Flow-Directed Closure Conversion for Typed Languages (Extended Summary)

Henry Cejtin* Suresh Jagannathan* Stephen Weeks*

Abstract

This paper presents a novel closure conversion strategy which is part of MLton, a whole-program compiler for Standard ML. Unlike other implementations, MLton performs closure-conversion at an early stage to translate programs to a simply-typed first-order intermediate language upon which most optimizations are performed. Closure conversion is guided by a global control-flow analysis. Like previous work on defunctionalization, the translation implements closures as elements of datatypes, and uses dispatches at certain call-sites to select the appropriate function to call. However, our use of control-flow analysis leads to a significantly improved translation.

There are two notable contributions of our work:

1. The translation uses a general flow-analysis framework which includes OCFA. The datatypes in the target language 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 [TBE+97] 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. Runtime performance is often more than a factor of two faster than Standard ML of New Jersey.

*Computer Science Division, NEC Research Institute, 4 Independence Way, Princeton, NJ 08540. {henry,suresh,sweeks}@research.nj.nec.com.

1 Introduction

This paper presents a closure conversion strategy which is part of MLton[1], novel compiler for Standard ML. Unlike other implementations, MLton performs closure-conversion at an early stage to translate programs to a simply-typed first-order intermediate language. The most important benefit of this approach is that once a program is thus transformed, numerous optimization techniques developed for other first-order languages can be immediately applied. In addition, a simply-typed intermediate language simplifies the overall structure 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.

The most 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 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. When a closure is applied, free variables in its body are first bound by selecting their values from the environment before the rest of the procedure body is evaluated.

Like previous work on defunctionalization [Rey72, BBH97], 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 [MMH96], 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 5 and 7.

Our approach extends the range of expressible flow analyses beyond that of previous work [TO98] by inserting coercions in the target program that preserve a closure's meaning, but change its type. In Section 5.1, we show that representations can be chosen so that coercions have no runtime cost. 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. Moreover, unlike earlier approaches, coercions permit the compiler to use specialized closure representations. Experimental results indicate that local optimizations can eliminate almost all the inserted coercions.

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. Our results also reveal that the compiler produces code often significantly faster than the code produced by Standard ML of New Jersey [App92].

[1] MLton is freely available from `http://www.neci.nj.nec.com/PLS/MLton/`.

The summary is structured as follows. We present related work in the next section. Section 3 describes the source and target languages for the closure converter. Section 4 defines the class of flow analyses that the translation can use. Section 5 presents the closure conversion algorithm. A detailed example illustrating the algorithm is given in Section 6. Section 7 describes MLton and presents experimental results. Section 8 concludes.

2 Related Work

Closure conversion algorithms for untyped target languages have been explored in detail [App92, SA94]. 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 variables through an explicit environment record, procedures having the same source-level type may compile to closures of different types. Minamide et al. [MMH96] 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. [BBH97]. 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 [Rey72], they suggest representing closures as members of a datatype, with one datatype for each different arrow type in the source program. For a given datatype, there is one constructor for each procedure of that type in the program. The translation of a λ-expression is the application of its constructor to a tuple of the function's free variables. A call to an unknown procedure is replaced with a dispatch on the closure datatype that branches to the appropriate piece of code, passing along the argument and the environment record. Their approach also handles polymorphism, essentially by expanding polymorphic functions during closure conversion.

However, this approach to translating procedures is very imprecise, and does not support specialized closure representations. For example, consider procedures `f` and `h` with type $\tau_1 \to \tau_2$, and procedure `g` with type $(\tau_1 \to \tau_2) \to \tau_3$. Now, suppose the following three calls occur in the program: `f(...)`, `g(f)`, and `g(h)`. Closure conversion as described above would lead to an environment record for `f` and `h` occurring within a datatype representing procedures with type $\tau_1 \to \tau_2$. Unfortunately, since `f`'s environment is wrapped within a constructor, the direct call to `f` must still perform a dispatch to extract `f`'s environment. It is not possible to directly jump to `f`'s code even in the case where `f` is known (as in the first application) since the closure tag (represented as the constructor) must be first "stripped" before performing the call.

Tolmach and Oliva [TO98] extend Bell et al. by using a weak monovariant flow analysis based on type inference [Hen92]. 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. As before, there is a unique constructor for every closure, and thus their algorithm also does not easily support specialized closure implementations.

We differ from these approaches by using datatype coercions to produce a simply-typed target program. Our translation inserts coercions whenever different flow sets merge, as can happen at calls or conditionals. Informally, a coercion changes closure tags on the procedures which may flow into a call to ensure that all branches of the resulting dispatch in the target program have the same type.

Dimock et al. [DMTW97] 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[PO95] 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 [TMC+96, JLST98, Sha97]. 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.

3 Language

We illustrate our flow-directed closure-conversion translation using the source language shown in Figure 1. A program consists of a collection of datatype declarations followed by an expression in A-normal form [FSDF93]. 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 contains a lambda calculus core augmented by constructor application, case, tuple construction, and selection of tuple components. 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 $e : \tau$ $(x : \tau)$ to mean that e (x) has type τ.

We assume that all bound variables in a program are distinct. We use *ExpOcc*, *BindOcc*, *LamOcc*, *AppOcc*, and *TupleOcc* to name the sets of specific occurrences of subterms of the forms e, b, $\lambda x.e$, $y\ z$, and $(x_1, \ldots, x_n)$, 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.

The target language (see Figure 2) is like the source language, except that there are no lambda or application expressions. Instead, the program is prefixed by a collection of mutually recursive first-order functions (corresponding to the code part of closure-converted procedures), each taking an environment tuple as its first argument. Procedure application explicitly specifies the first-order function to be called, and passes the environment record as the first argument. The target

$$
\begin{array}{rcl}
C & \in & \mathit{Con} \\
t & \in & \mathit{Tycon} \\
w, x, y, z & \in & \mathit{Var} \\
\tau & ::= & t \mid \tau \to \tau \mid \tau_1 * \ldots * \tau_n \\
P & ::= & \mathit{dataDec}^* \ e \\
\mathit{dataDec} & ::= & \mathsf{datatype}\ t\ =\ (C\ \mathsf{of}\ \tau)^* \\
b & ::= & e \mid \lambda w.e \mid y\ z \\
 & \mid & C\ y \mid \mathsf{case}\ y\ \mathsf{of}\ (C\ x \Rightarrow e)^* \\
 & \mid & (y_1, \ldots, y_n) \mid \#i\ y \\
e & ::= & x \mid \mathsf{let}\ x = b\ \mathsf{in}\ e
\end{array}
$$

Figure 1: Source Language

$$
\begin{array}{rcl}
\tau & ::= & t \mid \tau_1 * \ldots * \tau_n \\
P & ::= & \mathit{dataDec}^* \ \mathit{funDec}^* \ e \\
\mathit{dataDec} & ::= & \mathsf{datatype}\ t\ =\ (C\ \mathsf{of}\ \tau)^* \\
\mathit{funDec} & ::= & \mathsf{fun}\ f((x_1, \ldots, x_n), y) = e \\
b & ::= & e \mid f(x, y) \\
 & \mid & C\ y \mid \mathsf{case}\ y\ \mathsf{of}\ (C\ x \Rightarrow e)^* \\
 & \mid & (y_1, \ldots, y_n) \mid \#i\ y \\
e & ::= & x \mid \mathsf{let}\ x = b\ \mathsf{in}\ e
\end{array}
$$

Figure 2: Target Language

language is also simply-typed, where types are either type constructors or tuple types.

We specify the semantics via the inductively defined relations in Figure 3. The relation is written $e, \rho \hookrightarrow v$, and pronounced "in environment ρ, expression e evaluates to value v." A value is either a closure, a tuple of values, or built by application of a datatype constructor. The semantic rules for the target language are identical except for the rule for function application:

$$
\frac{\rho(r) = (v_1, \ldots, v_n) \qquad [x_i \mapsto v_i, y \mapsto \rho(x)], e \hookrightarrow v}{\rho, f(r, x) \hookrightarrow v}
$$

where $\mathsf{fun}\ f((x_1, \ldots, x_n), y) = e$ is a function declaration in the program.

4 Flow Analysis

Our flow analysis uses abstract values which approximate sets of exact values:

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

$$
\begin{array}{rcrcl}
v & \in & \mathit{Value} & = & (\mathit{LamOcc} \times \mathit{Env}) + \mathit{Value}^* + (\mathit{Constructor} \times \mathit{Value}) \\
\rho & \in & \mathit{Env} & = & \mathit{Var} \to \mathit{Value}
\end{array}
$$

$$\boxed{\rho, e \hookrightarrow v} \qquad \boxed{\rho, b \hookrightarrow v}$$

$$\frac{}{x, \rho \hookrightarrow \rho(x)} \qquad \frac{\rho, b \hookrightarrow v_b \quad \rho[x \mapsto v_b], e \hookrightarrow v}{\rho, \mathsf{let}\ x = b\ \mathsf{in}\ e \hookrightarrow v}$$

$$\frac{}{\rho, \lambda x.e \hookrightarrow \langle \lambda x.e, \rho|_{FV(\lambda x.e)} \rangle}$$

$$\frac{\rho(y) = \langle \lambda x.e, \rho' \rangle \quad \rho'[x \mapsto \rho(z)], e \hookrightarrow v}{\rho, (y\ z) \hookrightarrow v}$$

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

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

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

$$\frac{\rho(x) = C\ v \quad \rho[x \mapsto v], e \hookrightarrow v}{\rho, \mathsf{case}\ x\ \mathsf{of}\ \ldots C\ x \Rightarrow e \ldots \hookrightarrow v}$$

Figure 3: Source language semantics

An abstract value that 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 4.1 *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. $a \in \mathcal{P}(\mathit{LamOcc})$, $\tau = \tau_1 \to \tau_2$, *and for all* $f \in a$, $f : \tau_1 \to \tau_2$.
3. $a = a_1 * \ldots * a_n$, $\tau = \tau_1 * \ldots * \tau_n$, *and a_i is consistent with τ_i for all* $1 <= i <= n$

We will define our flow analysis as a type-respecting [JWW97] function from variables and constructors to abstract values in the program:

Definition 4.2 *A flow is a function* $F \in (\mathit{Var} + \mathit{Constructor}) \to \mathit{AVal}$ *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. 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.

Definition 4.4 The *last* variable of an expression, which yields the expression's value, is defined as follows:

$$\begin{array}{rcl} last(x) & = & x \\ last(\mathsf{let}\ x = b\ \mathsf{in}\ e) & = & last(e) \end{array}$$

Definition 4.5 *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 $\lambda y.e$, then $F(x) \geq \{\lambda y.e\}$.*
3. *if b is $y\ z$, then for all $\lambda w.e \in F(y)$,*
   (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 $x = \mathsf{case}\ y\ \mathsf{of}\ (C_i\ z_i \Rightarrow e_i)^*$, then for all i,*
   (a) *$F(z_i) = F(C_i)$, and*
   (b) *$F(x) \geq F(last(e_i))$*
6. *if b is $(y_1, \ldots, y_n)$, then $F(x) = (F(y_1), \ldots, F(y_n))$.*
7. *if b is $\#i\ y$ then $F(x) = \#i\ (F(y))$, where $\#i\ (a_1, \ldots, a_n) = a_i$.*

Figure 4: Safety constraints on flows.

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

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 t is C a constructor associated with type t. and $v \sqsubseteq_F F(\mathsf{C})$.
2. $(v_1, \ldots, v_n) \sqsubseteq_F (a_1, \ldots, a_n)$ if $v_i \sqsubseteq_F a_i$ for all $1 \leq i \leq n$.
3. $\langle \lambda x.e, \rho \rangle \sqsubseteq_F a$ if $\lambda x.e \in a$ and $\rho \sqsubseteq F$.

We impose the following ordering on abstract values:

Definition 4.3 *$a \geq a'$ if and only if*

- *$a = t = a'$ for some $t \in TyCon$,*
- *$a \supseteq a'$, where a and a' are in $\mathcal{P}(LamOcc)$, or*
- *$a = (a_1, \ldots, a_n), a' = (a'_1, \ldots, a'_n)$ and $a_i \geq a'_i$ for all $1 \leq i \leq n$.*

Figure 4 defines a collection of safety constraints such that any flow meeting them will conservatively approximate the runtime behavior of the program.

Theorem 4.6 (Safety $\Longrightarrow$ Soundness) *If F is safe, $\rho \sqsubseteq F$, and $\rho, e \hookrightarrow v$ then $v \sqsubseteq_F F(last(e))$.*

The constraints are standard for a monovariant control-flow analysis [Hei94, Pal95] 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 [Hen92] and used by Tolmach and Oliva [TO98].

5 Translation

Figures 5 and 6 define the closure converter. In the translation, new datatypes correspond to the sets of λ-occurrences computed by flow analysis. Closures are represented as constructed values of those datatypes. Coercions are inserted at each program point where the flow becomes less precise (because the safety constraints only require containment instead of equality). Each application is replaced by a dispatch on the closure (now simply a tagged environment), each branch of which calls the appropriate function on the environment and the argument.

We choose to represent function calls via dispatches (rather than directly embedding a code-pointer within the closure record) for two reasons. First, we can specialize calls to different procedures applied at a given call-site more easily. For example, argument flattening is a particularly important optimization in languages like SML which assume all arguments are implicitly tupled. By representing procedure calls as dispatches, the same argument tuple can be flattened in different ways depending upon the specific procedure being applied. Second, using dispatches rather than code-pointers allows us to use a simply-typed target language. The alternative would be a more complicated type system [MMH96] using existential types to "hide" the code-pointer.

The translation uses the following auxiliary functions:

- The function $\mathcal{T} : AVal \rightarrow Type$ returns the target language type corresponding to an abstract value. $\mathcal{T}(t) = t$, $\mathcal{T}$ applied to a set L of lambda occurrences returns the new datatype for L, and $\mathcal{T}((a_1, \ldots, a_n)) = \mathcal{T}(a_1) * \mathcal{T}(a_2) * \ldots * \mathcal{T}(a_n)$.

- The function $\mathcal{C} : \mathcal{P}(LamOcc) \times LamOcc \rightarrow Constructor$ is defined such that $\mathcal{C}(L, \lambda x.e)$ where $\lambda x.e \in L$ returns the name of the constructor in the target language which injects the tuple of free variables of $\lambda x.e$ into $\mathcal{T}(L)$.

- The function $\mathcal{R} : LamOcc \rightarrow Type$ is defined such that $\mathcal{R}(\lambda x.e)$ is the type of the environment record for $\lambda x.e$, i.e., $\mathcal{T}(F(x_1)) * \ldots * \mathcal{T}(F(x_n))$ where $x_i \in FV(\lambda x.e)$.

- $\mathcal{N} : LamOcc \rightarrow Var$ returns the name of the target language first-order function for $\lambda x.e$.

- $\mathcal{X} : Var \times AVal \times AVal \rightarrow BindOcc$ coerces between different representations and is given in Figure 5.

The translation is defined by the function:

$$[\![\bullet]\!] : Exp \rightarrow \ Exp$$

Definition 5.1 (Coerce) We define $\mathcal{X} : Var \times AVal \times AVal \rightarrow BindOcc$ as cases over 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. If a and a' are in $AVal^*$ then $\mathcal{X}(x, a, a')$ returns

   let $\cdots$
   $\quad y_i = \#i\ x$
   $\quad y_i' = \mathcal{X}(y_i, \#i\ a, \#i\ a')$
   $\quad \cdots$
   $\quad z' = (\ldots, y_i', \ldots)$
   in z'

   where $z', \ldots, y_i, y_i', \ldots$ are fresh variables.
3. If a and a' are in $\mathcal{P}(LamOcc)$ then $\mathcal{X}(x, a, a')$ returns

   case x of
   $\quad \ldots$
   $\quad C\ r \Rightarrow C'\ r$
   $\quad \ldots$

   where there is one branch for each $\lambda x.e \in a$ where $C = \mathcal{C}(a, \lambda x.e)$ and $C' = \mathcal{C}(a', \lambda x.e)$

Figure 5: The coercion function.

and is given in Figure 6. To translate a program, we first copy all the datatype declarations from the source language program. Next, for each $s = \{\ldots, \lambda x_i.e_i, \ldots\}$ which appears in the range of our flow analysis, we add the datatype datatype $t = \ldots C_i$ of $\tau_i \ldots$ where $t = \mathcal{T}(s)$, $C_i = \mathcal{C}(s, \lambda x_i.e_i,)$, and $\tau_i = \mathcal{R}(\lambda x_i.e_i)$. Then, for each $\lambda x.e$ which appears in the source program, we add a function declaration of the form fun$f((\ldots, y_i, \ldots), x) = [\![e]\!]$ where the y_i's correspond to the source program procedure's free variables. Finally, the body of the program, e, is translated to $[\![e]\!]$.

We highlight some of the interesting cases in the translation. Rule 4 builds a closure record by applying the appropriate constructor to the tuple of the procedure's free variables. Rule 5 translates an application to a dispatch on the closure record of the procedure being applied. There is a branch for each λ-occurrence that can flow into the function position. Coercions may be required at two points in the translation of an application. First, if the abstract value of the argument is more precise than the formal, a coercion must be inserted to change the actual parameter's tag to match the expected type of the formal. 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. Thus, coercions change the tag of closure records to conform to the flow set of the context in which they are being used.

5.1 Coercions

The proof of Theorem 5.2 shows that coercions preserve well-typing.

Theorem 5.2 (Well-typing of Coercions) *If $x : \mathcal{T}(a_1)$ and $a_1 \leq a_2$, then $\mathcal{X}(x, a_1, a_2) : \mathcal{T}(a_2)$.*

1. $[\![\text{let } x = b \text{ in } e]\!] \longrightarrow \text{let } [\![x = b]\!] \text{ in } [\![e]\!]$
2. $[\![x]\!] \longrightarrow x$
3. $[\![x = e]\!] \longrightarrow x = e$
4. $[\![x = \lambda w.e]\!] \longrightarrow C(y_1, \ldots, y_n)$ where $C = \mathcal{C}(F(x), \lambda w.e)$
   $y_1, \ldots, y_n = \mathsf{FV}(\lambda w.e)$
5. $[\![x = y\ z]\!] \longrightarrow x = \text{case } y \text{ of}$

$$\begin{array}{l} \ldots \\ C\ r \Rightarrow \\ \quad \text{let } z' = \mathcal{X}(z, F(z), F(w)) \\ \quad \quad\ \ v\ = f\ (r, z') \\ \quad \quad\ \ v' = \mathcal{X}(v, F(last(e)), F(x)) \\ \quad \text{in } v' \\ \ldots \end{array}$$

   where there is one branch for each $\lambda w.e \in F(y)$ where
   $C = \mathcal{C}(F(y), \lambda w.e)$
   $f = \mathcal{N}(\lambda w.e)$
   z', v, v' are fresh variables
6. $[\![x = C\ y]\!] \longrightarrow x = \text{let } y' = \mathcal{X}(y, F(y), F(C))$

$$\begin{array}{l} \quad\quad r\ = C\ y' \\ \text{in } r \end{array}$$

   where y' and r are fresh variables
7. $[\![x = \text{case } y \text{ of } C_i\ x_i \Rightarrow e_i \ldots]\!] \longrightarrow x = \text{case } y \text{ of}$

$$\begin{array}{l} \ldots \\ C\ x \Rightarrow \\ \quad \text{let } r = [\![e_i]\!] \\ \quad\quad\ \ r' = \mathcal{X}(r, F(last(e)), F(x)) \\ \quad \text{in } r' \\ \ldots \end{array}$$

   where r, r' are fresh variables
8. $[\![x = (y_1, \ldots, y_n)]\!] \longrightarrow x = (y_1, \ldots, y_n)$
9. $[\![x = \#i\ y]\!] \longrightarrow x = \#i\ y$

Figure 6: Translation.

Note, although for our 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).

We do not do this in MLton. As shown in Section 7, dynamic counts indicate coercions are so rare that their cost is largely unimportant. The advantage of allowing the coercions to change representations is that one can chose specialized representations for environment records. MLton uses this, for example, to flatten nested tuples.

5.2 Correctness

To show that our closure conversion algorithm preserves the meaning of programs, we first define $\mathcal{S}$ which converts values and environments from the source semantics to the target semantics. We overload $\mathcal{S}$ for both $\mathit{Value} \times \mathit{AVal} \to \mathit{Value}$ and $\mathit{Env} \to \mathit{Env}$, which are defined inductively by

- $\mathcal{S}(\rho)(x) = \mathcal{S}(\rho(x), F(x))$
- $\mathcal{S}(C\ v, a) = C\ \mathcal{S}(v, F(C))$
- $\mathcal{S}((\ldots, v_i, \ldots), a) = (\ldots, \mathcal{S}(v_i, \#i\ a), \ldots)$
- $\mathcal{S}(\langle \lambda x.e, \rho \rangle, a) = C\ (\ldots, \mathcal{S}(\rho)(y_i), \ldots)$ where $C = \mathcal{C}(a, \lambda x.e)$ and $\ldots, y_i, \ldots = \mathsf{FV}(\lambda x.e)$

The translation can ignore the context (represented as an abstract value) in which a constructor application occurs, because all such applications are merged by the flow analysis. A source-level closure is translated to a constructor application where the tag is the constructor name, and the value is a tuple containing the translated values of the free variables in the corresponding procedure.

Using $\mathcal{S}$, we can state the correctness of the translation as

Theorem 5.3 (Translation Correctness) *If* $e, \rho \hookrightarrow v$ *then* $[\![e]\!], \mathcal{S}(\rho) \hookrightarrow \mathcal{S}(v, F(\mathit{last}(e)))$

The proof of this theorem is by induction on $\hookrightarrow$.

6 Example

Consider the example in Figure 7 where we take the flow to be be the most precise one consistent with our safety conditions. This is exactly the flow which a standard closure analysis such as 0-CFA [Shi91] would compute.

Consider first the translation of the last expression, the call to `m`. Since `m` may be bound to a procedure corresponding to λ_2 or λ_4, 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 $\{\lambda_1, \lambda_3\}$.

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 λ_2 or λ_4.

7 Experiments

We give a brief overview of the relevant compiler passes and intermediate languages (ILs) in MLton. First, it translates the input SML program into an explicitly-typed, polymorphic intermediate

```
let f = λ1a. λ2b. a
    g = λ3c. λ4d. d
    h = case ... of
             ... ⇒ f
             ... ⇒ g
    m = h 13
    _ = m 7
```

(a) Source program

$$
\begin{array}{rcl}
F(\mathsf{f}) & = & \{\lambda_1\} \\
F(\mathsf{g}) & = & \{\lambda_3\} \\
F(\mathsf{h}) & = & \{\lambda_1, \lambda_3\} \\
F(\mathsf{m}) & = & \{\lambda_2, \lambda_4\}
\end{array}
$$

(b) Flow

```
datatype t1 = C1 of unit    (* λ1 *)
datatype t2 = C2 of int     (* λ2 *)
datatype t3 = C3 of unit    (* λ3 *)
datatype t4 = C4 of unit    (* λ4 *)
datatype t5 = C5 of unit    (* λ1 ∈ {λ1, λ3} *)
              C6 of unit    (* λ3 ∈ {λ1, λ3} *)
datatype t6 = C7 of int     (* λ2 ∈ {λ2, λ4} *)
              C8 of unit    (* λ4 ∈ {λ2, λ4} *)
fun F ((), a) = C2 a        (* code for λ1 *)
fun F' ((a), b) = a         (* code for λ2 *)
fun G ((), c) = C4 ()       (* code for λ3 *)
fun G' ((), d) = d          (* code for λ4 *)
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

Figure 7: Example

language (XML)[HM93]. XML does not have any module level constructs. All functor applications are performed at compile-time, 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 4 on the resulting SXML, and closure converts procedures to FOL (a first-order simply-typed language) via the algorithm described in Section 5. 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 [TAL92], a trampoline is used to satisfy proper 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 [FMRW97].

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, `life`, `count-graphs`, `lexgen`, `knuth-bendix` and `mlyacc` do mostly symbolic processing. `Ratio-regions` is an integer-intensive benchmark, while `simple` and `tsp` make heavy use of floating-point arithmetic. `MLton` is the compiler itself, and `kit` is the ML-kit [TBE+97, TB98]. The benchmarks were executed on a 450 MHz Intel Xeon with 1 GB of memory.

Program	lines SML	SML/NJ	MLton Total	to C	gcc
life	161	.9	3.2	1.9	1.3
count-graphs	204	1.2	4.02	2.5	1.52
tsp	495	.8	3.56	2.5	1.06
knuth-bendix	606	2.7	6.55	3.5	3.05
ratio-regions	627	2.2	6.22	4.1	2.12
simple	935	4.7	34.11	15.6	18.51
lexgen	1329	4.5	19.52	9.2	10.32
mlyacc	7297	30.1	144.86	90.1	54.76
MLton	47768	637.5	1672.0	321.9	1350.1
kit	73489	1375.75	2456.39	434.15	2022.24

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

Program	Flow	Convert
life	.01	.16
count-graphs	.01	.25
tsp	.01	.22
knuth-bendix	.01	.32
ratio-regions	.01	.35
simple	.04	.87
lexgen	.03	.78
mlyacc	.10	2.34
MLton	1.94	33.84
kit	1.34	27.96

Table 2: Flow analysis and closure conversion times (seconds).

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)[2] and MLton. The number of lines does not include approximately 8000 lines of basis library code that MLton prefixes to each program. The MLton compile times are broken down into the time spent to produce the C program, and the time taken by `gcc` to compile the C program. The compile time for SML/NJ is the time to compile the entire program wrapped in a `local` declaration whose body performs an `exportFn`.

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 also linear. A major component of MLton's compile-time, especially on large programs, is due to `gcc`. We have also observed pathological behavior from `gcc` on some small programs. For example, `logic`, a 400 line program takes only 6.4 seconds for MLton to produce C, but 203 seconds for `gcc` to produce the executable. We expect a native back-end to replace the `gcc` time with a small percentage of MLton's compilation-to-C time.

In Table 2, we provide the time taken by flow analysis and closure conversion. The flow analysis

[2]Except for the `kit` which is run under SML/NJ version 110.0.3 because 110.9.1 does not except Standard ML.

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

Table 3: Dynamic counts (millions/second).

times are shorter than previous work would suggest [Ash96, HM97, DGC98], 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 a polyvariant analysis on the source program[JWW97]. 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*[JM81] approach (see rules 4 and 5a in Figure 4).

Table 3 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. Entries less than .01 are written as ˜0. SXML-Known and SXML-Unknown measure the number of known and unknown procedure calls identified in the SXML program using only syntactic heuristics [App92]. 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-Known and FOL-Known is due to inlining. Dispatch indicates the number of dispatches 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.

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 4 gives runtime results for both SML/NJ and MLton.

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

Table 4: Runtimes (in seconds) and ratio of SML/NJ to MLton.

8 Conclusions

Our work shows 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.

References

[App92] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[Ash96] 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.

[BBH97] 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.

[DGC98] Greg Defouw, David Grove, and Craig Chambers. Fast interprocedural class analysis. In *ACM Symposium on Principles of Programming Languages*, pages 222–236, January 1998.

[DMTW97] Allyn Dimock, Robert Muller, Franklyn Turback, and J.B. Wells. Strongly-typed flow-directed representation transformations. In *International Conference on Functional Programming*, June 1997.

[FMRW97] 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.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *ACM Conference on Programming Language Design and Implementation*, 1993.

[Hei94] Nevin Heintze. Set-based analysis of ML programs. In *ACM Conference on LISP and Functional Programming*, pages 306–317, 1994.

[Hen92] Fritz Henglein. Simple closure analysis. Technical Report D-193, Department of Computer Science, University of Copenhagen, March 1992.

[HM93] 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.

[HM97] 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.

[JLST98] 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.

[JM81] Neil D. Jones and Stephen S. Muchnick. *Flow Analysis and Optimization of LISP-like Structures*, chapter 4, pages 102–131. Prentice–Hall, 1981.

[JWW97] Suresh Jagannathan, Stephen T. Weeks, and Andrew K. Wright. Type-directed flow analysis for typed intermediate languages. In *International Static Analysis Symposium*, September 1997.

[MMH96] 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.

[Pal95] Jens Palsberg. Closure analysis in constraint form. *ACM Transactions on Programming Languages and Systems*, 17(1):47–62, January 1995.

[PO95] 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.

[Rey72] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM Annual Conference*, pages 717–740, 1972.

[SA94] 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.

[Sha97] Zhong Shao. An overview of the FLINT/ML compiler. In *Proceedings of the 1997 ACM SIGPLAN Workshop on Types in Compilation*, June 1997.

[Shi91] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1991.

[TAL92] 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.

[TB98] Mads Tofte and Lars Birkedal. A region inference algorithm. *Transactions on Programming Languages and Systems*, 20(4):724–767, 1998.

[TBE+97] 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.

[TMC+96] 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.

[TO98] Andrew Tolmach and Dino Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 1998. To appear.