Compiling ML Polymorphism with Explicit Layout Bitmap *

Huu-Duc Nguyen † Atsushi Ohori
Research Institute of Electrical Communication
Tohoku University
{ducnh, ohori}@riec.tohoku.ac.jp

Abstract

Most of the current implementations of functional languages adopt so-called "tagged data representations" to support tracing garbage collection. The representations impose a burden of data conversion on the runtime performance and the interoperability between ML and other languages. In this paper, we present a type-directed compilation method for ML polymorphism that supports natural representations of integers and other atomic data.

This is achieved by compiling ML so that each runtime object (a heap block or a stack frame) has a "bitmap" that describes the pointer positions in the block. Since a polymorphic function may produce runtime objects of different types, the compiler needs to compute appropriate bitmaps for each instantiation of the function. This would require us to insert extra lambda abstractions and applications to pass the bits required in bitmap calculations. This compilation process should be done for both stack frames and heap-allocated objects including functions' closures and their environment records. We solve these problems by combining the type-directed compilation method with typed closure conversion, and type-preserving A-normalization.

The resulting compilation process is shown to be sound with respect to an untyped operational semantics with bitmap-inspecting garbage collection. The proposed compilation method has been implemented for the full Standard ML Language, demonstrating its practical feasibility.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Language Constructs and Features

General Terms Language, Performance, Theory

Keywords Type-directed Compilation, Polymorphism, Memory Management, Garbage Collection

* This research was partially supported by the Japan MEXT (Ministry of Education, Culture, Sports, Science and Technologies) leading project of "Comprehensive Development of Foundation Software for E-Society" under the title "dependable software development technology based on static program analysis."
This is the author's version of the paper to appear in ACM PPDP 2006.

† Part of the work has been done during the first author was studying at Japan Advanced Institute of Science and Technology.

PPDP'06 July 10–12, 2006, Venice, Italy.

1. Introduction

Most of the current implementations of functional languages adopt so-called "tagged data representations," where each word contains one bit tag indicating whether it is a pointer or not. This yields particularly simple memory management: a compiler only needs to set one bit when emitting code, and a tracing garbage collector can locate all the pointers in a heap block by simply scanning the tag bit in each word in the block.

Drawbacks of this approach are the runtime overhead arising from tagging and untagging operations (e.g. for arithmetic operations), and the lack of interoperability with other languages. Since integers and other atomic data do not have their natural runtime representation, they must be converted each time when they are passed to primitive operations or other languages.

One drastic approach to avoid this complication is simply to eliminate polymorphism at compile time through a whole-program analysis and transformation. MLton [1], MLJ [3] and SML.NET [4] have taken this approach. Although the results of these systems have demonstrated the feasibility of this approach for the development of a stand alone system, this causes code duplication of polymorphic functions, and it is not compatible with incremental and interactive programming – a common scenario in functional programming.

In order to avoid the problem of tagged data representation while enjoying the full benefits of polymorphic functional languages, tag-free garbage collection has been investigated [10, 27]. In this approach, the compiler generates type information for each runtime object, and the garbage collector traces the heap space using the type information retrieved from the stack frame of the current function. This solves the problem of incompatibility of integers and other atomic data. One drawback of this approach is the runtime overhead arising from manipulating type information. Another drawback, which we regard as a serious obstacle in achieving a high degree of interoperability, is the close dependency between the garbage collector and the compiler. The garbage collector needs to know the semantics of types and their precise runtime representations. This also implies that the traversal strategy is constrained by type structures. For example, Tolmach's algorithm relies on depth-first traversal and is not necessarily compatible with commonly used efficient strategies, including Cheney's popular collector [5].

The goal of the present work is to develop a compilation method for ML polymorphism so that runtime objects in a stack or a heap can be directly passed to primitive operations and other languages including C and Java. To achieve this goal, we consider the following two features to be prerequisites.

- Integers and other atomic data have their natural representations.
- Each heap block includes its layout information for garbage collection.

One natural strategy that satisfies these criteria is to include in each heap-allocated block a *bitmap* that describes the pointer positions in the block. For example, a nested record $(1,(2,3))$ is implemented as $([0,1];1,([0,0];2,3))$ where $[0,1]$ is a bitmap indicating the two word block of an atom and a pointer to a block, which in this case is another record $([0,0],2,3)$. Since this representation only includes the necessary information for garbage collection and does not place any constraint on scanning strategy of garbage collection, it should be efficient and compatible with heap management of typed languages featuring monomorphic allocation, such as Java.

For a polymorphic language such as ML, generating the correct bitmap for each allocation is challenging. To see the problem, let us consider the following program in Standard ML syntax,

```
let fun f x = let fun g y = (x,y)
              in (g 1; g "1")
              end
in (f 1 ; f "1") end
```

where (e_1,e_2) is a record to be allocated in a heap, and $(e_1;e_2)$ is sequential evaluation. Although there is only one heap allocation code, namely `(x,y)`, it will allocate records of 4 different bitmaps `[0,0]`, `[0,1]`, `[1,0]`, and `[1,1]`. In order to set a correct bitmap for each instance, it is necessary to evaluate the instance types of `x` and `y` to get the corresponding portions of the entire bitmap, compose them, and pass them to the code of `(x,y)`.

Our strategy is to develop a type-directed compilation algorithm that statically computes the necessary bit tags from the type information. Before presenting in details the technical development, we outline the compilation method and compare it with related works.

Method outline. The general idea behind computing bitmap information of heap-allocated objects inside a polymorphic function is to follow polymorphic record compilation [20] and to encode the necessary information in polymorphic types.

For example, consider the following explicitly typed polymorphic function.

```
let f:∀t.t → t × t = λx:t.(x,x)
in (f int 1; f string "1") end
```

Since the type of record expression `(x,x)` is $t\times t$, we know that this expression requires a bitmap corresponding to $t \times t$. This bitmap can be composed from bit tag information of objects of type t. We extend the function `f` by introducing new parameter `a` and type this new variable with a special *bit tag type* of the forms $\langle t\rangle$. In general, $\langle\sigma\rangle$ denotes (the singleton set of) 0 if σ is a type of *unboxed* objects (i.e. non-pointer atoms), and 1 if σ is a type of *boxed* objects (i.e. pointers). In ML Core language, if σ is not a type variable then $\langle\sigma\rangle$ is determined by the outermost type constructor of σ.

Using the bit tag type, we can obtain the necessary bit tag for each usage of `f` by inspecting the instance of the bit tag types. For example, if t is instantiated to int then the corresponding bit tag type is $\langle t\rangle[int/t] = \langle int\rangle$ which denotes the (singleton) value 0. The value of bitmap `(x,x)` is therefore $[0,0]$. Based on this idea, the above program is compiled to the following term.

```
let f:∀t.⟨t⟩ → t → t × t = λa:⟨t⟩.λx: t.([a,a]; x, x)
in (f int 0 1; f string 1 "1") end
```

The syntax $[e_1,\ldots,e_n]$ is the term constructor for bitmaps.

Related work. Most relevant to this paper is the work on tag-free garbage collection, where the garbage collector traces data using type information generated by the compiler. Goldberg [10] has proposed a tracing method by inspecting the call frame of the current function to determine the type of each runtime object. If the current function is a polymorphic function, then the type information is computed from the types of its arguments. Tolmach [27] has proposed a refined approach to pass types of arguments to each polymorphic function. This approach requires runtime construction and inspection of type information, which may be large and inefficient. In [17, 25], methods for optimizing type-passing compilation have been proposed. Compared with these approaches, our method can be regarded as a type-directed refinement of those type-passing approaches by statically computing most of the layout information at compile time. This reduces the runtime time and space overhead of type-passing. Another important consequence of the refinement is that our method produces self-contained layout information for each runtime object so that the garbage collector may be implemented independently of the language type system.

Our method yields an accurate collector based on reachability. There have also been several attempts [9, 18, 13] to develop more powerful GC exploiting type information at runtime. It is an interesting open issue whether we can combine these approaches and our type-based compilation approach.

Our method uses type information at runtime. This technique is related to previous work on type-directed compilation, including intensional type analysis [12], compilation of type classes [11, 24], and mixed representation optimization [15]. In comparison with these approaches, one type theoretical feature our method relies on is the introduction of a family of bit tag types, each of which denotes a singleton set of bit tags. This idea is first presented in [20]. Crary, Weirich, and Morriset [6] have proposed a similar mechanism.

As we shall briefly report in Section 5, we have implemented the mechanism presented here in our SML# [2] compiler, which is an extension of Standard ML.

The SML# compiler is composed of a series of type-directed type-preserving transformation steps. This approach has the same spirit of the TIL compiler [26] and the compilation of System F to a typed assembly language [19].

Paper organization. The rest of this paper is organized as follows. Section 2 develops the compilation method outlined in Section 1. Section 3 gives the solution to deal with a mutual dependency problem between the compilation method and closure conversion. Section 4 briefly introduces the method of generating layout bitmaps for stack frames. Optimizations and implementation are presented in Section 5. Section 6 concludes the paper.

2. The Bitmap-passing Compilation

In this section, we formally present the type-directed compilation method for generating codes consisting of bitmap computations from source expressions in an explicitly typed ML-style calculus – λ^{ML}. The compilation process is described in two stages. The first stage, named *bitmap-passing compilation*, transforms a source expression in λ^{ML} into a target expression in an explicitly typed target calculus – λ^B. Bitmaps and bit tags are explicitly encoded as terms in λ^B. The second stage erases all type annotations in the explicitly typed resulting term in λ^B to achieve a term in an implicitly typed target calculus – Λ^B.

As a formal presentation, we define an untyped operational semantics for the target calculus Λ^B, and show that the type system of Λ^B is sound with respect to the operational semantics. We also show the syntactic correctness of the proposed compilation algorithm. The combination of these two results guarantees that the type system of the source calculus is sound with respect to the operational semantics realized by our compilation algorithm followed by evaluation of the compiled term.

2.1 The Source Calculus – λ^{ML}

The source calculus is an explicitly typed ML-style calculus with rank-1 polymorphism [23]. Supporting rank-1 types is important

$$\vdash_{ML} \emptyset$$

$$\frac{\vdash_{ML} \Gamma \quad FTV(\overline{\tau}) \subseteq TV(\Gamma) \quad \overline{x} \cap dom(\Gamma) = \emptyset}{\vdash_{ML} \Gamma, arg(\overline{x : \tau})}$$

$$\frac{\vdash_{ML} \Gamma \quad TV(\Gamma) \cap \bar{t} = \emptyset}{\vdash_{ML} \Gamma, tvar(\bar{t})}$$

$$\frac{\vdash_{ML} \Gamma \quad FTV(\sigma) \subseteq TV(\Gamma) \quad x \notin dom(\Gamma)}{\vdash_{ML} \Gamma, local(x : \sigma)}$$

Figure 1. Context formation rules

in obtaining efficient target code by suppressing unnecessary type abstractions and type applications.

We write $\overline{a}$ for a sequence of objects denoted by the meta-variable a. The sets of terms (ranged over by e), monomorphic types (ranged over by τ), polymorphic types (ranged over by σ), and contexts (ranged over by Γ) are given by the following syntax.

$$\begin{array}{rcl} e & ::= & c^o \mid x \mid \lambda\overline{x : \tau}.e \mid (e\ \overline{e}) \mid \Lambda\bar{t}.e \mid (e\ \overline{\tau}) \mid \\ & & (e, \ldots, e) \mid \pi_i(e) \mid \texttt{let}\ x : \sigma = e\ \texttt{in}\ e\ \texttt{end} \\ \tau & ::= & o \mid t \mid \overline{\tau} \to \tau \mid \tau \times \cdots \times \tau \\ \sigma & ::= & \tau \mid \forall\bar{t}.\sigma \mid \overline{\tau} \to \sigma \mid \sigma \times \cdots \times \sigma \\ \Gamma & ::= & \emptyset \mid \Gamma, arg(\overline{x : \tau}) \mid \Gamma, local(x : \sigma) \mid \Gamma, tvar(\bar{t}) \end{array}$$

Lambda abstraction and lambda application are generalized with multiple arguments. Since bit tag abstractions and bit tag applications usually occur together with ordinary abstractions and applications, this generalization is useful for generating optimized code by uncurrying bit tag abstractions. We shall comment on this issue in Section 5. A context Γ is a sequence of assumptions of the forms $arg(\overline{x : \tau})$, $local(x : \sigma)$, and $tvar(\bar{t})$ for lambda variables, `let` variables, and type variables. To introduce bit tag abstraction, it is necessary to record type variables explicitly in the context.

Let $FTV(\sigma)$, $FTV(\overline{\tau})$ be the sets of free type variables of σ and $\overline{\tau}$, $TV(\Gamma)$ be the set of type variables declared in Γ. A type σ is *well formed* under a well formed context Γ, written $\Gamma \vdash \sigma$, if $FTV(\sigma) \subseteq TV(\Gamma)$. We write $\vdash \Gamma$ if a context Γ is well formed, and write $\Gamma \vdash_{ML} e : \sigma$ if e has type σ under a well formed context Γ. The set of rules to derive these judgments are given in Figures 1 and Figures 2.

In the typing rule for variable, we check $x : \sigma \in \Gamma$ by looking up the appearance of $x : \sigma$ in the $arg()$ and $local()$ assumptions of Γ. We also define $\Gamma(x) = \sigma$ for $x : \sigma \in \Gamma$ and $dom(\Gamma) = \{x | x : \sigma \in \Gamma\}$.

Figure 3 shows an example of untyped terms and the corresponding explicitly typed terms in the source calculus which depicts the relevance to rank-1 polymorphism. `f` is given a nested polymorphic type, and `g` is given a record type whose components are polymorphic types. This mechanism not only allows partial applications `f 1` and `#1 g` to have polymorphic types, but it also suppress redundant type instantiation and type abstraction. This feature is especially important for generating efficient code in our type-directed compilation where type instantiations may generate extra lambda abstractions.

2.2 The Explicitly Typed Bitmap-passing Calculus – λ^B

This sub-section defines the *explicitly typed bitmap-passing calculus*, written λ^B, as an immediate language for the compilation process.

$$\Gamma \vdash_{ML} c^o : o$$

$$\Gamma \vdash_{ML} x : \sigma \quad \text{if } x : \sigma \in \Gamma$$

$$\frac{\Gamma, arg(\overline{x : \tau}) \vdash_{ML} e : \sigma}{\Gamma \vdash_{ML} \lambda\overline{x : \tau}.e : \overline{\tau} \to \sigma}$$

$$\frac{\Gamma, tvar(\bar{t}) \vdash_{ML} e : \sigma}{\Gamma \vdash_{ML} \Lambda\bar{t}.e : \forall\bar{t}.\sigma}$$

$$\frac{\Gamma \vdash_{ML} e_1 : \overline{\tau} \to \sigma \quad \Gamma \vdash_{ML} \overline{e_2} : \overline{\tau}}{\Gamma \vdash_{ML} (e_1\ \overline{e_2}) : \sigma}$$

$$\frac{\Gamma \vdash_{ML} e : \forall\bar{t}.\sigma \quad \Gamma \vdash_{ML} \overline{\tau}}{\Gamma \vdash_{ML} (e_1\ \overline{\tau}) : \sigma[\overline{\tau}/\bar{t}]}$$

$$\frac{\Gamma \vdash_{ML} e_i : \sigma_i \quad \text{for all } 1 \le i \le n}{\Gamma \vdash_{ML} (e_1, \ldots, e_n) : \sigma_1 \times \cdots \times \sigma_n}$$

$$\frac{\Gamma \vdash_{ML} e : \sigma_1 \times \cdots \times \sigma_n}{\Gamma \vdash_{ML} \pi_i(e) : \sigma_i}$$

$$\frac{\Gamma \vdash_{ML} e_1 : \sigma_1 \quad \Gamma, local(x : \sigma_1) \vdash_{ML} e_2 : \sigma_2}{\Gamma \vdash_{ML} \texttt{let}\ x : \sigma_1 = e_1\ \texttt{in}\ e_2\ \texttt{end} : \sigma_2}$$

Figure 2. Type system of λ^{ML}

The set of terms of this calculus is given by the following syntax:

$$\begin{array}{rcl} e & ::= & c^o \mid x \mid \lambda\overline{x : \tau}.e \mid (e\ \overline{e}) \mid \Lambda\bar{t}.\lambda\overline{x : \langle t \rangle}.e \mid (e\ \overline{\tau}\ \overline{e}) \\ & & \mid (e; e, \ldots, e) \mid \pi_i(e) \mid \texttt{let}\ x : \sigma = e\ \texttt{in}\ e\ \texttt{end} \\ & & \mid [e, \ldots, e] \mid B \\ B & ::= & 0 \mid 1 \end{array}$$

$\Lambda\bar{t}.\lambda\overline{x : \langle t \rangle}.e$ is a bit tag function where $\overline{x}$ are bit tag parameters inserted by the bitmap-passing compilation (introduced later). $\overline{\langle t \rangle}$ represent *bit tag types* corresponding to the type variables $\bar{t}$. A bit tag function always comes together with a type abstraction, they are tightened up together by using a single term constructor in the syntax of the language. $(e\ \overline{\tau}\ \overline{e_b})$ is a bit tag application where $\overline{e_b}$ are actual bit tags generated from the types $\overline{\tau}$. We also use a single term constructor for both bit tag application and type instantiation. B is a static (constant) bit tag and $[e_1, \ldots, e_n]$ represents a bitmap composition from n bit tags. The first component e_0 in a record expression $(e_0; e_1, \ldots, e_n)$ is a bitmap term representing the bitmap information of this record.

To define typing rules for terms involving bitmaps, the set of types is extended with types for bitmaps and bit tags as:

$$\begin{array}{rcl} \tau & ::= & o \mid t \mid \beta \mid \gamma \mid \overline{\tau} \to \tau \mid \tau \times \cdots \times \tau \\ \sigma & ::= & \tau \mid \forall\bar{t}.\sigma \mid \overline{\tau} \to \sigma \mid \sigma \times \cdots \times \sigma \\ \beta & ::= & \langle \sigma \rangle \\ \gamma & ::= & \langle\!\langle \sigma, \ldots, \sigma \rangle\!\rangle \end{array}$$

where β ranges over bit tag types, and γ ranges over bitmap types.

In ML Core language, for any σ type other than type variable, whether σ is a boxed type or an unboxed type is determined by its outermost type constructor. We define the tag value $\texttt{tagOf}(\sigma)$ of a type σ as follows.

Untyped ML expressions | The corresponding explicitly typed source terms

```
let fun f x y = (x,y)                 ⟹   let f:∀t1.t1 → (∀t2.t2 → t1 × t2) =
in let val g = (f 1, 1);                       Λt1.λx: t1 .Λt2.λy:t2 .(x,y) in
   in #1 g 1 end                              let g:(∀t1.t1 → int × t1) × int = (f int 1, 1) in
end                                           π1(g) int 1 end end
```

Figure 3. Examples of Explicitly Typed Source Terms

$$\frac{\Gamma, tvar(\overline{t}), arg(\overline{b : \langle t \rangle}) \vdash_{TB} e : \sigma}{\Gamma \vdash_{TB} \Lambda\overline{t}.\lambda\overline{b : \langle t \rangle}.e : \forall\overline{t}.\overline{\langle t \rangle} \to \sigma}$$

$$\frac{\Gamma \vdash_{TB} e : \forall\overline{t}.\overline{\langle t \rangle} \to \sigma \quad \Gamma \vdash_{TB} \overline{e_b} : \overline{\langle \tau \rangle} \quad \Gamma \vdash_{TB} \overline{\tau}}{\Gamma \vdash_{TB} (e\ \overline{\tau}\ \overline{e_b}) : \sigma[\overline{\tau}/\overline{t}]}$$

$$\frac{\Gamma \vdash_{TB} e_i : \sigma_i\ (1 \le i \le n) \quad \Gamma \vdash_{TB} e_0 : \langle\!\langle \sigma_1, \ldots, \sigma_n \rangle\!\rangle}{\Gamma \vdash_{TB} (e_0; e_1, \ldots, e_n) : \sigma_1 \times \cdots \times \sigma_n}$$

$$\frac{\mathtt{tagOf}(\sigma) = B}{\Gamma \vdash_{TB} B : \langle \sigma \rangle}$$

$$\frac{\Gamma \vdash_{TB} e_i : \langle \sigma_i \rangle\ (1 \le i \le n)}{\Gamma \vdash_{TB} [e_1, \ldots, e_n] : \langle\!\langle \sigma_1, \ldots, \sigma_n \rangle\!\rangle}$$

Figure 4. Typing rules of λ^B

$$\begin{array}{rcl}
\mathtt{tagOf}(o) & = & \text{0 if } o \text{ is an unboxed base type} \\
\mathtt{tagOf}(o) & = & \text{1 if } o \text{ is an boxed base type} \\
\mathtt{tagOf}(\overline{\tau} \to \sigma) & = & 1 \\
\mathtt{tagOf}(\forall\overline{t}.\sigma) & = & 1 \\
\mathtt{tagOf}(\sigma_1 \times \cdots \times \sigma_n) & = & 1 \\
\mathtt{tagOf}(\langle \sigma \rangle) & = & 0 \\
\mathtt{tagOf}(\langle\!\langle \sigma_1, \ldots, \sigma_n \rangle\!\rangle) & = & 0 \\
\mathtt{tagOf}(t) & = & \text{undetermined}
\end{array}$$

Functions and records have bit tag 1 since they are heap-allocated objects. The case for $\forall\overline{t}.\sigma$ requires some explanation: this type always refers to a bit tag function type, so we can safely assume $\forall\overline{t}.\sigma$ to be a boxed type.

The well-formedness of types ($\Gamma \vdash_{TB} \sigma$) is similarly defined as one in the source language with the following extension of the function FTV to bit tag and bitmap types:

$$\begin{array}{rcl}
FTV(\langle \sigma \rangle) & = & FTV(\sigma) \\
FTV(\langle\!\langle \sigma_1, \ldots, \sigma_n \rangle\!\rangle) & = & \bigcup FTV(\sigma_i)
\end{array}$$

We write $\Gamma \vdash_{TB} e : \sigma$ if a target expression e has a target type σ under a well formed context Γ. The only differences between the source typing rules and the target typing rules are the cases for bit tag abstractions, bit tag applications, records, constant bit tags, and bitmaps. Figure 4 shows these typing rules.

2.3 Bitmap-passing Compilation

As we briefly introduced, the first stage of the compilation process is to transform a source expression in λ^{ML} into an explicitly typed expression in λ^B. The major differences between a target term and its source expression are bitmaps at records, bit tag abstractions at type abstractions and bit tag applications at type instantiations. The goal of bitmap-passing compilation is therefore to insert these objects into appropriate places for a given source expression.

In order to achieve this goal, we formulate the compilation algorithm by compilation judgments of the forms $\Gamma \vdash_{TB} e \rightsquigarrow e'$ where Γ is the context of the target calculus, e is a source expression and e' is a target expression. The set of compilation rules is given in Figure 5.

$$\Gamma \vdash_{TB} c^o \rightsquigarrow c^o$$

$$\Gamma \vdash_{TB} x \rightsquigarrow x \quad \text{If } x : \sigma \in \Gamma$$

$$\frac{\Gamma, arg(\overline{x : \tau}) \vdash_{TB} e \rightsquigarrow e'}{\Gamma \vdash_{TB} \lambda\overline{x : \tau}.e \rightsquigarrow \lambda\overline{x : \tau}.e'}$$

$$\frac{\Gamma \vdash_{TB} e_1 \rightsquigarrow e_1' \quad \Gamma \vdash_{TB} \overline{e_2} \rightsquigarrow \overline{e_2'}}{\Gamma \vdash_{TB} (e_1\ \overline{e_2}) \rightsquigarrow (e_1'\ \overline{e_2'})}$$

$$\frac{\Gamma, tvar(\overline{t}), arg(\overline{b : \langle t \rangle}) \vdash_{TB} e \rightsquigarrow e' \quad \overline{b} \text{ are fresh variables}}{\Gamma \vdash_{TB} \Lambda\overline{t}.e \rightsquigarrow \Lambda\overline{t}.\lambda\overline{b : \langle t \rangle}.e'}$$

$$\frac{\Gamma \vdash_{TB} e \rightsquigarrow e' \quad \Gamma \vdash_{TB} \overline{\tau} \rightsquigarrow \overline{e_b}}{\Gamma \vdash_{TB} (e\ \overline{\tau}) \rightsquigarrow (e'\ \overline{\tau}\ \overline{e_b})}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{TB} e_i \rightsquigarrow e_i' \quad \Gamma \vdash_{TB} e_i' : \sigma_i \\ \Gamma \vdash_{TB} \sigma_i \rightsquigarrow e_i^b \quad (1 \le i \le n)\end{array}}{\Gamma \vdash_{TB} (e_1, \ldots, e_n) \rightsquigarrow ([e_1^b, \ldots, e_n^b]; e_1', \ldots, e_n')}$$

$$\frac{\Gamma \vdash_{TB} e \rightsquigarrow e'}{\Gamma \vdash_{TB} \pi_i(e) \rightsquigarrow \pi_i(e')}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{TB} e_1 \rightsquigarrow e_1' \quad \Gamma \vdash_{TB} e_1 : \sigma' \\ \Gamma, local(x : \sigma') \vdash_{TB} e_2 \rightsquigarrow e_2'\end{array}}{\begin{array}{l}\Gamma \vdash_{TB} \mathtt{let}\ x : \sigma = e_1\ \mathtt{in}\ e_2\ \mathtt{end} \rightsquigarrow \\ \qquad \mathtt{let}\ x : \sigma' = e_1'\ \mathtt{in}\ e_2'\ \mathtt{end}\end{array}}$$

Figure 5. Bitmap-passing Compilation

A record $(e_1, \ldots, e_n)$ in the source calculus is compiled to $(e_0; e_1', \ldots, e_n')$ where e_i', for all $1 \le i \le n$, is the compiled term of e_i , and e_0 is the bitmap composition from n bit tags created from types of $e_1', \ldots, e_n'$. The bit tag creation algorithm, which are used in rules 6 and 7 in Figure 5, is defined by a set of rules to derive a judgment of the form $\Gamma \vdash_{TB} \sigma \rightsquigarrow e$, where e is the bit tag term created from the type σ under the context Γ. The following are two rules to derive this judgment.

$\Gamma \vdash_{TB} \sigma \rightsquigarrow \mathtt{tagOf}(\sigma)$	if σ has a proper outermost type constructor
$\Gamma \vdash_{TB} t \rightsquigarrow b$	where $b : \langle t \rangle \in \Gamma$

The compilation algorithm introduces new bit tag parameters $\overline{b}$ of types $\overline{\langle t \rangle}$ for each type abstraction $\Lambda\overline{t}.e$ in the source language to achieve the resulting term $\Lambda\overline{t}.\lambda\overline{b : \langle t \rangle}.e'$ in λ^B (e' is the compiled term of e). The introduction of bit tag parameters requires to insert actual bit tag parameters at type instantiations. A type instantiation term $(e\ \overline{\tau})$ in the source language is compiled to $(e'\ \overline{\tau}\ \overline{e_b})$, where e' is compiled from e and $\overline{e_b}$ is generated from $\overline{\tau}$ by using the bit tag creation rules.

The bitmap compilation algorithm only inserts bitmap abstractions and bitmap applications in target types and terms. Erasing

$$\tau \sim_B \tau$$

$$\frac{\sigma \sim_B \sigma'}{\overline{\tau} \to \sigma \sim_B \overline{\tau} \to \sigma'}$$

$$\frac{\sigma \sim_B \sigma'}{\forall \bar{t}.\sigma \sim_B \forall \bar{t}.\overline{\langle t \rangle} \to \sigma'}$$

$$\frac{\sigma_i \sim_B \sigma'_i \text{ for each } 1 \leq i \leq n}{\sigma_1 \times \cdots \times \sigma_n \sim_B \sigma'_1 \times \cdots \times \sigma'_n}$$

$$\emptyset \sim_B \emptyset$$

$$\frac{\Gamma \sim_B \Gamma' \quad x \notin dom(\Gamma') \quad \sigma \sim_B \sigma'}{(\Gamma, local(x:\sigma)) \sim_B (\Gamma', local(x:\sigma'))}$$

$$\frac{\Gamma \sim_B \Gamma' \quad \overline{x} \cap dom(\Gamma') = \emptyset}{(\Gamma, arg(\overline{x:\tau})) \sim_B (\Gamma', arg(\overline{x:\tau}))}$$

$$\frac{\Gamma \sim_B \Gamma' \quad \bar{b} \cap dom(\Gamma') = \emptyset}{(\Gamma, tvar(\bar{t})) \sim_B (\Gamma', tvar(\bar{t}), arg(\overline{\bar{b} : \langle t \rangle}))}$$

Figure 6. Simulation Relations on Types and Contexts

all bitmap abstractions from the type of a target expression should therefore recover the original type of the source expression. To formalize this property, we define simulation relations $\sigma \sim_B \sigma'$ between source and target types, and $\Gamma \sim_B \Gamma'$ between source and target contexts in Figure 6. The case of $tvar$ assumption shows that if $\Gamma \sim_B \Gamma'$ then there is always an assumption $arg(\overline{\bar{b} : \langle t \rangle})$ followed by a $tvar(\bar{t})$ in Γ'. This guarantees that the bit tag creation algorithm never fails for well-typed expression in the ML Core language.

The compilation algorithm preserves the typing relation as shown in the following theorem:

THEOREM 1. *Suppose* $\Gamma \vdash_{ML} e : \sigma$. *For any* Γ' *so that* $\Gamma \sim_B \Gamma'$ *the compilation algorithm succeeds as* $\Gamma' \vdash_{TB} e \rightsquigarrow e'$ *with* $\Gamma' \vdash_{TB} e' : \sigma'$ *where* $\sigma \sim_B \sigma'$

2.4 The Implicitly Typed Bitmap-passing Calculus – Λ^B

The second stage of the compilation process is the phase which eliminates all type annotations in the resulting term from the first stage to achieve a target term in an implicitly typed calculus – Λ^B. We define the set of terms of Λ^B by the following syntax:

$$\begin{aligned} e \quad ::= \quad & c^o \mid x \mid \lambda \overline{x}.e \mid (e\ \overline{e}) \mid (e; e, \ldots, e) \mid \pi_i(e) \\ & \mid \texttt{let}\ x = e\ \texttt{in}\ e\ \texttt{end} \mid [e, \ldots, e] \mid B \end{aligned}$$

Since terms in Λ^B are achieved from terms in λ^B by eliminating all type annotations, ordinary lambda abstraction and bit tag abstraction can share the same term constructor (i.e. $\lambda \overline{x}.e$), and ordinary lambda application and bit tag application can share the same term constructor (i.e. $(e\ \overline{e})$). Thus, in the typing rules for terms, each of abstraction and application may have two different typing derivations as shown in Figure 7.

We define the *type erasure* $\mathcal{E}(e)$ of a term e in λ^B by specifying the case analysis on the structure of e. The following are two typical cases for bit tag abstractions and bit tag applications.

$$\begin{aligned} \mathcal{E}(\Lambda \bar{t}.\lambda \overline{x : \langle t \rangle}.e) &= \lambda \overline{x}.\mathcal{E}(e) \\ \mathcal{E}((e\ \overline{\tau}\ \overline{e_b})) &= (\mathcal{E}(e)\ \overline{\mathcal{E}(e_b)}) \end{aligned}$$

$$\Gamma \vdash_B c^o : o$$

$$\Gamma \vdash_B x : \sigma \quad \text{if } x : \sigma \in \Gamma$$

$$\frac{\Gamma, arg(\overline{x:\tau}) \vdash_B e : \sigma}{\Gamma \vdash_B \lambda \overline{x}.e : \overline{\tau} \to \sigma}$$

$$\frac{\Gamma, tvar(\bar{t}), arg(\overline{\bar{b} : \langle t \rangle}) \vdash_B e : \sigma}{\Gamma \vdash_B \lambda \bar{b}.e : \forall \bar{t}.\overline{\langle t \rangle} \to \sigma}$$

$$\frac{\Gamma \vdash_B e_1 : \overline{\tau} \to \sigma \quad \Gamma \vdash_B \overline{e_2} : \overline{\tau}}{\Gamma \vdash_B (e_1\ \overline{e_2}) : \sigma}$$

$$\frac{\Gamma \vdash_B e : \forall \bar{t}.\overline{\langle t \rangle} \to \sigma \quad \Gamma \vdash_B \overline{e_b} : \overline{\langle \tau \rangle} \quad \Gamma \vdash_B \overline{\tau}}{\Gamma \vdash_B (e\ \overline{e_b}) : \sigma[\overline{\tau}/\bar{t}]}$$

$$\frac{\Gamma \vdash_B e_i : \sigma_i\ (1 \leq i \leq n) \quad \Gamma \vdash_B e_0 : \langle\!\langle \sigma_1, \ldots, \sigma_n \rangle\!\rangle}{\Gamma \vdash_B (e_0; e_1, \ldots, e_n) : \sigma_1 \times \cdots \times \sigma_n}$$

$$\frac{\Gamma \vdash_B e : \sigma_1 \times \cdots \times \sigma_n}{\Gamma \vdash_B \pi_i(e) : \sigma_i}$$

$$\frac{\Gamma \vdash_B e_1 : \sigma_1 \quad \Gamma, local(x : \sigma_1) \vdash_B e_2 : \sigma_2}{\Gamma \vdash_B \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end} : \sigma_2}$$

$$\frac{\texttt{tagOf}(\sigma) = B}{\Gamma \vdash_B B : \langle \sigma \rangle}$$

$$\frac{\Gamma \vdash_B e_i : \langle \sigma_i \rangle\ (1 \leq i \leq n)}{\Gamma \vdash_B [e_1, \ldots, e_n] : \langle\!\langle \sigma_1, \ldots, \sigma_n \rangle\!\rangle}$$

Figure 7. Typing rules of Λ^B

Type erasure preserves type as shown in the following theorem.

THEOREM 2. *Suppose* Γ, e, σ *is a well-formed context, a term and a well-formed type in* λ^B *so that* $\Gamma \vdash_{TB} e : \sigma$. *Then we also have* $\Gamma \vdash_B \mathcal{E}(e) : \sigma$.

A *type substitution*, or simply *substitution*, is a function that maps from a finite set of type variables to monotypes. We write $S = [\tau_1/t_1, \ldots, \tau_n/t_n]$ for the substitution that maps t_i to τ_i, and define $dom(S)$ for the set $\{t_1, \ldots t_n\}$. A substitution S is extended to the set of all type variables by letting $S(t) = t$ for all $t \notin dom(S)$. By this extension, the application of a substitution S to a type σ, written $S(\sigma)$, is achieved by simultaneously substituting all free occurrences of a type variables t in σ by $S(t)$. $S(\Gamma)$ is a context achieved from Γ by applying S to any type assumption in Γ, i.e. replacing $x : \sigma$ by $x : S(\sigma)$.

Since type variables are recorded explicitly in contexts, the well-formedness of context may not be preserved under type substitution. In order to keep this property, we put a particular restriction on type substitutions.

First, we define orders on type variables declared in a context Γ as $t_1 \prec_\Gamma t_2$ if t_1 is recorded before t_2 in Γ. Note that $t_1 \prec_\Gamma t_2$ implies that both t_1 and t_2 are recorded in the context Γ. Second, we define that a substitution S *respects* a context Γ if for all $t_1 \in TV(\Gamma) \cap dom(S)$, and for all $t_2 \in FTV(S(t_1))$, we have $t_2 \prec_\Gamma t_1$. Intuitively, if S respects Γ then for any type variable t in the domain of S and being recorded in Γ, $S(t)$ should be well-formed under Γ and does not contain any free type variable declared after t.

The type system of Λ^B is stable under type substitution as shown in the following lemma.

LEMMA 1. *Let e be an expression, σ be a type, Γ be a well formed context and S be a substitution that respects Γ. If $\Gamma \vdash_B e : \sigma$ then $S(\Gamma) \vdash_B e : S(\sigma)$.*

2.5 Semantics of Λ^B

We define semantics of Λ^B, in style of natural semantic [14], that faithfully models evaluation of expressions under bitmap-inspecting garbage collection. The sets of runtime values (ranged over by v) and runtime environments (ranged over by E) are given by the following grammar.

$$\begin{array}{rcl} v & ::= & c^o \mid i \mid cls(E, \lambda\overline{x}.e) \mid (v; v, \ldots, v) \mid wrong \\ E & ::= & \emptyset \mid E, x : v \end{array}$$

Bit tags and bitmaps are evaluated into unsigned integers (denoted by i). A bit tag value is either 0 or 1. A bitmap value represents a sequence of bit tags. We use the notation $[B_1 \circ \cdots \circ B_n]$ for the unsigned integer obtained by setting the least i^{th} bit to the bit tag value B_i.

Implementation note. A bitmap value may exceed one word representation if the number of bit tag components is greater than the size n of one machine word. In order to make sure that bitmaps always fit into one word data, we assume that the maximum number of fields in a record is always smaller than n or equal n. A long record can be converted into a nested record by applying a source-to-source transformation before being passed to the compilation algorithm. For example, the record $(e_1, \ldots, e_{40})$ in the source language can be transformed into $(e_1, \ldots, e_{31}, (e_{32}, \ldots, e_{40}))$ in a 32-bit architecture. Nested representation of records may introduce extra performance cost. But we believe that this extreme case does not often occur in a real-life application. As we have experienced from 17 benchmark suites of Standard ML of New Jersey, only `mlyacc` requires such kind of transformation.

$cls(E, \lambda\overline{x}.e)$ is a function closure, where E is an environment assigning values to variables. $(v_0; v_1, \ldots, v_n)$ represents a record value whose first component v_0 is the bitmap value of the record block. *wrong* represents a runtime type error.

Instead of explicitly modeling bitmap-inspecting garbage collection, we define the operational semantics in such a way that it checks the correctness of a bitmap every time a record is created or used. If this check fails then the evaluation halts with *wrong*. For performing this check, we define the trace bit $\texttt{tagOf}(v)$ corresponding to v as follows.

$$\begin{array}{rcl} \texttt{tagOf}(c^o) & = & 0 \text{ for unboxed values } c^o \\ \texttt{tagOf}(c^o) & = & 1 \text{ for boxed values } c^o \\ \texttt{tagOf}(i) & = & 0 \\ \texttt{tagOf}(cls(E, \lambda\overline{x}.e)) & = & 1 \\ \texttt{tagOf}((v_0; v_1, \ldots, v_n)) & = & 1 \end{array}$$

The operational semantics is defined in the style of [14] by giving a set of rules to derive a evaluation relation of the form $E \vdash_B e \Downarrow v$, which reads: "$e$ evaluates to v under E". Figure 8 gives the set of evaluation rules.

In evaluating a record, the runtime system first evaluates the record's bitmap into an integer. After evaluating the record's fields, the runtime system checks the bit tag consistency by comparing the bit tags of the actual fields' values ($\texttt{tagOf}(v_i)$) with the corresponding bit tags obtained from the bitmap's value (B_i). If this check fails, then the evaluation process will go wrong. Later in the soundness theorem we shall show that, for a well typed term, this check never fails. This implies that the evaluation of well-typed program never goes wrong. Hence, in a practical runtime system, we do not need to implement this check.

In the rule for evaluating a bitmap, the value of each bit tag component, i.e. B_i, is evaluated. The bitmap's value is obtained by

$$E \vdash_B c^o \Downarrow c^o$$

$$E \vdash_B x \Downarrow E(x)$$

$$E \vdash_B \lambda\overline{x}.e \Downarrow cls(E, \lambda\overline{x}.e)$$

$$\frac{E \vdash_B e_1 \Downarrow cls(E_0, \lambda\overline{x}.e_0) \quad E \vdash_B \overline{e_2} \Downarrow \overline{v_2} \quad E_0, \overline{x : v_2} \vdash_B e_0 \Downarrow v_0}{E \vdash_B (e_1\ \overline{e_2}) \Downarrow v_0}$$

$$\frac{E \vdash_B e_0 \Downarrow i \quad \text{where } i = [B_1 \circ \cdots \circ B_n] \quad E \vdash_B e_j \Downarrow v_j \quad \texttt{tagOf}(v_j) = B_j \text{ for all } 1 \le j \le n}{E \vdash_B (e_0; e_1, \ldots, e_n) \Downarrow (i; v_1, \ldots, v_n)}$$

$$\frac{E \vdash_B e \Downarrow (v_0; v_1, \ldots, v_n)}{E \vdash_B \pi_i(e) \Downarrow v_i}$$

$$\frac{E \vdash_B e_1 \Downarrow v_1 \quad E, x : v_1 \vdash_B e_2 \Downarrow v_2}{E \vdash_B \texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end} \Downarrow v_2}$$

$$E \vdash_B B \Downarrow B$$

$$\frac{E \vdash_B e_i \Downarrow B_i \quad (1 \le i \le n)}{E \vdash_B [e_1, \ldots, e_n] \Downarrow [B_1 \circ \cdots \circ B_n]}$$

Figure 8. Operational semantics of Λ^B

$$\models_B \emptyset : \emptyset$$

$$\frac{\models_B E : \Gamma \quad \models_B \overline{v} : \overline{\tau}}{\models_B (E, \overline{x : v}) : (\Gamma, arg(\overline{x : \tau}))}$$

$$\frac{\models_B E : \Gamma \quad \models_B v : \sigma}{\models_B (E, x : v) : (\Gamma, local(x : \sigma))}$$

$$\frac{\models_B E : \Gamma}{\models_B E : (\Gamma, tvar(\bar{t}))}$$

Figure 9. Typing rules on environments

setting the least i^{th} bit of a zero word to B_i. This would be done by a sequence of logical bitwise operations. This is the only kind of computation left for the bitmap-passing compilation method. In Section 5, we shall present several optimizations for reducing runtime overhead arising by this kind of computation.

This set of rules should be taken with the following implicit rules yielding $wrong$: if evaluation of any component yields $wrong$ or undefined or does not satisfy the specified condition then the entire term will yield $wrong$. For example, the rules for $(e_0; e_1, \ldots, e_n)$ include, among others, the following one:

$$\frac{E \vdash_B e_0 \Downarrow [B_1 \circ \cdots \circ B_n] \quad E \vdash_B e_1 \Downarrow v_1 \quad \texttt{tagOf}(v_1) \neq B_1}{E \vdash_B (e_0; e_1, \ldots, e_n) \Downarrow wrong}$$

To show that the type system of Λ^B is sound with respect to the operational semantics, we define typing judgments on runtime values and environments of the forms $\models_B v : \sigma$ and $\models_B E : \Gamma$ where σ is a closed type and Γ is a *ground context* (a context that assigns closed types to variables). The set of rules to derive these judgments is given in Figure 10 and Figure 9.

As in a conventional type system, the type system of the target calculus is sound, i.e. it guarantees type-error free evaluation of any type correct expression. We show this in the following theorem.

$$\models_B c^o : o$$

$$\models_B i : \langle\sigma\rangle \text{ if } i = \texttt{tagOf}(\sigma)$$

$$\frac{i = [B_1 \circ \cdots \circ B_n] \quad \models_B B_j : \langle\sigma_j\rangle \quad \text{for all } 1 \le j \le n}{\models_B i : \langle\!\langle \sigma_1, \ldots, \sigma_n \rangle\!\rangle}$$

$$\frac{\begin{array}{c}\text{There exists a ground context } \Gamma \text{ and closed types } \overline{\tau} \\ \models_B E : \Gamma \quad \Gamma, arg(\overline{x:\tau}) \vdash_B e : \sigma\end{array}}{\models_B cls(E, \lambda\overline{x}.e) : \overline{\tau} \to \sigma}$$

$$\frac{\begin{array}{c}\text{There exists a ground context } \Gamma \text{ so that} \\ \models_B E : \Gamma \quad \Gamma, tvar(\overline{t}), arg(\overline{b : \langle t\rangle}) \vdash_B e : \sigma\end{array}}{\models_B cls(E, \lambda\overline{b}.e) : \forall\overline{t}.\overline{\langle t\rangle} \to \sigma}$$

$$\frac{\models_B v_i : \sigma_i \text{ for all } 1 \le i \le n \quad \models_B v_0 : \langle\!\langle \sigma_1, \ldots, \sigma_n \rangle\!\rangle}{\models_B (v_0; v_1, \ldots, v_n) : \sigma_0 \times \cdots \times \sigma_n}$$

Figure 10. Typing rules on values

THEOREM 3. *Let Γ be a context, e be an expression, σ be a type so that $\Gamma \vdash_B e : \sigma$. Let S be a substitution that maps closed types to type variables so that $TV(\Gamma) \subseteq dom(S)$. Let E be a run-time environment so that $\models_B E : S(\Gamma)$. If $E \vdash_B e \Downarrow v$ then $\models_B v : S(\sigma)$.*

This theorem, together with Theorem 1 and Theorem 2, guarantee that the type system of the source calculus is sound with respect to the operational semantics realized by the compilation algorithm (bitmap-passing compilation and type erasure), followed by the evaluation of a compiled term.

3. Combination with Closure Conversion

Closure conversion is an important program transformation for separating between code and data. A function containing free variables is transformed into a *closure* – a data structure consisting of a pointer to a closed code and another data structure which represents the environment or context of the function.

The code is allocated statically, but the environment and the closure, which are considered as records in this paper, need to be allocated in a heap. Generating a bitmap for a closure record is trivial since it is just a pair of a (static) pointer to code and a pointer to a heap object. However, generating a correct bitmap for an environment record requires to perform bitmap-passing compilation. On the other hand, bit tag abstractions generated by bitmap-passing compilation need to be closure converted. Due to this dependency, we implement the bitmap-passing compiler with closure conversion as a combined algorithm, read bitmap-passing closure conversion, which performs bitmap-passing compilation and closure conversion simultaneously.

Consider the following polymorphic function

`f`:$\forall t.t \to t \times t$ = λ`x:t.(x,x)`,

the bitmap-passing compilation inserts a bit tag abstraction for the type variable t. Without considering uncurrying optimization, which we will discuss later, closure conversion must generate two closures: one for bit tag abstractions, the other for the function. Then the combined algorithm would transform the above function to the following explicitly typed term.

```
f:∀t.⟨t⟩ → t → t × t =
Λt.⟪code(nil,b:⟨t⟩,
        ⟪code(⟨t⟩,x:t,([env[0],env[0]];x,x)),
          ([0];b)⟫,
   ([];)⟫
```

`code(`σ`,`$\overline{x:\tau}$`,e)` represents a code of the function $\lambda\overline{x:\tau}.e$ abstracted on an environment record of type σ. $\langle\!\langle e_1, e_2\rangle\!\rangle$ is a closure encapsulating the code e_1 and the environment record e_2. `env[i]` is the access term to the i^{th} field of the environment record.

Similar to the original compilation method, the bitmap-passing closure conversion algorithm would be described in two stages. The first stage transforms a source expression into an explicitly typed target expression. The second one eliminates all type annotations in the resulting term of the first stage to achieve an implicitly typed target term. We have also proved two important theoretical results: the syntactic correctness of the compilation algorithm, and the soundness of the type system of the target calculus. Due to the limitation of space, in this paper, we just demonstrate the bitmap-passing closure conversion by giving the definition of the target calculus – Λ^{BC}, establishing the set of its typing rules, and showing the compilation rules. In an extended version of this paper, we shall give more details of the bitmap-passing closure conversion.

3.1 Syntax and Typing Rules of Λ^{BC}

The target calculus of bitmap-passing closure conversion, Λ^{BC}, is an implicitly typed calculus. We define the syntax of Λ^{BC} by the following grammar.

$$\begin{array}{rcl} e & ::= & c^o \mid x \mid \texttt{env}_i \mid \texttt{code}(\overline{x}, e) \mid \langle\!\langle e, e\rangle\!\rangle \mid (e\ \overline{e}) \\ & & \mid (e; e, \ldots, e) \mid \pi_i(e) \mid \texttt{let } x = e \texttt{ in } e \texttt{ end} \\ & & \mid [e, \ldots, e] \mid B \end{array}$$

$\langle\!\langle e_1, e_2\rangle\!\rangle$ is a function's closure where e_1 is the function's code and e_2 is the function's environment record consisting of all free variables appearing in the function's body. A function's code of the form $\texttt{code}(\overline{x}, e)$ ($\overline{x}$ are the formal parameters, and e is the code's body) is a code abstracted on the function's environment, i.e. each occurrence of a free variable y in the function's body is replaced by $\texttt{env}_i$ in the code's body (i is the index of y in the environment record). Other term constructors are defined the same as in Λ^B.

We extend the set of types of Λ^{BC} by a new category of types for code, i.e. $\sigma_e \to_C \sigma$ where σ_e stands for type of the function's environment and σ represents the type of the function.

A typing context in Λ^{BC} is a tuple $(\Gamma_T; \Gamma_F; \Gamma_A; \Gamma_L)$ where Γ_T is a sequence of type variables, and $\Gamma_F, \Gamma_A, \Gamma_L$ are sequences of type assumptions of the form $x : \sigma$ for recording type information of free variables, arguments and local variables, respectively.

The set of typing rules for Λ^{BC} is given in Figure 11. The reader can notice that the free variable environment does not need to contain free variable names. Free variables are accessed by index in the function's code. The formulation of free variable environment is just to facilitate the bitmap-passing closure conversion algorithm represented later. In Figure 11, we only show the cases relating to closure conversion. Other cases are easily derived from the typing rules of Λ^B.

3.2 The Bitmap-passing Closure Conversion

The key idea of closure conversion is to abstract a function's code over the set of free variables appearing in the function's body. This is traditionally done by first constructing a function's environment as a record consisting of all free variables, then replacing all occurrences of each free variable in the function's body by the corresponding environment access term.

Applying this strategy, the compiler needs to compute the set of free variables in the function's body. Without combining with bitmap-passing compilation, collecting free variable is a simple task. The compiler only needs to inspect all sub-expressions of the function's body to find the necessary information. When combining closure conversion with bitmap-passing compilation, however, this collecting process can not detect the free bit tag variables since

$$(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BC} x : \sigma \quad \text{If } x:\sigma \in \Gamma_A$$

$$(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BC} x : \sigma \quad \text{If } x:\sigma \in \Gamma_L$$

$$\frac{\Gamma_F = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\} \quad i \leq n}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BC} \mathtt{env}_i : \sigma_i}$$

$$\frac{(\Gamma_T;(y_1:\sigma_1,\ldots,y_n:\sigma_n);(\overline{x:\tau});\emptyset) \vdash_{BC} e : \sigma \quad \sigma_{env} = \sigma_1 \times \cdots \times \sigma_n}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BC} \mathtt{code}(\overline{x}, e) : \sigma_{env} \rightarrow_C \overline{\tau} \rightarrow \sigma}$$

$$\frac{((\Gamma_T,\overline{t});(y_1:\sigma_1,\ldots,y_n:\sigma_n);\overline{b:\langle t\rangle};\emptyset) \vdash_{BC} e : \sigma \quad \sigma_{env} = \sigma_1 \times \cdots \times \sigma_n}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BC} \mathtt{code}(\overline{b}, e) : \sigma_{env} \rightarrow_C \forall\overline{t}.\overline{\langle t\rangle} \rightarrow \sigma}$$

$$\frac{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BC} e_1 : \sigma_{env} \rightarrow_C \sigma \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BC} e_2 : \sigma_{env}}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BC} \langle\!\langle e_1, e_2\rangle\!\rangle : \sigma}$$

$$\frac{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BC} e_1 : \overline{\tau} \rightarrow \sigma \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BC} \overline{e_2} : \overline{\tau}}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BC} (e_1\ \overline{e_2}) : \sigma}$$

$$\frac{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BC} e : \forall\overline{t}.\overline{\langle t\rangle} \rightarrow \sigma \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BC} \overline{\tau} \quad (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BC} \overline{e_b} : \overline{\langle\tau\rangle}}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BC} (e\ \overline{e_b}) : \sigma[\overline{\tau}/\overline{t}]}$$

$$\frac{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BC} e_1 : \sigma_1 \quad (\Gamma_T;\Gamma_F;\Gamma_A;(\Gamma_L, x:\sigma_1)) \vdash_{BC} e_2 : \sigma_2}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BC} \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2\ \mathtt{end} : \sigma_2}$$

Figure 11. Typing Rules of Λ^{BC}

they are only available after the compilation process is done for the function's body. Fortunately, since we explicitly record type variables in the context, and each bit tag variable should be generated from a type variable, we can assume a set of free bit tag variables corresponding to the set of free type variables for compiling the function's body. Following this strategy, we define a function $FV_{BC}((\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L), e)$ which computes the set of free variables appearing in a source expression e under the compile context $(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L)$ as follows.

$$\begin{array}{rcl} FV_{BC}((\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L), e) & = & FV(e) \cup \\ & & FBV((\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L)) \end{array}$$

where $FV(e)$ is the traditional free variable collecting function, $FBV((\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L))$ is the function that returns the set of all bit tag variables recording in the context. As the same as bitmap-passing compilation, we assume that for each type variable declared in the compile context, there must be a corresponding bit tag variable recorded in the context. This assumption is always guaranteed by the simulation relation on context (see previous section). Under this assumption, the bit tag creation algorithm never fails and the function FV_{BC} never misses any free variables including bit tag ones.

Using FV_{BC}, we define the bitmap-passing closure conversion algorithm of the form $(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BC} e \rightsquigarrow e'$ where the compile context $(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L)$ is a context in Λ^{BC}, e is the source expression, and e' is the target expression. The only interesting cases are the cases for transforming variables, functions, and type abstractions. Figure 12 shows the compilation rules for these cases.

$$\frac{x:\sigma \in \Gamma_A}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BC} x \rightsquigarrow x}$$

$$\frac{x:\sigma \in \Gamma_L}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BC} x \rightsquigarrow x}$$

$$\frac{x:\sigma \text{ is the } i^{th} \text{ element of } \Gamma_F}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BC} x \rightsquigarrow \mathtt{env}_i}$$

$$\frac{\begin{array}{c} FV_{BC}(e,(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L)) = \{y_1,\ldots,y_n\} \\ (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BC} (y_1,\ldots,y_n) \rightsquigarrow e_{env} \\ (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BC} (y_1,\ldots,y_n) : \sigma_1 \times \cdots \times \sigma_n \\ (\Gamma_T;(\{y_1:\sigma_1,\ldots,y_n:\sigma_n\});\overline{x:\tau};\emptyset) \vdash_{BC} e \rightsquigarrow e' \end{array}}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BC} \lambda\overline{x}.e \rightsquigarrow \langle\!\langle \mathtt{code}(\overline{x}, e'), e_{env}\rangle\!\rangle}$$

$$\frac{\begin{array}{c} FV_{BC}(e,(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L)) = \{y_1,\ldots,y_n\} \\ (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BC} (y_1,\ldots,y_n) \rightsquigarrow e_{env} \\ (\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BC} (y_1,\ldots,y_n) : \sigma_1 \times \cdots \times \sigma_n \\ \overline{b} \text{ are fresh variables} \\ ((\Gamma_T,\overline{t});(\{y_1:\sigma_1,\ldots,y_n:\sigma_n\});\overline{b:\langle t\rangle};\emptyset) \vdash_{BC} e \rightsquigarrow e' \end{array}}{(\Gamma_T;\Gamma_F;\Gamma_A;\Gamma_L) \vdash_{BC} \Lambda\overline{t}.e \rightsquigarrow \langle\!\langle \mathtt{code}(\overline{b}, e'), e_{env}\rangle\!\rangle}$$

Figure 12. Bitmap-passing closure conversion

4. Generating Layout Bitmap for Stack Frames

We have seen how bitmap information for heap-allocated objects are generated in previous sections. For implementing a correct bitmap-passing garbage collector, bitmap information of temporary variables used in each function also needs to be computed.

We consider a stack-based implementation, where a stack frame is allocated for each function, and all the temporary variables used by the function are allocated in the stack frame. The function's code is implemented by a sequence of instructions in three address format $x_1 = op(x_2, x_3)$ where x_1, x_2, x_3 are locations in a stack frame relative to the frame pointer. Compiling a lambda term to three address code can be done by translating it to A-normal form [8] and minimizing the number of stack slots. Our strategy for setting up a bitmap for such a stack frame is first to type each local variable in A-normal form, and to compile a term to *typed* three address code by type-preserving A-normalization [21]. We then minimize the number of stack slots by variable liveness analysis of machine code through proof transformation [22]. A bitmap for a stack frame can then be computed from the types of local variables by the following strategy.

First, we type local variables in three address instructions with one of these types

$$\tau ::= boxed \mid unboxed \mid t$$

where t stands for type variables. Let $\{t_1,\ldots,t_n\}$ be the set of type variable appearing in a stack frame. We represent the layout information of a stack frame by the following data:

1. the number of slots of type $unboxed$,
2. the number of slots of type $boxed$,
3. a set of bit tags types $\{\langle t_1\rangle,\ldots,\langle t_n\rangle\}$, and
4. the number of slots of each t_i.

Second, we refine the type-directed compilation algorithm to compute the necessary layout information. Due to the space limitation, instead of presenting the details of the refined compilation algorithm, we outline the necessary computation steps below.

1. Before the bitmap-passing closure conversion, we identify the set of type variables $\{t_1,\ldots,t_n\}$ that will appear as types

of local variables and arguments, and record it in a function definition. This is easily done by analyzing the types of the set of sub-terms in the function body,

2. The bitmap-passing closure conversion algorithm is refined so that it regards the function body to create an additional record of type $t_1 \times \cdots \times t_n$.
3. After the bitmap-passing closure conversion algorithm, we perform type-preserving A-normalization and code generation to obtain three address code.
4. The type of each address of instruction is evaluated to one of $boxed$, $unboxed$, and t. We then perform variable liveness analysis and compute the maximal number of simultaneously live variables for each type, and assign variables to a frame index.

   This problem can be regarded as register allocation with unbounded number of different kinds of registers (for each type), and can be solved by the technique register-allocation by proof-transformation [22].
5. We produce the stack frame layout by counting the number of slots for each type.

5. Implementation and Optimization

We have integrated all the compilation steps described in our type-directed compiler, SML#, for the full Standard ML language extended with rank-1 polymorphism and record polymorphism. The compiler consists of several type-directed compilation steps, including the following:

- rank-1 type reconstruction,
- polymorphic record compilation,
- bitmap-passing unboxed closure conversion,
- A-normalization, and
- three address code generation.

We have also implemented an abstract machine that executes three address code and a bitmap-inspecting copying collection, and have successfully tested for the compiled code.

Beside the state-of-the-art type system, SML# supports full natural representation for integers, floating point numbers, and other atomic data. This is achieved by applying our bitmap-passing closure conversion and *unboxed compilation* (on which we would like to report elsewhere). As the result, SML# can directly share the heap space with C language. For example, our FFI (foreign function interface) allows to pass an unboxed, natural represented real array to a C function. More information about SML# can be found at [2]

In our implementation, we have dealt with several implementation issues and optimizations. The rest of this section briefly describes these tasks.

5.1 Mutually Recursive Function Definition

The very first thing we have to do for implementing a practical compiler is to extend the type-directed compilation method for dealing with recursion. The extension is not so difficult for monomorphic recursion (i.e. recursive functions are monomorphic). For polymorphic mutually recursive functions, however, the problem of sharing bit tag abstractions among the functions needs to be worked out.

To understand the issue, we let us consider a polymorphic mutually recursive function definition of the form:

$$\begin{array}{l}\texttt{let rec } \forall\bar{t}.f_1 : \overline{\tau_1} \to \sigma_1\texttt{ = } \lambda\overline{x_1 : \tau_1}.e_1 \\ \qquad\qquad \cdots \\ \qquad\qquad f_n : \overline{\tau_n} \to \sigma_n\texttt{ = } \lambda\overline{x_1 : \tau_1}.e_1 \\ \texttt{in e end}\end{array}$$

This term locally defines n polymorphic functions $f_1, \ldots, f_n$ which share the same set of type abstraction $\bar{t}$, and each of them can be referred in the body of any other function. A naive application of the presented compilation method is to insert the same set of bit tag abstractions to all functions. The resulting term would therefore be

$$\begin{array}{l}\texttt{let rec } \forall\bar{t}.f_1 : \overline{\langle t\rangle} \to \overline{\tau_1} \to \sigma_1\texttt{ = } \lambda\overline{b : \langle t\rangle}.\lambda\overline{x_1 : \tau_1}.e'_1 \\ \qquad\qquad \cdots \\ \qquad\qquad f_n : \overline{\langle t\rangle} \to \overline{\tau_n} \to \sigma_n\texttt{ = } \lambda\overline{b : \langle t\rangle}.\lambda\overline{x_n : \tau_n}.e'_n \\ \texttt{in } e' \texttt{ end}\end{array}$$

where $\bar{b}$ are the common bit tag parameters, e' is compiled from e, $e'_1, \ldots, e'_n$ is compiled from $e_1, \ldots, e_n$, and each occurrence of f_i in e_j is substituted by the bit tag application $(f_i\ \bar{b})$ in the target body e'_j.

This simple strategy, however, introduces a serious extra performance cost of passing around the same set of bit tag values among the functions (e.g. each invocation of f_i inside the recursion requires to pass the same set of actual bit tag parameters). We solved this by lifting the bit tag abstractions out of the mutually recursive function definitions. First, each function definition in the source expression is transformed into an ordinary polymorphic function declaration as follows.

$$\begin{array}{l}\texttt{val } f_i : \forall\bar{t}.\overline{\tau_i} \to \sigma_i\texttt{ =} \\ \quad \Lambda\bar{t}.\lambda\overline{x_i : \tau_i}.\texttt{ let rec } f_1 : \overline{\tau_1} \to \sigma_1 = \lambda\overline{x_1 : \tau_1}.e_1 \\ \qquad\qquad\qquad\qquad\qquad\qquad \cdots \\ \qquad\qquad\qquad\qquad\qquad\qquad f_n : \overline{\tau_n} \to \sigma_n = \lambda\overline{x_n : \tau_n}.e_n \\ \qquad\qquad\qquad\qquad \texttt{in } (f_i\ \overline{x_i})\texttt{ end}\end{array}$$

Then, we apply the compilation algorithm extended for monomorphic recursion to produce the target code. Since the mutually recursive function in the transformed term are monomorphic, no bit tag passing is required inside the recursion. Better codes can therefore be achieved. Straightforward application of this strategy would result in duplication of mutually recursive function definitions in every polymorphic function. The sets of the functions' codes generated by closure conversion are, however, identical. Thus the compiler only needs to generate one set, and let all the polymorphic functions share the generated set of codes. In this way, our implementation achieves the efficiency of the resulting code without code duplication.

5.2 User-defined Data Construction

Yet another important issue for implementing a practical compiler is to support user-defined data constructions. ML `datatype` provides a flexible way for programmers to define data structures for their own purposes. Data constructions are heap-allocated objects, they also requires layout bitmaps for garbage collection. Our compilation method, however, does not directly deal with the generation of bitmaps for data constructions. Instead of that, the compiler transforms each data construction into a record, then pass it to the type-directed compilation algorithm. More precisely, a data construction is converted into a pair whose first component represents the data tag, and the second component is the data content.

Let us consider the following example,

```
datatype t = A of int | B of int * bool
val x = A (1)
val y = B (2,3)
```

The variables `x` and `y` are bound to two data constructions which would be transformed into two pairs, $(L_A$`,1)` and $(L_B$`,(2,3))`, before passing to the type-direct compilation. L_A and L_B are data tags that implicitly represent the data constructors `A` and `B`.

Suppose that the data tags are typed by the type τ_{tag} (in our SML# compiler, we use int as the implementation type of τ_{tag}). The user-defined datatype `t` can then be implicitly regarded as a disjoint union of $\tau_{tag} * int$ and $\tau_{tag} * (int * int)$. The first component projection (i.e. `#1`) can be regarded as implicit injection, and the second component projection can be used to extract the data component from a data object (in each case branch after implicit projection). Under this representation, the following `case` expression

```
case x of A m => m | B (n,p) => p
```

can be transformed into

```
case #1 z of L_A => #2 z | L_B => #2 (#2 z)
```

Apparently, this transformation process helps us to reduce the complexity of the type-directed compilation, and gives a uniform treatment of data objects.

5.3 Module Language

Standard ML language supports the large scale programming by its powerful module system. Many approaches have been proposed to compile the Standard ML module language including the recordnizing method by David MacQueen[16], and the flattening method by Martin Elsman[7] (this method has been adopted in our implementation of SML#). Both of these approaches transform modules into ordinary ML Core language's terms (records or flattened declarations). They are totally compatible with our compilation scheme, and do not put any constraint on the generation of bitmaps. Here, just a minor implementation detail about subtyping in the module compilation, which relates to the creation of bit tags, needs to be explained.

Consider the following example

```
structure S1 = struct fun f x = (x,x) end
```

The flattening algorithm of SML# compiler for module compilation would transformed this structure into the following explicitly typed declaration

```
val S1_f : ∀t.t → t × t = Λt.λ x : t. (x,x)
```

Suppose that we have another structure specified by module subtyping as

```
structure S2 = S1 : sig val f : int → int × int end
```

This structure is flattened into a single declaration for the component `f`. This component can be generated by specializing `S1_f` using type information. The resulting declaration would be

```
val S2_f:int → int × int = λx:int.S1_f int x
```

Our type-directed compiler would therefore produce the following target code for `S1_f` and `S2_f`

```
val S1_f:∀t.t → t × t = Λt.λb : ⟨t⟩.λx : t.([b,b],x,x)
val S2_f:int → int × int = λx : int.S1_f int 0 x
```

5.4 Optimizations

In our compilation framework, we have also considered several optimization techniques to minimize runtime overhead arising from bit tag passing and bitmap composition, and have implemented some of them.

Uncurrying. We define the source and all the intermediate calculi to contain multiple-variable lambda abstraction, and perform uncurrying optimization for bit tag variables. With our rank-1 type reconstruction, most type abstractions (and therefore bit tag abstractions) occur at lambda abstraction terms. In this common case, the compiler merges the bit tag abstraction and the lambda abstraction to a single multiple-variable abstraction and records this as its type. Using the recorded type information, the compiler merges nested application of bit tag values and arguments. If such a function is instantiated without its arguments, then the compiler first performs eta-expansion, and then merges the bit tag values and the variables introduced by the eta-expansion. The Figure 13 shows an example of un-currying optimization. In this example, we obtain better code that contains only one function closure instead of the two that would have resulted from the bitmap-passing closure conversion without the un-currying optimization.

Source Untyped ML expression

```
let fun f x = (x,x) in f 1 end
```

Explicitly typed bitmap-passing closure expression

```
let f:∀t.⟨t⟩ → t → t × t =
    Λt.⟪code(nil,b:⟨t⟩,
             ⟪code(⟨t⟩,x:t,([env1,env1];x,x)),
              ([0];b)⟫),
       ([];)⟫
in f int 0 1 end
```

Bitmap-passing closure expression with uncurrying optimization

```
let f:∀t.{⟨t⟩,t} → t × t =
    Λt.<<code(nil,{b:⟨t⟩,x:t},([b,b];x,x)),([];)>>
in f int {0,1} end
```

Figure 13. Examples of uncurrying optimization

Source Untyped ML expression

```
let fun f x = ((x,x),(x,x)) in f 1 end
```

Bitmap-passing expression

```
let f:∀t.⟨t⟩ → t → (t × t) × (t × t) =
    Λt.λb : ⟨t⟩.λx : t.([1,1];([b,b],x,x),([b,b],x,x))
in f int 0 1 end
```

Bitmap-passing expression with sharing bitmap composition optimization

```
let f:∀t.⟨t⟩ → t → (t × t) × (t × t) =
    Λt.λb : ⟨t⟩.λx : t.let bm:[t,t]=[b,b]
                       in ([1,1];(bm,x,x),(bm,x,x))
                       end
in f int 0 1 end
```

Figure 14. Examples of Lifting Bitmap Optimization

Sharing bitmap compositions. Bitmap-passing compilation generates a bitmap composition for each record. Records of the same type have the same bitmap. Sharing the bitmap composition among records of the same type would reduce the runtime overhead arising by bitmap computation. This optimization has the same spirit as common expression elimination. However, we can take advantage from bitmap type recording in explicitly typed terms for designing this optimization efficiently. Our strategy is to lift out every bitmap composition to the nearest dependent bit tag abstraction (i.e. the one whose one or more bit tag arguments appear in the compositions). Once we gathered many bitmap compositions in the same place, we will have a chance to reduce the duplication of bitmap composition. Figure 14 shows an example of the lifting optimization in bitmap-passing compilation. In this example, the bitmap composition `[b,b]` is performed only once in the result of the optimized version instead of twice in the result of the original bitmap-passing compilation algorithm.

This optimization is especially efficient for the case of polymorphic mutually recursive functions. With the original algorithm, a bitmap composition may have to be performed many times (to produce the same bitmap value) inside a runtime function invocation loop. Lifting such bitmap composition out of recursive function definition, we will only have to perform this composition once for an entire recursive function invocation.

Arithmetic optimization. In our implementation, a bitmap composition is realized by a sequence of logical bitwise operations. For example, $[b_1, b_2]$ is realized by

$$\begin{aligned} v_1 &= b_2 \ll 1 \\ v_2 &= v_1 \; AND \; b_1 \end{aligned}$$

Two bitmap compositions may not be the same, but they can share a part of computation. For example, the bitmap $[b_1, b_2, b_3]$ and the bitmap $[b_1, b_2, b_4]$ can share the same arithmetic computation for the first two bits, i.e. $[b_1, b_2]$.

Figuring a bitmap composition as a computation tree, a set of bitmap composition can be represented as a computation forest. One task of arithmetic optimization is to find and to merge identical branches in the forest. Furthermore, some parts of the bitmap computation can be performed statically. For example, the bitmap $[0, 1, b]$ can be computed as

$$\begin{aligned} v_1 &= b_2 \ll 2 \\ v_2 &= v_1 \; AND \; 2 \end{aligned}$$

Since the result of the bitmap computation for first two bits can be statically determined as 2. Thus another task of arithmetic optimization is to find statically computable branches in the computation forest and to replace them by the corresponding static values.

Reducing polymorphism. The major source of runtime overhead in our compilation method is the runtime cost for passing bit tags and for computing bitmaps. This cost is only payed in the presence of polymorphism. Reducing polymorphism would, therefore, help us to reduce the performance cost. One natural way to do this is to duplicate code of polymorphic functions based on their usages. This strategy, however, impairs with the incremental and interactive programming. Fortunately, in our compilation method, we know that a polymorphic function can be sufficiently represented by a small set of its instances. Suppose that bit tags are the only type information which are passed at runtime, a polymorphic function $\Lambda t.\lambda x : t.e$ can be sufficiently implemented by $\lambda x : unboxed.e[unboxed/t]$ and $\lambda x : boxed.e[boxed/t]$ where $boxed$ and $unboxed$ are generic boxed type and generic unboxed type, respectively. In one compilation session, the compiler can generate code for both instances. If this polymorphic function is instantiated in other compilation session, based on the instance type of the function, we can statically determine which instance function can be used in place (if we can not statically determine the function instance, we can still use the original polymorphic function).

6. Conclusions

In this paper, we have presented a type-directed compilation of ML that supports the natural representations of integers and other atomic data. This is achieved by compiling ML so that each runtime object (a heap block or a stack frame) has a "bitmap" that describes the pointer positions in the block. We have defined a typed bitmap-passing calculus with rank-1 polymorphism, and have established that the type system is sound with respect to an operational semantics that models bitmap-inspecting garbage collection. We have then given a type-directed compilation for ML to this calculus, and have shown that it preserves typing. The dependency problem among bitmap compilation, closure conversion and A-normalization has been solved by the proposed combined algorithm. We have implemented the presented algorithms in our SML# compiler.

There are a number of further issues to be addressed. We only mention a few of them below.

Unboxed representation. Most compilers for polymorphic languages adopt *boxed* representations for objects whose sizes exceed one word data. As the consequence, such unnatural representations impair the sharing capability of over-one-word objects such as floating point numbers. We have developed a type-directed compilation method that combines our bitmap-passing compilation and an unboxed compilation which supports a natural representation of over-one-word data. The unboxed compilation method has the same spirit with bitmap-passing compilation where useful information of types are statically computed or abstracted. Here, unboxed compilation uses size information for dealing with vary data width. Many computation of locations and sizes may be added in a naive application of this compilation process. However, they can be either statically computed or optimized in the final code by our proposed optimization techniques. Moreover, the unboxed compilation, as well as bitmap-passing compilation, does not introduce any extra runtime overhead for the monomorphic portion of the source program. Due to the limitation of space, we skip the presentation of unboxed compilation, and would like to report this elsewhere.

Separate compilation. An important assumption of the bitmap-passing compilation is that each type variable must be bound to a type abstraction. This assumption guarantees that the bit tag creation algorithm always succeeds. In separate compilation, a type variable may represent an abstract user-defined type whose implementation type is given in another compile unit. In such cases, the original bit tag creation algorithm will fail since it can not statically compute the corresponding bit tag for the type variable. Treating abstract types as generalized types is one solution to overcome this problem. This, however, adds many unnecessary runtime overhead by our type-directed compilation method. Fortunately, we can observe that at link time when all compiled units are linked together to form an executable code, actual types of all user-defined type variables are known. Then, we can just postpone the computation of bit tag values from abstract types until link time. Refining bitmap-passing compilation following this strategy is, therefore, feasible, and this would be an important future issue.

Acknowledgments

The authors thank Liu Bochao and Kiyoshi Yamatodani for discussion and comments during our implementation of the SML# compiler. which have been very helpful in our better understanding and development of the bitmap-passing compilation method presented in this paper.

References

[1] http://www.mlton.org.

[2] http://www.pllab.riec.tohoku.ac.jp/smlsharp/.

[3] N. Benton, A. Kennedy, and G. Russell. Compiling Standard ML to Java bytecodes. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1), pages 129–140, 1999.

[4] N. Benton, A. Kennedy, and C. V. Russo. Adventures in interoperability: the SML.net experience. In *PPDP '04: Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 215–226, New York, NY, USA, 2004. ACM Press.

[5] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, 1970.

[6] K. Crary, S. Weirich, and J. G. Morrisett. Intensional polymorphism in type-erasure semantics. In *International Conference on Functional Programming*, pages 301–312, 1998.

[7] M. Elsman. Program modules, separate compilation, and intermodule optimization.

[8] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings ACM SIGPLAN 1993 Conf. on Programming Language Design and Implementation, PLDI'93, Albuquerque, NM, USA, 23–25 June 1993*, volume 28(6), pages 237–247. ACM Press, New York, 1993.

[9] P. Fradet. Collecting more garbage. In *LFP '94: Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 24–33, New York, NY, USA, 1994. ACM Press.

[10] B. Goldberg. Tag-free garbage collection for strongly typed programming languages. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 165–176, New York, NY, USA, 1991. ACM Press.

[11] C. V. Hall, K. Hammond, S. L. P. Jones, and P. L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, March 1996.

[12] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, California, 1995.

[13] A. Igarashi and N. Kobayashi. Garbage collection based on a linear type system, 2000.

[14] G. Kahn. Natural semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Sciences on STACS 87*, pages 22–39, London, UK, 1987. Springer-Verlag.

[15] X. Leroy. Unboxed objects and polymorphic typing. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 177–188, Albequerque, New Mexico, 1992.

[16] D. MacQueen. An implementation of standard ML modules. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming, Snowbird, UT*, pages 212–223, New York, NY, 1988. ACM.

[17] Y. Minamide, J. G. Morrisett, and R. Harper. Typed closure conversion. In *Symposium on Principles of Programming Languages*, pages 271–283, 1996.

[18] G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 66–77, New York, NY, USA, 1995. ACM Press.

[19] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.

[20] A. Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, November 1995.

[21] A. Ohori. A curry-howard isomorphism for compilation and program execution. In *Typed Lambda Calculus and Applications*, pages 280–294, 1999.

[22] A. Ohori. Register allocation by proof transformation. *Sci. Comput. Program.*, 50(1-3):161–187, 2004.

[23] A. Ohori and N. Yoshida. Type inference with rank 1 polymorphism for type-directed compilation of ML. In *International Conference on Functional Programming*, pages 160–171, 1999.

[24] J. Peterson and M. Jones. Implementing type classes. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 227–236, New York, NY, USA, 1993. ACM Press.

[25] B. Saha and Z. Shao. Optimal type lifting. In *TIC '98: Proceedings of the Second International Workshop on Types in Compilation*, pages 156–177, London, UK, 1998. Springer-Verlag.

[26] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 181–192, 1996.

[27] A. P. Tolmach. Tag-free garbage collection using explicit type parameters. In *LISP and Functional Programming*, pages 1–11, 1994.