Flow-Directed Closure Conversion for Typed Languages

Henry Cejtin[*], Suresh Jagannathan[**], and Stephen Weeks[***]

Abstract. This paper presents a new closure conversion algorithm for simply-typed languages. We have have implemented the algorithm as part of MLton, a whole-program compiler for Standard ML (SML). MLton first applies all functors and eliminates polymorphism by code duplication to produce a simply-typed program. MLton then performs closure conversion to produce a first-order, simply-typed program. In contrast to typical functional language implementations, MLton performs most optimizations on the first-order language, *after* closure conversion. There are two notable contributions of our work:

1. The translation uses a general flow-analysis framework which includes OCFA. The types in the target language fully capture the results of the analysis. MLton uses the analysis to insert coercions to translate between different representations of a closure to preserve type correctness of the target language program.
2. The translation is practical. Experimental results over a range of benchmarks including large real-world programs such as the compiler itself and the ML-Kit [25] indicate that the compile-time cost of flow analysis and closure conversion is extremely small, and that the dispatches and coercions inserted by the algorithm are dynamically infrequent.

1 Introduction

This paper presents a new closure conversion algorithm for simply-typed languages. We have implemented the algorithm as part of MLton[1] , a whole-program compiler for Standard ML (SML). MLton first applies all functors and eliminates polymorphism by code duplication to produce a simply-typed program. MLton then performs closure conversion to produce a first-order, simply-typed program. Unlike typical functional language implementations, MLton performs most optimizations on the first-order language, after closure conversion. The most important benefit of this approach is that numerous optimization techniques developed for other first-order languages can be immediately applied. In addition, a simply-typed intermediate language simplifies the overall structure

[*] Entertainment Decisions, Inc. henry@clairv.com
[**] NEC Research Institute,suresh@research.nj.nec.com
[***] Intertrust STAR Laboratories, sweeks@intertrust.com
[1] MLton is freely available under GPL from http://www.neci.nj.nec.com/PLS/MLton/.

of the compiler. Our experience with MLton indicates that simply-typed intermediate languages are sufficiently expressive to efficiently compile higher-order languages like Standard ML.

An immediate question that arises in pursuing this strategy concerns the representation of closures. Closure conversion transforms a higher-order program into a first-order one by representing each procedure with a tag identifying the code to be executed (typically a code pointer) when the procedure is applied, and an environment containing the values of the procedure's free variables. The code portion of a procedure is translated to take its environment as an extra argument.

Like previous work on defunctionalization [19, 3], the translation implements closures as elements of a datatype, and dispatches at call-sites to the appropriate code. We differ in that the datatypes in the target language express all procedures that may be called at the same call-site as determined by flow analysis. Consequently, the size of dispatches at calls is inversely related to the precision of the analysis.

Using dispatches instead of code pointers to express function calls has two important benefits: (1) the target language can remain simply-typed without the need to introduce existential types [16], and (2) optimizations can use different calling conventions for different procedures applied at the same call-site. However, if the simplicity and optimization opportunities afforded by using dispatches are masked by the overhead of the dispatch itself, this strategy would be inferior to one in which the code pointer is directly embedded within the closure record. We show that the cost of dispatches for the benchmarks we have measured is a small fraction of the benchmark's overall execution time. We elaborate on these issues in Sections 4 and 6.

Our approach extends the range of expressible flow analyses beyond that of previous work [26] by inserting coercions in the target program that preserve a closure's meaning, but change its type. Using coercions, the translation expresses higher-order flow information in the first-order target language in a form verifiable by the type system. Since the results of flow analysis are completely expressed in the types of the target program, ordinary optimizations performed on the target automatically take advantage of flow information computed on the source. In Section 4, we show that representations can be chosen so that coercions have no runtime cost.

Experimental results over a range of benchmarks including the compiler itself (approximately 47K lines of SML code) and the ML Kit (approximately 75K lines) indicate that the compile-time cost of flow analysis and closure conversion is small, and that local optimizations can eliminate almost all inserted coercions. Also, MLton often produces code significantly faster than the code produced by Standard ML of New Jersey [1].

The remainder of the paper is structured as follows. Section 2 describes the source and target languages for the closure converter. Section 3 defines the class of flow analyses that the translation can use. Section 4 presents the closure conversion algorithm. A detailed example illustrating the algorithm is given in Section 5. Section 6 describes MLton and presents experimental results. Sections 7 presents related work and conclusions.

2 Source and Target Languages

We illustrate our flow-directed closure conversion translation using the source language shown on the left-hand side of Figure 1. A program consists of a collection of datatype declarations followed by an expression. As in ML, a datatype declaration defines a new sum type along with constructors to create and discriminate among values of that type. The source language is a lambda calculus core augmented by constructor application, case, tuple construction, selection of tuple components, and exceptions. Exceptions are treated as elements of a datatype. The source language is simply-typed, where types are either type constructors, arrow types, or tuple types. We omit the type rules and assume that every expression and variable is annotated with its type. We write $e : \tau$ to mean that e has type τ. We write $x : \tau$ to mean that variable x has type τ. We assume that all bound variables in a program are distinct. We use *Exp*, *Bind*, *Lam*, *App*, and *Tuple* to name the sets of specific occurrences of subterms of the forms e, b, `fn` x `=>` e, $y\ z$, and `(...,` x`, ...)`, respectively, in the given program (occurrences can be defined formally using paths or unique expression labels). *TyCon* names the set of datatypes declared in a program.

Source Language

C	$\in$	*Con*
t	$\in$	*Tycon*
w, x, y, z	$\in$	*Var*
τ	$::=$	t
	$\vert$	τ `->` τ
	$\vert$	`... *` τ `* ...`
P	$::=$	`let ...` *data* `... in` e `end`
data	$::=$	`datatype` t `= ... |` C `of` τ`| ...`
e	$::=$	x
	$\vert$	`let` x `=` b `in` e `end`
b	$::=$	e
	$\vert$	`fn` w `=>` e
	$\vert$	$y\ z$
	$\vert$	$C\ y$
	$\vert$	`case` y `of ... |` $C\ z$ `=>` e `| ...`
	$\vert$	`(...,` y`, ...)`
	$\vert$	`#`i y
	$\vert$	`raise` y
	$\vert$	e_1 `handle` y `=>` e_2

Target Language

f	$\in$	*Func*
τ	$::=$	t
	$\vert$	`... *` τ `* ...`
P	$::=$	`let ...` *data* `... in`
		`let ...` *fun* `... in` e `end`
		`end`
data	$::=$	`datatype` t `= ... |` C `of` τ`| ...`
fun	$::=$	`fun` f`(...,` x`, ...) =` e
e	$::=$	x
	$\vert$	`let` x `=` b `in` e `end`
b	$::=$	e
	$\vert$	f`(...,` y`, ...)`
	$\vert$	$C\ y$
	$\vert$	`case` y `of ... |` $C\ z$ `=>` e `| ...`
	$\vert$	`(...,` y`, ...)`
	$\vert$	`#`i y
	$\vert$	`raise` y
	$\vert$	e_1 `handle` y `=>` e_2

Fig. 1. Source and target languages.

Like the source language, the target language (see right-hand side of Figure 1) is simply-typed, but without arrow types, since the target language does not contain lambda expressions. A target language program is prefixed by a collection of mutually recursive first-order functions, and function application explicitly specifies the first-order function to be called.

$$\begin{array}{lll} v \in \mathit{Value} & = & (\mathit{Lam} \times \mathit{Env}) + \mathit{Value}^* + (\mathit{Con} \times \mathit{Value}) \\ \rho \in \mathit{Env} & = & \mathit{Var} \to \mathit{Value} \end{array}$$

$$\boxed{\rho, e \hookrightarrow v/p}$$

$$\frac{}{\rho, x \hookrightarrow \rho(x)}$$

$$\frac{\rho, b \hookrightarrow v_b \qquad \rho[x \mapsto v_b], e \hookrightarrow v/p}{\rho, \texttt{let } x \texttt{ = } b \texttt{ in } e \texttt{ end} \hookrightarrow v/p}$$

$$\frac{\rho, b \hookrightarrow p}{\rho, \texttt{let } x \texttt{ = } b \texttt{ in } e \texttt{ end} \hookrightarrow p}$$

$$\boxed{\rho, b \hookrightarrow v/p}$$

$$\frac{}{\rho, \texttt{fn } w \texttt{ => } e \hookrightarrow \langle \texttt{fn } w \texttt{ => } e, \rho|_{FV(\texttt{fn } w \texttt{ => } e)} \rangle}$$

$$\frac{\rho(y) = \langle \texttt{fn } w \texttt{ => } e, \rho' \rangle \qquad \rho'[w \mapsto \rho(z)], e \hookrightarrow v}{\rho, y\ z \hookrightarrow v}$$

$$\frac{}{\rho, C\ y \hookrightarrow \langle C, (\rho(y)) \rangle}$$

$$\frac{\rho(y) = \langle C, v \rangle \qquad \rho[z \mapsto v], e \hookrightarrow v'}{\rho, \texttt{case } y \texttt{ of } \ldots \texttt{ | } C\ z \texttt{ => } e \texttt{ | } \ldots \hookrightarrow v'}$$

$$\frac{\rho(y) = \langle C, v \rangle \qquad \rho[z \mapsto v], e \hookrightarrow p}{\rho, \texttt{case } y \texttt{ of } \ldots \texttt{ | } C\ z \texttt{ => } e \texttt{ | } \ldots \hookrightarrow p}$$

$$\frac{}{\rho, (\ldots, y, \ldots) \hookrightarrow (\ldots, \rho(y), \ldots)}$$

$$\frac{\rho(y) = (\ldots, v_i, \ldots)}{\rho, \texttt{\#}i\ y \hookrightarrow v_i}$$

$$\frac{}{\rho, \texttt{raise } y \hookrightarrow [\rho(y)]}$$

$$\frac{\rho, e_1 \hookrightarrow v}{\rho, e_1 \texttt{ handle } y \texttt{ => } e_2 \hookrightarrow v}$$

$$\frac{\rho, e_1 \hookrightarrow [v_1] \qquad \rho[y \mapsto v_1], e_2 \hookrightarrow v_2}{\rho, e_1 \texttt{ handle } y \texttt{ => } e_2 \hookrightarrow v_2}$$

Fig. 2. Source language semantics.

We specify the source language semantics via the inductively defined relations in Figure 2. For example, expression evaluation defined via the relation written $\rho, e \hookrightarrow v/p$, is pronounced "in environment ρ, expression e evaluates either to value v or an exception packet p." In this regard, the semantics of exceptions is similar to the presentation given in [15]. We write $[v]$ to denote an exception packet containing the value v. A value is either a closure, a tuple of values, or a value built by application of a datatype constructor. The semantic rules for the target language are identical except for the rule for function application:

$$\frac{[\ldots\ x_i \mapsto \rho(y_i)\ \ldots], e \hookrightarrow v/p}{\rho, f\texttt{(}\ldots\texttt{,}\ y_i\texttt{,}\ \ldots\texttt{)} \hookrightarrow v/p}$$

where $\texttt{fun}\ f\texttt{(}\ldots\texttt{,}\ x_i\texttt{,}\ \ldots\texttt{)} = e$ is a function declaration in the program.

3 Flow Analysis

Our flow analysis is a standard monovariant analysis that uses abstract values to approximate sets of exact values:

$$a \in AVal = TyCon + \mathcal{P}(Lam) + AVal^*$$

An abstract value may either be a *Tycon*, which represents all constructed values of that type, a set of λ-occurrences, which represents a set of closures, or a sequence of abstract values, which represents a set of tuples.

Definition 1. *An abstract value a is consistent with a type τ if and only if one of the following holds:*

1. $a = t$ *and* $\tau = t$.
2. $a \in \mathcal{P}(Lam)$, $\tau = \tau_1$ `->` τ_2, *and for all* $f \in a$, $f : \tau_1$ `->` τ_2.
3. $a = (\ldots,\ a_i,\ \ldots)$, $\tau = \ldots\ \texttt{*}\ \tau_i\ \texttt{*}\ \ldots$, a_i *is consistent with* τ_i *for all* i.

We define our flow analysis as a type-respecting [12] function from variables, constructors, and exception packets to abstract values in the program.

Definition 2. *A flow is a function* $F : (Var + Con + \{\mathsf{packet}\}) \to AVal$ *such that*

1. *For all x in P, if $x : \tau$ then $F(x)$ is consistent with τ.*
2. *For all C in P, if C carries values of type τ then $F(C)$ is consistent with τ.*

Informally, $F(x)$ conservatively approximates the set of values that x may take on at runtime. Similarly, $F(C)$ over-approximates the set of values to which C may be applied at runtime. The special token packet models exception values; all exception values are collected into the abstract value $F(\mathsf{packet})$.

To formally specify the meaning of an analysis, we define a pair of relations by mutual induction. The first, between environments and flows ($\rho \sqsubseteq F$), describes when an environment is approximated by the flow.

$$\rho \sqsubseteq F \text{ if for all } x \in dom(\rho),\ \rho(x) \sqsubseteq_F F(x)$$

The second relation, between values and abstract values ($v \sqsubseteq_F a$), describes when a value is approximated by an abstract value (relative to a flow).

1. $\mathsf{C}\ v \sqsubseteq_F t$ if C is a constructor associated with datatype t, and $v \sqsubseteq_F F(\mathsf{C})$.

2. $(\ldots,\ v_i,\ \ldots) \sqsubseteq_F (\ldots,\ a_i,\ \ldots)$ if $v_i \sqsubseteq_F a_i$ for all i.
3. $\langle \texttt{fn}\ x\ \texttt{=>}\ e, \rho\rangle \sqsubseteq_F a$ if $\texttt{fn}\ x\ \texttt{=>}\ e \in a$ and $\rho \sqsubseteq F$.

Figure 3 defines a collection of safety constraints such that any flow meeting them will conservatively approximate the runtime behavior of the program. We use the following partial order on abstract values:

Definition 3. *$a \geq a'$ if and only if*

- *$a = t = a'$ for some $t \in$ TyCon,*
- *$a \supseteq a'$, where $a, a' \in \mathcal{P}(Lam)$, or*
- *$a = (\ldots,\ a_i,\ \ldots)$, $a' = (\ldots,\ a'_i,\ \ldots)$ and $a_i \geq a'_i$ for all i.*

Theorem 1. *If F is safe and $\rho \sqsubseteq F$ then*

- *if $\rho, e \hookrightarrow v$ then $v \sqsubseteq_F F(last(e))$.*
- *if $\rho, e \hookrightarrow [v]$ then $v \sqsubseteq_F F(\mathsf{packet})$.*
- *if $\rho, b \hookrightarrow v$ and x = $b \in P$ then $v \sqsubseteq_F F(x)$.*
- *if $\rho, b \hookrightarrow [v]$ then $v \sqsubseteq_F F(\mathsf{packet})$.*

Proof. By induction on $\rho, e \hookrightarrow v$ and $\rho, b \hookrightarrow v$ □

Definition 4. *The* last *variable of an expression, which yields the expression's value, is defined as follows:*

$$\begin{aligned} &last(x) &&= x \\ &last(\texttt{let}\ x\ \texttt{=}\ b\ \texttt{in}\ e\ \texttt{end}) &&= last(e) \end{aligned}$$

Definition 5. *A flow F is* safe *if and only if, for all $x = b$ in P,*

1. *if b is e, then $F(x) = F(last(e))$.*
2. *if b is $\texttt{fn}\ y\ \texttt{=>}\ e$, then $F(x) \geq \{\texttt{fn}\ y\ \texttt{=>}\ e\}$.*
3. *if b is $y\ z$, then for all $\texttt{fn}\ w\ \texttt{=>}\ e \in F(y)$,*
   (a) $F(w) \geq F(z)$, and
   (b) $F(x) \geq F(last(e))$
4. *if b is $C\ y$, then $F(C) \geq F(y)$.*
5. *if b is $x = \texttt{case}\ y\ \texttt{of}\ \ldots\ \texttt{|}\ C_i\ z_i\ \texttt{=>}\ e_i\ \texttt{|}\ \ldots$, then for all i,*
   (a) $F(z_i) = F(C_i)$, and
   (b) $F(x) \geq F(last(e_i))$
6. *if b is $\texttt{(}\ldots\texttt{,}\ y_i\texttt{,}\ \ldots\texttt{)}$, then $F(x) = (\ldots,\ F(y_i),\ \ldots)$.*
7. *if b is $\texttt{\#}i\ y$ and $F(y) = (\ldots,\ a_i,\ \ldots)$ then $F(x) = a_i$.*
8. *if b is $\texttt{raise}\ y$ then $F(\mathsf{packet}) \geq Fy$.*
9. *if b is $e_1\ \texttt{handle}\ z\ \texttt{=>}\ e_2$ then $F(z) \geq F(\mathsf{packet})$, $F(x) \geq F(last(e_1))$, and $F(x) \geq F(last(e_2))$.*

Fig. 3. Safety constraints on flows.

The constraints are standard for a monovariant control-flow analysis [9, 17] with the following two exceptions. First, rule 4 merges all arguments to a constructor. This is to avoid introducing recursive coercions, and to reduce the number of coercions performed at runtime. Second, we use "=" instead of "$\geq$" in some flow constraints to simplify the specification of the translation, although it is straightforward to incorporate the extra generality in practice. One can also prove that for any program, there is a minimum safe flow; this corresponds to the usual 0CFA. Another example of a safe flow is the unification-based flow analysis described by Henglein [11] and used by Tolmach and Oliva [26]. We can view this analysis as adhering to the safety constraints in Figure 3 with containment ($\geq$) replaced by equality in the rules.

4 Closure Conversion

Given a safe flow F for the following source program:

```
          let ...(datatype t = ... | C of τ| ...)... in e end
```

the closure conversion algorithm produces the following target program:

```
let datatype t = ... | C of 𝒯(F(C))| ...
    ...
    datatype 𝒯(L) = ... | 𝒞(L, fn x => e) of (... * 𝒯(F(y_i)) * ...)| ...
    ...
in let ...
       fun 𝒩(fn x => e)(r, x) = let ... y_i = #i r ... in ⟦e⟧ end
       ...
   in ⟦e⟧
   end
end
```

The translation inserts one datatype declaration for each set L that appears in the range of F, with one constructor for each λ-expression in L. We write $\mathcal{T}(L)$ to denote the new datatype for L and $\mathcal{C}(L,$ `fn` x `=>` $e)$ to denote the name of the constructor corresponding to `fn` x `=>` $e \in L$. The constructor's argument has the type of the tuple of free variables of `fn` x `=>` e, that is $(\ldots, y_i, \ldots)$. We extend $\mathcal{T}$ to abstract values by defining $\mathcal{T}(t) = t$ and $\mathcal{T}((\ldots,\ a_i,\ \ldots)) = \ldots * \mathcal{T}(a_i) * \ldots$

The translation also creates one function declaration for each λ-expression that occurs in the source program. The name of the target language first-order function for `fn` x `=>` e is denoted by $\mathcal{N}($`fn` x `=>` $e)$. Each function extracts all the free variables of the closure record passed as the first argument, and then continues with the translated body.

The translation uses auxiliary functions $[\![\bullet]\!] : \mathit{Exp} \rightarrow\ \mathit{Exp}$ and $[\![\bullet]\!]_x : \mathit{Bind} \rightarrow\ \mathit{Bind}$, which appear in Figure 4. The interesting cases in the translation are for λ-expressions and application. Rule 2b builds a closure record by applying the appropriate constructor to the tuple of the procedure's free variables. Rule 2c translates an application to a dispatch on the closure record of the procedure being applied. Because the safety constraints only require containment instead of equality, the translation inserts coercions at program points where the flow becomes less precise.

1. (a) $[\![$`let` x `=` b `in` e `end`$]\!] =$ `let` x `=` $[\![b]\!]_x$ `in` $[\![e]\!]$ `end`
   (b) $[\![x]\!] = x$
2. (a) $[\![e]\!]_x = [\![e]\!]$
   (b) $[\![$`fn` w `=>` $e]\!]_x = C$`(...,` y`, ...)`,
   where $C = \mathcal{C}(F(x),$ `fn` w `=>` $e)$ and $\mathsf{FV}($`fn` w `=>` $e) = \ldots\ y\ \ldots$.
   (c) $[\![y\ z]\!]_x =$ `case` y `of`

```
                  ...
                  | $\mathcal{C}(F(y),$ fn $w$ => $e)$ $r$ => let  $z'$ = $\mathcal{X}(z, F(z), F(w))$
                                                   $v$ = $\mathcal{N}($fn $w$ => $e)(r, z')$
                                                   $v'$ = $\mathcal{X}(v, F(last(e)), F(x))$
                                              in $v'$
                                              end
                  ...
```

   where there is one branch for each `fn` w `=>` $e \in F(y)$ and z', v, and v' are fresh.
   (d) $[\![C\ y]\!]_x =$

```
              let  $y'$ = $\mathcal{X}(y, F(y), F(C))$
                   $r$ = $C\ y'$
              in $r$
              end
```

   where y' and r are fresh variables.
   (e) $[\![$`case` y `of ... |` $C\ z$ `=>` e `| ...`$]\!]_x =$

```
   case $y$ of
     ...
     | $C$ $z$ => let  $r$ = $[\![e]\!]$
                     $r'$ = $\mathcal{X}(r, F(last(e)), F(x))$
                in $r'$
                end
     ...
```

   where r, r' are fresh variables.
   (f) $[\![$`(...,` y`, ...)`$]\!]_x =$ `(...,` y`, ...)`
   (g) $[\![$`#`$i\ y]\!]_x =$ `#`$i\ y$
   (h) $[\![$`raise` $y]\!]_x =$ `raise` y
   (i) $[\![e_1$ `handle` z `=>` $e_2]\!]_x =$

```
   let  $y_1$= $[\![e_1]\!]$
        $y_2$= $\mathcal{X}(y_1, F(last(e_1)), F(x))$
   in $y_2$ end
   handle $z$ => let  $y_3$= $e_2$
                      $y_4$= $\mathcal{X}(y_2, F(last(e_2)), F(last(x)))$
                 in $y_4$ end
```

Fig. 4. Closure conversion of expressions.

The coercion function $\mathcal{X}$, defined in Figure 5, changes the representation of a value from a more precise to a less precise type. For example, the translation of an application may require coercions at two points. First, if the abstract value of the argument is more precise than the formal, a coercion is inserted to change the argument's type to the formal's. Second, a coercion is required if the abstract value of the result is more precise than the abstract value of variable to which it becomes bound.

We define $\mathcal{X} : \mathit{Var} \times \mathit{AVal} \times \mathit{AVal} \to \mathit{Bind}$ by cases on abstract values.
(Note, $\mathcal{X}(x, a, a')$ is only defined when $a \leq a'$.)

1. if $a = a'$ then $\mathcal{X}(x, a, a') = x$.
2. $\mathcal{X}(x, (\ldots,\ a_i,\ \ldots), (\ldots,\ a'_i,\ \ldots)) =$
```
let  ...
     $y_i$ = #$i$ $x$
     $y'_i$ = $\mathcal{X}(y_i, a_i, a'_i)$
     ...
     $z'$ = (..., $y'_i$, ...)
in $z'$
end
```
   where $z', \ldots, y_i, y'_i, \ldots$ are fresh variables.
3. $\mathcal{X}(x, L, L') =$
```
case $x$ of
  ...
| $\mathcal{C}(L,$fn $x$ => $e)$ $r$ => $\mathcal{C}(L',$fn $x$ => $e)$ $r$
  ...
```
   where there is one branch for each `fn` x `=>` $e \in L$.

Fig. 5. The coercion function.

4.1 Practical Issues

Although for a simple type system we must express coercions as a case expression with each arm simply changing the constructor (and the type) representing the closure, it is easy to pick an underlying representation for these datatypes so that no machine code actually has to be generated. In terms of the underlying memory objects, all coercions are the identity. If these datatypes are all represented as a tag word (whose only function is to distinguish between the summands forming the datatype) followed by some fixed representation of the value being carried by that summand, then the only thing which might be changed by the coercion function is the tag word. It is thus easy to pick the tags so that they also don't change (for instance, use the address for the code of the procedure). However, we do not do this in MLton. As shown in Section 6, dynamic counts indicate coercions are so rare that their cost is unimportant. The advantage of allowing the coercions to change representations is that one can choose specialized representations for environment records.

The closure conversion algorithm is designed to be safe-for-space [1]. Note that each closure record is destructed at the beginning of each first order function. The alternative of replacing each reference to a closed-over variable with a selection from the closure record violates space safety because it keeps the entire record alive. Another possible violation is rule 2c, which can turn a tail-call into a non-tail-call by requiring a coercion after the call. However, since each such coercion corresponds to a step up the lattice of abstract values which is of finite height, the space usage of the program can only increase by a constant factor.

Finally, it is possible to share all of the dispatches generated for calls to a given set of λ-expressions. However, MLton does not do this, since it has not been necessary for performance.

5 Example

Consider the example in Figure 6.

```
let f = fn a => fn b => a
    g = fn c => fn d => d
    h = case ... of
             ...⇒ f
             ...⇒ g
    m = h 13
    _ = m 7
```

(a) Source program

$F(\texttt{f}) = \{\texttt{fn a}\}$
$F(\texttt{g}) = \{\texttt{fn c}\}$
$F(\texttt{h}) = \{\texttt{fn a}, \texttt{fn c}\}$
$F(\texttt{m}) = \{\texttt{fn b}, \texttt{fn d}\}$

(b) Flow

```
datatype t1 = C1 of unit   (* fn a *)
datatype t2 = C2 of int    (* fn b *)
datatype t3 = C3 of unit   (* fn c *)
datatype t4 = C4 of unit   (* fn d *)
datatype t5 = C5 of unit   (* fn a ∈ F(h) *)
            | C6 of unit   (* fn c ∈ F(h) *)
datatype t6 = C7 of int    (* fn b ∈ F(m) *)
            | C8 of unit   (* fn d ∈ F(m) *)
fun F (r, a) = C2 a        (* code for fn a *)
fun F' (r, b) =
  let a = #1 r in a end    (* code for fn b *)
fun G (r, c) = C4 ()       (* code for fn c *)
fun G' (r, d) = d          (* code for fn d *)
val f = C1()
val g = C3()
val h = case ... of
             ...⇒ case f of C1 r ⇒ C5 r
             ...⇒ case g of C3 r ⇒ C6 r
val m = case h of
          C5 r ⇒ (case F (r, 13) of C2 r ⇒ C7 r)
        | C6 r ⇒ (case G (r, 13) of C4 r ⇒ C8 r)
val _ = case m of
          C7 r ⇒ F' (r, 7)
        | C8 r ⇒ G' (r, 7)
```

(c) Target program

Fig. 6. Example.

The source appears in part (a), the 0CFA flow is in part (b), and the result of closure conversion appears in part (c). We use `fn a` to represent the entire λ-expression beginning with `fn a`. Consider the translation of the last expression, the call to `m`. Since `m` may be bound to a procedure corresponding to `fn b` or `fn d`, the call must dispatch appropriately. For the expression which defines `h`, each branch of the `case`-expression must coerce a procedure corresponding to a known λ-expression to one which is associated with an element of {`fn a`, `fn c`}. In the expression defining `m`, both a dispatch and a coercion occur: first a dispatch based on the λ-expression which provides the code for the `h` is required. Then, each arm of this case expression must coerce the result (a function with known code) to one associated with either `fn b` or `fn d`.

6 Experiments

We have have implemented the algorithm as part of MLton, a whole-program compiler for Standard ML. MLton does not support separate compilation, and takes advantage of whole program information in order to perform many optimizations. Here, we give a brief overview of the relevant compiler passes and intermediate languages. First, MLton translates the input SML program into an explicitly-typed, polymorphic intermediate language (XML)[8]. XML does not have any module level constructs. All functor applications are performed at compile-time[6], and all uses of structures and signatures are eliminated by moving declarations to the top-level and appropriately renaming variables. Next, MLton translates the XML to SXML (a simply-typed language) by *monomorphisation*, eliminating all uses of polymorphism by duplicating each polymorphic expression for each monotype at which it is used. After monomorphisation, small higher-order functions are duplicated; a size metric is used to prevent excessive code growth. MLton then performs flow analysis as described in Section 3 on the resulting SXML, and closure converts procedures to FOL (a first-order simply-typed language) via the algorithm described in Section 4. After a series of optimizations (e.g., inlining, tuple flattening, redundant argument elimination, and loop invariant code motion), the FOL program is translated to a C program, which is then compiled by `gcc`. Like [22], a trampoline is used to satisfy tail-recursion. To reduce trampoline costs, multiple FOL procedures may reside in the same C procedure; a dispatch on C procedure entry jumps to the appropriate code [7].

To demonstrate the practicality of our approach, we have measured its impact on compile time and code size for benchmarks with sizes up to 75K lines. Among the benchmarks, `knuth-bendix`, `life`, `lexgen`, `mlyacc`, and `simple` are standard [1]; `ratio-regions` is integer intensive; `tensor` is floating-point intensive, and `count-graphs` is mostly symbolic[2]. `MLton` is the compiler itself, and `kit` is the ML-kit [25, 24]. The benchmarks were executed on a 450 MHz Intel Xeon with 1 GB of memory.

In Table 1, we give the number of lines of SML for each benchmark, along with compile times both under SML/NJ (version 110.9.1)[3] and MLton. The number of lines does not include approximately 8000 lines of basis library code

[2] `ratio-regions` was written by Jeff Siskind (`qobi@research.nj.nec.com`), `tensor` was written by Juan Jose Garcia Ripoll (`worm@arrakis.es`), and `count-graphs` was written by Henry Cejtin (`henry@clairv.com`).

[3] Except for the `kit` which is run under SML/NJ version 110.0.3 because 110.9.1 incorrectly rejects the `kit` as being ill-typed.

that MLton prefixes to each program. The compile time given for SML/NJ is the time to batch compile the entire program. In order to improve the performance of the code generated by SML/NJ, the entire program is wrapped in a `local` declaration whose body performs an `exportFn`. For MLton, we give the total compile time, the time taken by flow analysis and closure conversion, and the percentage of compile time spent by `gcc` to compile the C code.

The flow analysis times are shorter than previous work [2, 10, 4] would suggest, for several reasons. First, the sets of abstract values are implemented using hash consing and the binary operations (in particular set union) are cached to avoid re-computation. Second, because of monomorphisation, running 0CFA on SXML is equivalent to the polyvariant analysis given in [12]. Thus, it is more precise than 0CFA performed directly on the (non-monomorphised) source alone, and hence fewer set operations are performed. Third, the analysis only tracks higher-order values. Finally, the analysis is less precise for datatypes than the usual *birthplace*[13] approach (see rules 4 and 5a in Figure 3). Also, unlike earlier attempts to demonstrate the feasibility of 0CFA [20] which were limited to small programs or intramodule analysis, our benchmarks confirm that flow analysis is practical for programs even in excess of 50K lines.

MLton compile-times are longer than SML/NJ. However, note that the ratio of MLton's to SML/NJ's compile-time does not increase as program size increases. We believe MLton's compile-time is in practice linear. In fact, `gcc` is a major component of MLton's compile-time, especially on large programs. We expect a native back-end to remove much of this time.

	lines		MLton			
Program	SML	SML/NJ	Total	Flow	Convert	`gcc`%
`count-graphs`	204	1.2	4.02	.01	.25	38%
`kit`	73489	1375.75	2456.39	1.34	27.96	82%
`knuth-bendix`	606	2.7	6.55	.01	.32	47%
`lexgen`	1329	4.5	19.52	.03	.78	53%
`life`	161	.9	3.2	.01	.16	41%
`MLton`	47768	637.5	1672.0	1.94	33.84	81%
`mlyacc`	7297	30.1	144.86	.10	2.34	38%
`ratio-regions`	627	2.2	6.22	.01	.35	34%
`simple`	935	4.7	34.11	.04	.87	54%
`tensor`	2120	9.7	10.12	.03	.32	30%
`tsp`	495	.8	3.56	.01	.22	30%

Table 1. Program sizes (lines) and compile times (seconds).

Table 2 gives various dynamic counts for these benchmarks to quantify the cost of closure conversion. To make the presentation tractable, the entries are in millions per second of the running time of the program. Nonzero entries less than .01 are written as ~0. SXML Known and Unknown measure the number of known and unknown procedure calls identified in the SXML program using only syntactic heuristics [1]. FOL Known indicates the number of known procedure calls remaining in the FOL program after flow analysis and all optimizations on the FOL program have been performed. The difference between SXML and FOL Known is due to inlining and code simplificaton. Dispatch indicates the number of case expressions introduced in the FOL program to express procedure calls where the flow set is not a singleton. Thus, the difference between Dispatch and Unknown gives a rough measure of the effectiveness of flow analysis above syntactic analyses in identifying the procedures applied at call-sites. Finally, Coerce indicates the number of coercions performed on closure tags to ensure that the closure's type adheres to the appropriate flow set.

	SXML		FOL		
Program	Known	Unknown	Known	Dispatch	Coerce
`count-graphs`	60.2	~0	1.0	0	0
`kit`	13.1	.11	5.8	.02	~0
`knuth-bendix`	28.8	~0	11.3	~0	0
`lexgen`	63.4	2.68	15.4	~0	0
`life`	28.4	0	22.3	0	0
`MLton`	14.5	.48	5.2	.34	.01
`mlyacc`	37.5	.03	10.6	~0	0
`ratio-regions`	119.4	0	14.3	0	0
`simple`	34.2	.26	6.2	.26	0
`tensor`	140.6	~0	7.6	~0	0
`tsp`	34.5	~0	3.4	~0	0

Table 2. Dynamic counts (millions/second).

For most benchmarks, monomorphisation, and aggressive syntactic inlining make most calls known. However, for several of the benchmarks, there still remain a significant number of unknown calls. Flow analysis uniformly helps in reducing this number. Indeed, the number of dispatches caused by imprecision in the analysis is always less than 5% of the number of calls executed. Notice also that the number of coercions performed is zero for the majority of the benchmarks; this means imprecision in the flow analysis rarely results in unwanted merging of closures with different representations.

Table 3 gives runtime results for both SML/NJ and MLton. Of course, because the two systems have completely different compilation strategies, optimizers, backends, and runtime systems, these numbers do not isolate the performance of our closure conversion algorithm. However, they certainly demonstrate its feasibility.

7 Related Work and Conclusions

Closure conversion algorithms for untyped target languages have been explored in detail [1, 21]. Algorithms that use a typed target language, however, must solve the problem created when procedures of the same type differ in the number and types of their free variables. Since closure conversion exposes the types of these

Program	SML/NJ (sec)	MLton (sec)	NJ/MLton
`count-graphs`	28.8	11.9	2.40
`kit`	27.5	30.9	.89
`knuth-bendix`	44.1	15.2	2.90
`lexgen`	52.7	31.8	1.66
`life`	51.5	54.2	.95
`MLton`	198.7	101.3	1.96
`mlyacc`	43.4	20.6	2.11
`ratio-regions`	122.5	18.9	6.48
`simple`	25.3	18.4	1.38
`tensor`	154.4	19.8	7.78
`tsp`	191.7	25.4	7.54

Table 3. Runtimes (in seconds) and ratio of SML/NJ to MLton.

variables through an explicit environment record, procedures having the same source-level type may compile to closures of different types. Minamide et al. [16] address this problem by defining a new type system for the target language that uses an existential type to hide the environment component of a closure record in the closure's type, exposing the environment only at calls. Unfortunately, the target language is more complex than the simply-typed λ-calculus and makes it difficult to express control-flow information. For example, the type system prevents expressing optimizations that impose specialized calling conventions for different closures applied at a given call-site.

An alternative to Minamide et al.'s solution was proposed by Bell et al. [3]. Their approach has the benefit of using a simply-typed target language, but does not express control-flow information in the target program. Inspired by a technique first described by Reynolds [19], they suggest representing closures as members of a datatype, with one datatype for each different arrow type in the source program. Tolmach and Oliva [26] extend Bell et al. by using a weak monovariant flow analysis based on type inference [11]. They refine the closure datatypes so that there is one datatype for each equivalence class of procedures as determined by unification. Although their approach does express flow analysis in a simply-typed target language, it is restricted to flow analyses based on unification. We differ from these approaches by using datatype coercions to produce a simply-typed target program and in our use of 0CFA.

Dimock et al. [5] describe a flow-directed representation analysis that can be used to drive closure conversion optimizations. Flow information is encoded in the type system through the use of intersection and union types. Like our work, their system supports multiple closure representations in a strongly-typed context. However, they support only a limited number of representation choices, and rely critically on a more complex type system to express these choices. Our work also uses flow information to make closure representation decisions, but does so within a simply-typed λ calculus.

Palsberg and O'Keefe[18] define a type system that accepts the same set of programs as 0CFA viewed as safety analysis. Their type system is based on simple types, recursive types, and subtyping. Although they do not discuss closure conversion, our coercions correspond closely to their use of subtyping. By inserting coercions, we remove the need for subtyping in the target language, and can use a simpler language based on simple types, sum types, and recursive types.

Our work is also related to other compiler efforts based on typed intermediate representations [23, 14]. Besides helping to verify the implementation of compiler optimizations by detecting transformations that violate type safety, typed intermediate languages expose representations (through types) useful for code generation. For example, datatypes in the target language describe environment representations as determined by flow analysis on the source language. Types therefore provide a useful bridge to communicate information across different compiler passes.

The results of our flow-directed closure conversion translation in MLton demonstrate the following:

1. First-order simply-typed intermediate languages are an effective tool for compilation of languages like ML.
2. The coercions and dispatches introduced by flow-directed closure conversion have negligible runtime cost.
3. Contrary to folklore, OCFA can be implemented to have negligible compile-time cost, even for large programs.

References

1. Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.
2. J. Michael Ashley. A practical and flexible flow analysis for higher-order languages. In *ACM Symposium on Principles of Programming Languages*, pages 195–207, January 1996.
3. Jeffrey M. Bell, Françoise Bellegarde, and James Hook. Type-driven defunctionalization. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 25–37, Amsterdam, The Netherlands, 9–11 June 1997.
4. Greg Defouw, David Grove, and Craig Chambers. Fast interprocedural class analysis. In *ACM Symposium on Principles of Programming Languages*, pages 222–236, January 1998.
5. Allyn Dimock, Robert Muller, Franklyn Turback, and J.B. Wells. Strongly-typed flow-directed representation transformations. In *International Conference on Functional Programming*, June 1997.
6. Matrin Elsman. Static interpretation of modules. In *International Conference on Functional Programming*, September 1999.
7. Marc Feeley, James Miller, Guillermo Rozas, and Jason Wilson. Compiling higher-order languages into fully tail-recursive portable c. Technical Report Technical Report 1078, Department of Computer Science, University of Montreal, 1997.
8. Robert Harper and John C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, April 1993.

9. Nevin Heintze. Set-based analysis of ML programs. In *ACM Conference on LISP and Functional Programming*, pages 306–317, 1994.
10. Nevin Heintze and David A. McAllester. Linear-time subtransitive control flow analysis. In *Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation (PLDI)*, pages 261–272, Las Vegas, Nevada, 15–18 June 1997. *SIGPLAN Notices* 32(5), May 1997.
11. Fritz Henglein. Simple closure analysis. Technical Report D-193, Department of Computer Science, University of Copenhagen, March 1992.
12. Suresh Jagannathan, Stephen T. Weeks, and Andrew K. Wright. Type-directed flow analysis for typed intermediate languages. In *International Static Analysis Symposium*, September 1997.
13. Neil D. Jones and Stephen S. Muchnick. *Flow Analysis and Optimization of LISP-like Structures*, chapter 4, pages 102–131. Prentice–Hall, 1981.
14. Simon L. Peyton Jones, John Launchbury, Mark Shields, and Andrew Tolmach. Bridging the gulf: a common intermediate language for ML and Haskell. In *ACM Symposium on Principles of Programming Languages*, pages 49–51, January 1998.
15. Robin Milner, Mads Tofte, Robert Harper, and David B. Macqueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.
16. Yasuhiko Minamide, J. Gregory Morrisett, and Robert Harper. Typed closure conversion. In *ACM Symposium on Principles of Programming Languages*, pages 271–283, St. Petersburg, FL, January 1996.
17. Jens Palsberg. Closure analysis in constraint form. *ACM Transactions on Programming Languages and Systems*, 17(1):47–62, January 1995.
18. Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. *ACM Transactions on Programming Languages and Systems*, 17(4):576–599, July 1995. Preliminary version in Principles of Programming Languages, pages 367–378, January 1995.
19. John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM Annual Conference*, pages 717–740, 1972.
20. Manuel Serrano and Pierre Weis. $1+1=1$: an optimizing Caml compiler. In *Workshop on ML and its applications*, Orlando, Florida, June 1994. ACM SIGPLAN. Also appears as INRIA RR-2301.
21. Zhong Shao and Andrew W. Appel. Space-efficient closure representations. In *ACM Conference on LISP and Functional Programming*, pages 150–161, Orlando, FL, Jun 1994.
22. David Tarditi, Anurag Acharya, and Peter Lee. No assembly required: Compiling Standard ML to C. *ACM Letters on Programming Languages and Systems*, June 1992. Appears as CMU-CS-90-187.
23. David Tarditi, J. Gregory Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *ACM Conference on Programming Language Design and Implementation*, May 1996.
24. Mads Tofte and Lars Birkedal. A region inference algorithm. *Transactions on Programming Languages and Systems*, 20(4):724–767, 1998.
25. Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Olesen, Peter Sestoft, and Peter Bertelsen. Programming with regions in the ML Kit. Technical Report Technical Report DIKU-TR-97/12, Department of Computer Science, University of Copenhagen, 1997.
26. Andrew Tolmach and Dino Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, July 1998.