRAY
 structure Vector:
 sig
 val sub: 'a vector * int -> 'a
 end
 structure Word8Array: UNSAFE_MONO_ARRAY
```

```
structure Word8Vector: UNSAFE_MONO_VECTOR
structure Word16Array: UNSAFE_MONO_ARRAY
structure Word16Vector: UNSAFE_MONO_VECTOR
structure Word32Array: UNSAFE_MONO_ARRAY
structure Word32Vector: UNSAFE_MONO_VECTOR
structure Word64Array: UNSAFE_MONO_ARRAY
structure Word64Vector: UNSAFE_MONO_VECTOR

structure PackReal32Big : PACK_REAL
structure PackReal32Little : PACK_REAL
structure PackReal64Big : PACK_REAL
structure PackReal64Little : PACK_REAL
structure PackRealBig : PACK_REAL
structure PackRealLittle : PACK_REAL
structure PackWord16Big : PACK_WORD
structure PackWord16Little : PACK_WORD
structure PackWord32Big : PACK_WORD
structure PackWord32Little : PACK_WORD
structure PackWord64Big : PACK_WORD
structure PackWord64Little : PACK_WORD
end
```

## 328 Useless

`Useless` is an optimization pass for the `SSA IntermediateLanguage`, invoked from `SSASimplify`.

### 328.1 Description

This pass:

- removes components of tuples that are constants (use unification)
- removes function arguments that are constants
- builds some kind of dependence graph where
  - a value of ground type is useful if it is an arg to a primitive
  - a tuple is useful if it contains a useful component
  - a constructor is useful if it contains a useful component or is used in a `Case` transfer

If a useful tuple is coerced to another useful tuple, then all of their components must agree (exactly). It is trivial to convert a useful value to a useless one.

### 328.2 Implementation

- `useless.fun`

### 328.3 Details and Notes

It is also trivial to convert a useful tuple to one of its useful components — but this seems hard.

Suppose that you have a `ref/array/vector` that is useful, but the components aren't — then the components are converted to `unit`, and any primitive args must be as well.

Unify all handler arguments so that `raise/handle` has a consistent calling convention.

---

## 329 Users

Here is a list of companies, projects, and courses that use or have used MLton. If you use MLton and are not here, please add your project with a brief description and a link. Thanks.

### 329.1 Companies

- [Hardcore Processing](#) uses MLton as a [crosscompiler from Linux to Windows](#) for graphics and game software.
  - [CEX3D Converter](#), a conversion program for 3D objects.
  - [Interactive Showreel](#), which contains a crossplatform GUI-toolkit and a realtime renderer for a subset of RenderMan written in Standard ML.
  - various [games](#)
- [MathWorks/PolySpace Technologies](#) builds their product that detects runtime errors in embedded systems based on abstract interpretation.
- [Reactive Systems](#) uses MLton to build Reactis, a model-based testing and validation package used in the automotive and aerospace industries.

### 329.2 Projects

- [ADATE](#), Automatic Design of Algorithms Through Evolution, a system for automatic programming i.e., inductive inference of algorithms. ADATE can automatically generate non-trivial and novel algorithms written in Standard ML.
- [CIL](#), a compiler for SML based on intersection and union types.
- [ConCert](#), a project investigating certified code for grid computing.
- [Cooperative Internet hosting tools](#)
- [Guugelhupf](#), a simple search engine.
- [HaMLet](#), a model implementation of Standard ML.
- [KeplerCode](#), independent verification of the computational aspects of proofs of the Kepler conjecture and the Dodecahedral conjecture.
- [Metis](#), a first-order prover (used in the [HOL4 theorem prover](#) and the [Isabelle theorem prover](#)).
- [mlftpd](#), an ftp daemon written in SML. [TomMurphy](#) is also working on [replacements for standard network services](#) in SML. He also uses MLton to build his entries ([2001](#), [2002](#), [2004](#), [2005](#)) in the annual ICFP programming contest.
- [MLOPE](#), an offline partial evaluator for Standard ML.
- [RML](#), a system for developing, compiling and debugging and teaching structural operational semantics (SOS) and natural semantics specifications.
- [Skalpel](#), a type-error slicer for SML
- [SSA PRE](#), an implementation of Partial Redundancy Elimination for MLton.
- [Stabilizers](#), a modular checkpointing abstraction for concurrent functional programs.
- [Self-Adjusting SML](#), self-adjusting computation, a model of computing where programs can automatically adjust to changes to their data.
- [TL System](#), providing general-purpose support for rewrite-based transformation over elements belonging to a (user-defined) domain language.
- [Tina](#) (Time Petri net Analyzer)
- [Twelf](#) an implementation of the LF logical framework.
- [WaveScript/WaveScript](#), a sensor network project; the WaveScript compiler can generate SML (MLton) code.

### 329.3 Courses

- [Harvard CS-152](#), undergraduate programming languages.
  - [Høgskolen i Østfold IAI30202](#), programming languages.
-

## 330 Utilities

This page is a collection of basic utilities used in the examples on various pages. See

- [InfixingOperators](#), and
- [ProductType](#)

for longer discussions on some of these utilities.

```
(* Operator precedence table *)
infix 8 * / div mod (* +1 from Basis Library *)
infix 7 + - ^ (* +1 from Basis Library *)
infixr 6 :: @ (* +1 from Basis Library *)
infix 5 = <> > >= < <= (* +1 from Basis Library *)
infix 4 <\ \>
infixr 4 </ />
infix 3 o
infix 2 >|
infixr 2 |<
infix 1 := (* -2 from Basis Library *)
infix 0 before &

(* Some basic combinators *)
fun const x _ = x
fun cross (f, g) (x, y) = (f x, g y)
fun curry f x y = f (x, y)
fun fail e _ = raise e
fun id x = x

(* Product type *)
datatype ('a, 'b) product = & of 'a * 'b

(* Sum type *)
datatype ('a, 'b) sum = INL of 'a | INR of 'b

(* Some type shorthands *)
type 'a uop = 'a -> 'a
type 'a fix = 'a uop -> 'a
type 'a thunk = unit -> 'a
type 'a effect = 'a -> unit
type ('a, 'b) emb = ('a -> 'b) * ('b -> 'a)

(* Infixing, sectioning, and application operators *)
fun x <\ f = fn y => f (x, y)
fun f \> y = f y
fun f /> y = fn x => f (x, y)
fun x </ f = f x

(* Piping operators *)
val op>| = op</
val op|< = op\>
```

### 331 ValueRestriction

The value restriction is a rule that governs when type inference is allowed to polymorphically generalize a value declaration. In short, the value restriction says that generalization can only occur if the right-hand side of an expression is syntactically a value. For example, in

```
val f = fn x => x
val _ = (f "foo"; f 13)
```

the expression `fn x => x` is syntactically a value, so `f` has polymorphic type `'a -> 'a` and both calls to `f` type check. On the other hand, in

```
val f = let in fn x => x end
val _ = (f "foo"; f 13)
```

the expression `let in fn x => x end` is not syntactically a value and so `f` can either have type `int -> int` or `string -> string`, but not `'a -> 'a`. Hence, the program does not type check.

[The Definition of Standard ML](#) spells out precisely which expressions are syntactic values (it refers to such expressions as *non-expansive*). An expression is a value if it is of one of the following forms.

- a constant (`13`, `"foo"`, `13.0`, ...)
- a variable (`x`, `y`, ...)
- a function (`fn x => e`)
- the application of a constructor other than `ref` to a value (`Foo v`)
- a type constrained value (`v: t`)
- a tuple in which each field is a value (`v1, v2, ...`)
- a record in which each field is a value (`{l1 = v1, l2 = v2, ...}`)
- a list in which each element is a value (`[v1, v2, ...]`)

#### 331.1 Why the value restriction exists

The value restriction prevents a `ref` cell (or an array) from holding values of different types, which would allow a value of one type to be cast to another and hence would break type safety. If the restriction were not in place, the following program would type check.

```
val r: 'a option ref = ref NONE
val r1: string option ref = r
val r2: int option ref = r
val () = r1 := SOME "foo"
val v: int = valOf (!r2)
```

The first line violates the value restriction because `ref NONE` is not a value. All other lines are type correct. By its last line, the program has cast the string `"foo"` to an integer. This breaks type safety, because now we can add a string to an integer with an expression like `v + 13`. We could even be more devious, by adding the following two lines, which allow us to treat the string `"foo"` as a function.

```
val r3: (int -> int) option ref = r
val v: int -> int = valOf (!r3)
```

Eliminating the explicit `ref` does nothing to fix the problem. For example, we could replace the declaration of `r` with the following.

```
val f: unit -> 'a option ref = fn () => ref NONE
val r: 'a option ref = f ()
```

The declaration of `f` is well typed, while the declaration of `r` violates the value restriction because `f ()` is not a value.

### 331.2 Unnecessarily rejected programs

Unfortunately, the value restriction rejects some programs that could be accepted.

```
val id: 'a -> 'a = fn x => x
val f: 'a -> 'a = id id
```

The type constraint on `f` requires `f` to be polymorphic, which is disallowed because `id id` is not a value. MLton reports the following type error.

```
Error: z.sml 2.19.
 Can't bind type variable: 'a.
 in: val 'a (f): ('a -> 'a) = id id
```

MLton indicates the inability to make `f` polymorphic by saying that it can't bind the type variable `'a` at the declaration. MLton doesn't explicitly mention the value restriction, but that is the reason. If we leave the type constraint off of `f`

```
val id: 'a -> 'a = fn x => x
val f = id id
```

then the program succeeds; however, MLton gives us the following warning.

```
Warning: z.sml 2.1.
 Unable to locally determine type of variable: f.
 type: ??? -> ???
 in: val f = id id
```

This warning indicates that MLton couldn't polymorphically generalize `f`, nor was there enough context using `f` to determine its type. This in itself is not a type error, but it is a hint that something is wrong with our program. Using `f` provides enough context to eliminate the warning.

```
val id: 'a -> 'a = fn x => x
val f = id id
val _ = f 13
```

But attempting to use `f` as a polymorphic function will fail.

```
val id: 'a -> 'a = fn x => x
val f = id id
val _ = f 13
val _ = f "foo"
```

### 331.3 Alternatives to the value restriction

There would be nothing wrong with treating `f` as polymorphic in

```
val id: 'a -> 'a = fn x => x
val f = id id
```

One might think that the value restriction could be relaxed, and that only types involving `ref` should be disallowed. Unfortunately, the following example shows that even the type `'a -> 'a` can cause problems. If this program were allowed, then we could cast an integer to a string (or any other type).

```
val f: 'a -> 'a =
 let
 val r: 'a option ref = ref NONE
 in
 fn x =>
 let
 val y = !r
```

```

 val () = r := SOME x
 in
 case y of
 NONE => x
 | SOME y => y
 end
 end
end
val _ = f 13
val _ = f "foo"

```

The previous version of Standard ML took a different approach ([MilnerEtA190](#), [Tofte90](#), [ImperativeTypeVariable](#)) than the value restriction. It encoded information in the type system about when ref cells would be created, and used this to prevent a ref cell from holding multiple types. Although it allowed more programs to be type checked, this approach had significant drawbacks. First, it was significantly more complex, both for implementers and for programmers. Second, it had an unfortunate interaction with the modularity, because information about ref usage was exposed in module signatures. This either prevented the use of references for implementing a signature, or required information that one would like to keep hidden to propagate across modules.

In the early nineties, Andrew Wright studied about 250,000 lines of existing SML code and discovered that it did not make significant use of the extended typing ability, and proposed the value restriction as a simpler alternative ([Wright95](#)). This was adopted in the revised [Definition](#).

### 331.4 Working with the value restriction

One technique that works with the value restriction is [EtaExpansion](#). We can use eta expansion to make our `id id` example type check follows.

```

val id: 'a -> 'a = fn x => x
val f: 'a -> 'a = fn z => (id id) z

```

This solution means that the computation (in this case `id id`) will be performed each time `f` is applied, instead of just once when `f` is declared. In this case, that is not a problem, but it could be if the declaration of `f` performs substantial computation or creates a shared data structure.

Another technique that sometimes works is to move a monomorphic computation prior to a (would-be) polymorphic declaration so that the expression is a value. Consider the following program, which fails due to the value restriction.

```

datatype 'a t = A of string | B of 'a
val x: 'a t = A (if true then "yes" else "no")

```

It is easy to rewrite this program as

```

datatype 'a t = A of string | B of 'a
local
 val s = if true then "yes" else "no"
in
 val x: 'a t = A s
end

```

The following example (taken from [Wright95](#)) creates a ref cell to count the number of times a function is called.

```

val count: ('a -> 'a) -> ('a -> 'a) * (unit -> int) =
 fn f =>
 let
 val r = ref 0
 in
 (fn x => (r := 1 + !r; f x), fn () => !r)
 end
val id: 'a -> 'a = fn x => x
val (countId: 'a -> 'a, numCalls) = count id

```

The example does not type check, due to the value restriction. However, it is easy to rewrite the program, staging the ref cell creation before the polymorphic code.

```
datatype t = T of int ref
val count1: unit -> t = fn () => T (ref 0)
val count2: t * ('a -> 'a) -> (unit -> int) * ('a -> 'a) =
 fn (T r, f) => (fn () => !r, fn x => (r := 1 + !r; f x))
val id: 'a -> 'a = fn x => x
val t = count1 ()
val countId: 'a -> 'a = fn z => #2 (count2 (t, id)) z
val numCalls = #1 (count2 (t, id))
```

Of course, one can hide the constructor `T` inside a `local` or behind a signature.

### 331.5 Also see

- [ImperativeTypeVariable](#)

### 332 VariableArityPolymorphism

Standard ML programmers often face the problem of how to provide a variable-arity polymorphic function. For example, suppose one is defining a combinator library, e.g. for parsing or pickling. The signature for such a library might look something like the following.

```
signature COMBINATOR =
 sig
 type 'a t

 val int: int t
 val real: real t
 val string: string t
 val unit: unit t
 val tuple2: 'a1 t * 'a2 t -> ('a1 * 'a2) t
 val tuple3: 'a1 t * 'a2 t * 'a3 t -> ('a1 * 'a2 * 'a3) t
 val tuple4: 'a1 t * 'a2 t * 'a3 t * 'a4 t
 -> ('a1 * 'a2 * 'a3 * 'a4) t
 ...
 end
```

The question is how to define a variable-arity tuple combinator. Traditionally, the only way to take a variable number of arguments in SML is to put the arguments in a list (or vector) and pass that. So, one might define a tuple combinator with the following signature.

```
val tupleN: 'a list -> 'a list t
```

The problem with this approach is that as soon as one places values in a list, they must all have the same type. So, programmers often take an alternative approach, and define a family of `tuple<N>` functions, as we see in the `COMBINATOR` signature above.

The family-of-functions approach is ugly for many reasons. First, it clutters the signature with a number of functions when there should really only be one. Second, it is *closed*, in that there are a fixed number of tuple combinators in the interface, and should a client need a combinator for a large tuple, he is out of luck. Third, this approach often requires a lot of duplicate code in the implementation of the combinators.

Fortunately, using `Fold01N` and `products`, one can provide an interface and implementation that solves all these problems. Here is a simple pickling module that converts values to strings.

```
structure Pickler =
 struct
 type 'a t = 'a -> string

 val unit = fn () => ""

 val int = Int.toString

 val real = Real.toString

 val string = id

 type 'a accum = 'a * string list -> string list

 val tuple =
 fn z =>
 Fold01N.fold
 {finish = fn ps => fn x => concat (rev (ps (x, []))),
 start = fn p => fn (x, l) => p x :: l,
 zero = unit}
 z

 val ` =
```

```

fn z =>
 Fold01N.step1
 {combine = (fn (p, p') => fn (x & x', l) => p' x' :: "," :: p (x, l))}
 z
end

```

If one has  $n$  picklers of types

```

val p1: a1 Pickler.t
val p2: a2 Pickler.t
...
val pn: an Pickler.t

```

then one can construct a pickler for  $n$ -ary products as follows.

```

tuple `p1 `p2 ... `pn $: (a1 & a2 & ... & an) Pickler.t

```

For example, with `Pickler` in scope, one can prove the following equations.

```

"" = tuple $ ()
"1" = tuple `int $ 1
"1,2.0" = tuple `int `real $ (1 & 2.0)
"1,2.0,three" = tuple `int `real `string $ (1 & 2.0 & "three")

```

Here is the signature for `Pickler`. It shows why the `accum` type is useful.

```

signature PICKLER =
 sig
 type 'a t

 val int: int t
 val real: real t
 val string: string t
 val unit: unit t

 type 'a accum
 val ` : ('a accum, 'b t, ('a, 'b) prod accum,
 'z1, 'z2, 'z3, 'z4, 'z5, 'z6, 'z7) Fold01N.step1
 val tuple: ('a t, 'a accum, 'b accum, 'b t, unit t,
 'z1, 'z2, 'z3, 'z4, 'z5) Fold01N.t
 end

structure Pickler: PICKLER = Pickler

```

### 333 Variant

A *variant* is an arm of a datatype declaration. For example, the datatype

```
datatype t = A | B of int | C of real
```

has three variants: A, B, and C.

---

### 334 VesaKarvonen

Vesa Karvonen is a student at the [University of Helsinki](#). His interests lie in programming techniques that allow complex programs to be expressed clearly and concisely and the design and implementation of programming languages.



Things he'd like to see for SML and hopes to be able to contribute towards:

- A practical tool for documenting libraries. Preferably one that is based on extracting the documentation from source code comments.
- A good IDE. Possibly an enhanced SML mode (`esml-mode`) for Emacs. Google for [SLIME video](#) to get an idea of what he'd like to see. Some specific notes:
  - show type at point
  - robust, consistent indentation
  - show documentation
  - jump to definition (see [EmacsDefUseMode](#))

[EmacsBgBuildMode](#) has also been written for working with MLton.

- Documented and cataloged libraries. Perhaps something like [Boost](#), but for SML libraries. Here is a partial list of libraries, tools, and frameworks Vesa is or has been working on:
  - Asynchronous Programming Library ([README](#))
  - Extended Basis Library ([README](#))

- Generic Programming Library ([README](#))
- Pretty Printing Library ([README](#))
- Random Generator Library ([README](#))
- RPC (Remote Procedure Call) Library ([README](#))
- [SDL](#) Binding ([README](#))
- Unit Testing Library ([README](#))
- Use Library ([README](#))
- Windows Library ([README](#))

Note that most of these libraries have been ported to several [SML implementations](#).

---

### 335 WarnUnusedAnomalies

The `warnUnused` [MLBasis annotation](#) can be used to report unused identifiers. This can be useful for catching bugs and for code maintenance (e.g., eliminating dead code). However, the `warnUnused` annotation can sometimes behave in counter-intuitive ways. This page gives some of the anomalies that have been reported.

- Functions whose only uses are recursive uses within their bodies are not warned as unused:

```
local
fun foo () = foo () : unit
val bar = let fun baz () = baz () : unit in baz end
in
end
```

```
Warning: z.sml 3.5.
Unused variable: bar.
```

- Components of actual functor argument that are necessary to match the functor argument signature but are unused in the body of the functor are warned as unused:

```
functor Warning (type t val x : t) = struct
 val y = x
end
structure X = Warning (type t = int val x = 1)
```

```
Warning: z.sml 4.29.
Unused type: t.
```

- No component of a functor result is warned as unused. In the following, the only uses of `f2` are to match the functor argument signatures of functor `G` and functor `H` and there are no uses of `z`:

```
functor F(structure X : sig type t end) = struct
 type t = X.t
 fun f1 (_ : X.t) = ()
 fun f2 (_ : X.t) = ()
 val z = ()
end
functor G(structure Y : sig
 type t
 val f1 : t -> unit
 val f2 : t -> unit
 val z : unit
end) = struct
 fun g (x : Y.t) = Y.f1 x
end
functor H(structure Y : sig
 type t
 val f1 : t -> unit
 val f2 : t -> unit
 val z : unit
end) = struct
 fun h (x : Y.t) = Y.f1 x
end
functor Z() = struct
 structure S = F(structure X = struct type t = unit end)
 structure SG = G(structure Y = S)
 structure SH = H(structure Y = S)
end
structure U = Z()
val _ = U.SG.g ()
val _ = U.SH.h ()
```



### 336 WesleyTerpstra

Wesley W. Terpstra is a PhD student at the Technische Universitat Darmstadt (Germany).

Research interests

- Distributed systems (P2P)
- Number theory (Error-correcting codes)

My interest in SML is centered on the fact the the language is able to directly express ideas from number theory which are important for my work. Modules and Functors seem to be a very natural basis for implementing many algebraic structures. MLton provides an ideal platform for actual implementation as it is fast and has unboxed words.

Things I would like from MLton in the future:

- Some better optimization of mathematical expressions
  - IPv6 and multicast support
  - A complete GUI toolkit like mGTK
  - More supported platforms so that applications written under MLton have a wider audience
-

### 337 WholeProgramOptimization

Whole-program optimization is a compilation technique in which optimizations operate over the entire program. This allows the compiler many optimization opportunities that are not available when analyzing modules separately (as with separate compilation).

Most of MLton's optimizations are whole-program optimizations. Because MLton compiles the whole program at once, it can perform optimization across module boundaries. As a consequence, MLton often reduces or eliminates the run-time penalty that arises with separate compilation of SML features such as functors, modules, polymorphism, and higher-order functions. MLton takes advantage of having the entire program to perform transformations such as: defunctorization, monomorphisation, higher-order control-flow analysis, inlining, unboxing, argument flattening, redundant-argument removal, constant folding, and representation selection. Whole-program compilation is an integral part of the design of MLton and is not likely to change.

---

## 338 WishList

This page is mainly for recording recurring feature requests. If you have a new feature request, you probably want to query interest on one of the [mailing lists](#) first.

Please be aware of MLton's policy on [language changes](#). Nonetheless, we hope to provide support for some of the "immediate" [SuccessorML](#) proposals in a future release.

### 338.1 Support for link options in ML Basis files

Introduce a mechanism to specify link options in [ML Basis](#) files. For example, generalizing a bit, a ML Basis declaration of the form

```
option "option"
```

could be introduced whose semantics would be the same (as closely as possible) as if the option string were specified on the compiler command line.

The main motivation for this is that a MLton library that would introduce bindings (through [FFI](#)) to an external library could be packaged conveniently as a single MLB file. For example, to link with library `foo` the MLB file would simply contain:

```
option "-link-opt -lfoo"
```

Similar feature requests have been discussed previously on the mailing lists:

- <http://www.mlton.org/pipermail/mlton/2004-July/025553.html>
- <http://www.mlton.org/pipermail/mlton/2005-January/026648.html>

## 339 XML

[XML](#) is an [IntermediateLanguage](#), translated from [CoreML](#) by [Defunctorize](#), optimized by [XMLSimplify](#), and translated by [Monomorphise](#) to [SXML](#).

### 339.1 Description

[XML](#) is polymorphic, higher-order, with flat patterns. Every [XML](#) expression is annotated with its type. Polymorphic generalization is made explicit through type variables annotating `val` and `fun` declarations. Polymorphic instantiation is made explicit by specifying type arguments at variable references. [XML](#) patterns can not be nested and can not contain wildcards, constraints, flexible records, or layering.

### 339.2 Implementation

- [xml.sig](#)
- [xml.fun](#)
- [xml-tree.sig](#)
- [xml-tree.fun](#)

### 339.3 Type Checking

[XML](#) also has a type checker, used for debugging. At present, the type checker is also the best specification of the type system of [XML](#). If you need more details, the type checker ([type-check.sig](#), [type-check.fun](#)), is pretty short.

Since the type checker does not affect the output of the compiler (unless it reports an error), it can be turned off. The type checker recursively descends the program, checking that the type annotating each node is the same as the type synthesized from the types of the expressions subnodes.

### 339.4 Details and Notes

[XML](#) uses the same atoms as [CoreML](#), hence all identifiers (constructors, variables, etc.) are unique and can have properties attached to them. Finally, [XML](#) has a simplifier ([XMLShrink](#)), which implements a reduction system.

#### 339.4.1 Types

[XML](#) types are either type variables or applications of n-ary type constructors. There are many utility functions for constructing and destructing types involving built-in type constructors.

A type scheme binds list of type variables in a type. The only interesting operation on type schemes is the application of a type scheme to a list of types, which performs a simultaneous substitution of the type arguments for the bound type variables of the scheme. For the purposes of type checking, it is necessary to know the type scheme of variables, constructors, and primitives. This is done by associating the scheme with the identifier using its property list. This approach is used instead of the more traditional environment approach for reasons of speed.

#### 339.4.2 XmlTree

Before defining [XML](#), the signature for language [XML](#), we need to define an auxiliary signature `XML_TREE`, that contains the datatype declarations for the expression trees of [XML](#). This is done solely for the purpose of modularity — it allows the simplifier and type checker to be defined by separate functors (which take a structure matching `XML_TREE`). Then, `Xml` is defined as the signature for a module containing the expression trees, the simplifier, and the type checker.

Both constructors and variables can have type schemes, hence both constructor and variable references specify the instance of the scheme at the point of references. An instance is specified with a vector of types, which corresponds to the type variables in the scheme.

[XML](#) patterns are flat (i.e. not nested). A pattern is a constructor with an optional argument variable. Patterns only occur in `case` expressions. To evaluate a case expression, compare the test value sequentially against each pattern. For the first pattern that matches, destruct the value if necessary to bind the pattern variables and evaluate the corresponding expression. If no pattern matches, evaluate the default. All patterns of a case statement are of the same variant of `Pat.t`, although this is not enforced by ML's type system. The type checker, however, does enforce this. Because tuple patterns are irrefutable, there will only ever be one tuple pattern in a case expression and there will be no default.

[XML](#) contains value, exception, and mutually recursive function declarations. There are no free type variables in [XML](#). All type variables are explicitly bound at either a value or function declaration. At some point in the future, exception declarations may go away, and exceptions may be represented with a single datatype containing a `unit ref` component to implement genericity.

[XML](#) expressions are like those of [CoreML](#), with the following exceptions. There are no records expressions. After type inference, all records (some of which may have originally been tuples in the source) are converted to tuples, because once flexible record patterns have been resolved, tuple labels are superfluous. Tuple components are ordered based on the field ordering relation. [XML](#) eta expands primitives and constructors so that there are always fully applied. Hence, the only kind of value of arrow type is a lambda. This property is useful for flow analysis and later in code generation.

An [XML](#) program is a list of toplevel datatype declarations and a body expression. Because datatype declarations are not generative, the defunctorizer can safely move them to toplevel.

---

## 340 XMLShrink

XMLShrink is an optimization pass for the [XML IntermediateLanguage](#), invoked from [XMLSimplify](#).

### 340.1 Description

This pass performs optimizations based on a reduction system.

### 340.2 Implementation

- [shrink.sig](#)
- [shrink.fun](#)

### 340.3 Details and Notes

The simplifier is based on [Shrinking Lambda Expressions in Linear Time](#).

The source program may contain functions that are only called once, or not even called at all. Match compilation introduces many such functions. In order to reduce the program size, speed up later phases, and improve the flow analysis, a source to source simplifier is run on [XML](#) after type inference and match compilation.

The simplifier implements the reductions shown below. The reductions eliminate unnecessary declarations (see the side constraint in the figure), applications where the function is immediate, and case statements where the test is immediate. Declarations can be eliminated only when the expression is nonexpansive (see Section 4.7 of the [Definition](#)), which is a syntactic condition that ensures that the expression has no effects (assignments, raises, or nontermination). The reductions on case statements do not show the other irrelevant cases that may exist. The reductions were chosen so that they were strongly normalizing and so that they never increased tree size.

- ```
let x = e1 in e2
```

reduces to

```
e2 [x -> e1]
```

if $e1$ is a constant or variable or if $e1$ is nonexpansive and x occurs zero or one time in $e2$
- ```
(fn x => e1) e2
```

reduces to

```
let x = e2 in e1
```
- ```
e1 handle e2
```

reduces to

```
e1
```

if $e1$ is nonexpansive
- ```
case let d in e end of p1 => e1 ...
```

reduces to

```
let d in case e of p1 => e1 ... end
```

- `case C e1 of C x => e2`

reduces to

```
let x = e1 in e2
```

---

## 341 XMLSimplify

The optimization passes for the [XML IntermediateLanguage](#) are collected and controlled by the `XmlSimplify` functor (`xml-simplify.sig`, `xml-simplify.fun`).

The following optimization passes are implemented:

- [XMLSimplifyTypes](#)
- [XMLShrink](#)

The optimization passes can be controlled from the command-line by the options

- `-diag-pass <pass>` — keep diagnostic info for pass
  - `-drop-pass <pass>` — omit optimization pass
  - `-keep-pass <pass>` — keep the results of pass
  - `-xml-passes <passes>` — xml optimization passes
-

## 342 XMLSimplifyTypes

[XMLSimplifyTypes](#) is an optimization pass for the [XML IntermediateLanguage](#), invoked from [XMLSimplify](#).

### 342.1 Description

This pass simplifies types in an [XML](#) program, eliminating all unused type arguments.

### 342.2 Implementation

- [simplify-types.sig](#)
- [simplify-types.fun](#)

### 342.3 Details and Notes

It first computes a simple fixpoint on all the `datatype` declarations to determine which `datatype` tycon args are actually used. Then it does a single pass over the program to determine which polymorphic declaration type variables are used, and rewrites types to eliminate unused type arguments.

This pass should eliminate any spurious duplication that the [Monomorphise](#) pass might perform due to phantom types.

---

## 343 Zone

[Zone](#) is an optimization pass for the [SSA2 IntermediateLanguage](#), invoked from [SSA2Simplify](#).

### 343.1 Description

This pass breaks large [SSA2](#) functions into zones, which are connected subgraphs of the dominator tree. For each zone, at the node that dominates the zone (the "zone root"), it places a tuple collecting all of the live variables at that node. It replaces any variables used in that zone with offsets from the tuple. The goal is to decrease the liveness information in large [SSA](#) functions.

### 343.2 Implementation

- [zone.fun](#)

### 343.3 Details and Notes

Compute strongly-connected components to avoid put tuple constructions in loops.

There are two (expert) flags that govern the use of this pass

- `-max-function-size <n>`
- `-zone-cut-depth <n>`

Zone splitting only works when the number of basic blocks in a function is greater than `n`. The `n` used to cut the dominator tree is set by `-zone-cut-depth`.

There is currently no attempt to be safe-for-space. That is, the tuples are not restricted to containing only "small" values.

In the `HOL` program, the particular problem is the main function, which has 161,783 blocks and 257,519 variables — the product of those two numbers being about 41 billion. Now, we're not likely going to need that much space since we use a sparse representation. But even 1/100th would really hurt. And of course this rules out bit vectors.

## 344 ZZZOrphanedPages

The contents of these pages have been moved to other pages.

These templates are used by other pages.

- [CompilerPassTemplate](#)
  - [TalkTemplate](#)
-