Finitary Polymorphism for Optimizing Type-Directed Compilation

ATSUSHI OHORI, Tohoku University, JAPAN
KATSUHIRO UENO, Tohoku University, JAPAN
HISAYUKI MIMA, Tohoku University, JAPAN

We develop a type-theoretical method for optimizing type directed compilation of polymorphic languages, implement the method in SML#, which is a full-scale compiler of Standard ML extended with several advanced features that require type-passing operational semantics, and report its effectiveness through performance evaluation. For this purpose, we first define a predicative second-order lambda calculus with finitary polymorphism, where each type abstraction is explicitly constrained to a finite type universe, and establishes the type soundness with respect to a type-passing operational semantics. Different from a calculus with stratified type universes, type universes of the calculus are terms that represent a finite set of instance types. We then develop a universe reconstruction algorithm that takes a term of the standard second-order lambda calculus, checks if the term is typable with finitary polymorphism, and, if typable, constructs a term in the calculus of finitary polymorphism. Based on these results, we present a type-based optimization method for polymorphic functions. Since our formalism is based on the second-order lambda calculus, it can be used to optimize various polymorphic languages. We implement the optimization method for native (tag-free) data representation and record polymorphism, and evaluate its effectiveness through benchmarks. The evaluation shows that 83.79% of type passing abstractions are eliminated, and achieves the average of 15.28% speed-up of compiled code.

CCS Concepts: • Theory of computation → Type theory; • Software and its engineering → Functional languages; Compilers;

Additional Key Words and Phrases: Finitary Polymorphism, Second-Order Lambda Calculus, Type-Directed Compilation, Compiler Optimization, SML#

ACM Reference Format:
Atsushi Ohori, Katsuhiro Ueno, and Hisayuki Mima. 2018. Finitary Polymorphism for Optimizing Type-Directed Compilation. Proc. ACM Program. Lang. 2, ICFP, Article 81 (September 2018), 29 pages. https://doi.org/10.1145/3236776

1 INTRODUCTION

The general motivation of this study is to develop an optimization method for type-directed compilation of polymorphic languages with various advanced features such as record polymorphism [Ohori 1995], type classes [Hall et al. 1996], and others. When an ML-style polymorphic language is extended with some of those features, the conventional implementation for “parametric polymorphism” is no longer applicable, and type-directed compilation must be employed. To see the

Authors’ addresses: Atsushi Ohori, Research Institute of Electrical Communication, Tohoku University, Katahira 2-1-1, Aoba-ku, Sendai, Miyagi, 980-8577, JAPAN, ohori@riec.tohoku.ac.jp; Katsuhiro Ueno, Research Institute of Electrical Communication, Tohoku University, Katahira 2-1-1, Aoba-ku, Sendai, Miyagi, 980-8577, JAPAN, katsu@riec.tohoku.ac.jp; Hisayuki Mima, (Current address) DWANGO Co., Ltd. KABUKIZA TOWER., 4-12-15 Ginza, Chuo-ku, Tokyo, 104-0061, JAPAN, hisayuki_mima@dwango.co.jp.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2018 Copyright held by the owner/author(s).
2475-1421/2018/9-ART81
https://doi.org/10.1145/3236776
need, consider the field selection operation in a polymorphic record calculus

\[
\#\text{age} : \forall s.\forall t[\#\text{age} : s].t \rightarrow s
\]

that extracts the age field from a record of type \(t\) containing an \(\text{age} : s\) field. This operation depends on the offset of the age field, which differs in the individual instance types of \(t\). This type-dependent operation cannot be compiled to efficient code by the conventional compilation method for polymorphic functions based on parametric polymorphism. One solution to this problem is to compile this function to a higher-order function of the from

\[
\lambda I : \text{offset}(\text{age})(t) . \lambda x : t.x[I] : \forall s.\forall t[\#\text{age} : s].\text{offset}(\text{age})(t) \rightarrow t \rightarrow s
\]

by inserting an extra lambda abstraction to receive the necessary type dependent attribute, in this case the offset of the age field of type \(t\). The type \(\text{offset}(\text{age})(t)\) of extra parameter \(I\) is an example of singleton type, denoting the offset value of age field of type \(t\). Similar mechanism has been used to compile type classes (first-class overloading).

In addition to those advanced functionalities such as record polymorphism, type-directed compilation is also required for those features such as tag-free garbage collection and native data representation that should be useful in any polymorphic languages. For example, compiling to native code with native data representation is essential for a polymorphic language to take full advantages of rapidly progressing system features such as efficient concurrent threads on multicore processors. Indeed, one of the major obstacles for high-level polymorphic languages to scale to thread programming on multicore processors is the lack of interoperability with a system thread library provided through C API. Most of the currently implemented polymorphic language compilers adopt “uniform representations” for closures and other data structures with inline tags, which make interface to system libraries quite difficult. The type-directed compilation method provides a uniform solution. For example, consider the very simple code

\[
pair = \lambda x.(x, 3.14) : \forall t.t \rightarrow t \times \text{real}
\]

which takes a value of type \(t\) and allocate a tuple \(t \times \text{real}\). Compiling this function to native data representation implies that for any argument value of type \(\tau\), this function create a heap allocated block containing \(\tau\) and \text{real} (\text{double in C}) compatible with the corresponding \text{struct} in C. The type directed compiler first annotates the type variable \(t\) with kind \text{TAG} and \text{SIZE} as follows

\[
pair = \lambda x.(x, 3.14) : \forall t[\text{TAG, SIZE}].t \rightarrow t \times \text{real}
\]

to indicate that the size and the tag attributes are needed for \(t\) at runtime, and then generates the following code that uses these type attributes at runtime:

\[
pair = \lambda t:\text{TAG} \cdot \lambda s:\text{SIZE} \cdot \lambda x : t.x.t.\ « \text{code to allocate a block of type } t \times \text{real} »
\]

\[
: \forall t[\text{TAG, SIZE}].\text{tag}(t) \rightarrow \text{size}(t) \rightarrow t \rightarrow t \times \text{real}
\]

where \(\text{tag}(t)\) and \(\text{size}(t)\) are singleton types denoting the size and tag of type \(t\). This method is implemented in the SML\# compiler for achieving direct C interface. Since polymorphic functions applied to some monomorphic instance behave the same as the monomorphic function of the instance type, ML’s first-class higher-order polymorphic functions, for example, can be directly passed to POSIX thread library through the standard C API.

We should note that native data representation alone can also be achieved by performing whole program analysis and making specialized copies of a polymorphic function, as proposed and implemented in the MLton compiler [Weeks 2006]. The drawback is the difficulty of implementing interactive compilation and separate compilation of large systems. Type-directed compilation achieves native data representation without incurring these limitations, and it also uniformly support various other advanced features such as record polymorphism.
Various forms of type-directed compilation have been exploited in a number of works including tag-free garbage collection \cite{Tolmach 1994}, intentional type analysis \cite{Harper and Morrisett 1995}, compiling with typed intermediate representation \cite{Tarditi et al. 1996}, tag-free polymorphic equality \cite{Elsman 1998}, and native data representation \cite{Nguyen and Ohori 2006}. With these successful results, we believe that type-directed compilation is one of viable approaches for polymorphic languages to achieve various advanced features. Nevertheless, one problem of this type-directed compilation approach is the high runtime overhead for passing type attributes and auxiliary computation on these type attributes to obtain the same monomorphic semantics as that of the specialized instance. To observe the problem, consider the term `pair` again. To compile this function so that it always creates data in native data representation compatible with that of C, the compiler need to pass the size and tag values at runtime. In addition, the compiler also needs to generate code that allocates a two-field record, with the second field aligned to double word boundary, and footer information to indicate whether the first field is a pointer or not for a tracing garbage collection to work. For example, for the following applications of the function `pair`:

\[
\begin{align*}
  a &= \text{pair } 1 : \text{int} \ast \text{real} \\
  b &= \text{pair } (1, 1) : (\text{int} \ast \text{int}) \ast \text{real},
\end{align*}
\]

the compiled code of `pair` must generate the following heap objects:

| word32 offset | record $a$ | record $b$ |
|---------------|------------|------------|
| 0             | 1          | a pointer to a record $(1, 1)$ |
| 1             | padding    |
| 2             | 3.14       | 3.14       |
| 3             | 0x0        | 0x1        |

where the extra 4th-word is a layout bitmap for garbage collection; 0x1 of record $b$, for example, indicates that the first double word is a pointer and the second is a non-pointer. As seen in the example, type-directed compilation of a polymorphic function to native data representation incurs a significant runtime overhead in passing extra arguments and computing runtime layout, compared to the standard uniform representation with inline tags.

This may be regarded as the price to pay for a polymorphic language to enjoy various advanced features including native data representation. The motivation of this work is to reduce this price to minimal by eliminating the unnecessary overhead. Our development is based on the following simple and natural observation. In an ML-style implicitly typed polymorphic language, all functions are given their most general types, and many of them are polymorphic types. Apparently, this is gross over-generalization. Locally-defined small functions may have very few sets of different instances, typically only one in many cases. Hence, if we can define a typed intermediate calculus that precisely represents the set of instances for each type abstraction, then we can use such a typed calculus as an intermediate language to optimize the compiled code by suppressing redundant polymorphism.

This observation is not new. In a general perspective, MLton’s approach and other optimization approaches of duplication of polymorphic functions for integers, reals, and pointers all share this general idea and observation. The original contributions of the present paper is to develop a type-theoretical framework for finitary polymorphism to analyze the problem, to propose a practical optimization algorithm that achieves optimal specialization of polymorphic functions, and to demonstrate the feasibility through implementation of an optimizer in a full-scale ML compiler and evaluating the effectiveness. As we discuss in Section 9, various researches on intersection types
also share the observation similar to ours. However, there does not seem to exist a second-order type derivation system that represents and enforces finitary polymorphism.

Specific technical contributions of the paper include the following.

- We define a calculus with finitary polymorphism. It is a second-order calculus where each type abstraction is constrained to a finite type universe. This feature is essential for our purpose of tracing the exact set of instances. The type system is shown to be sound with respect to call-by-value type-passing operational semantics. Although the purpose of this development is to apply it to optimizing type-directed compilation, the calculus itself should be of some interest in studying typability and type inference in predicative second-order type systems. The feature that distinguishes our type system from the existing predicative calculi is that universes are terms and type membership in a universe is a syntactic judgment in the type system. This structure enables us to trace the exact set of instance types of each type abstraction. We believe that this is the first successful attempt to define a predicative second-order calculus without assuming a predefined hierarchy of type universes.

- We develop a universe reconstruction algorithm for the second-order lambda calculus. Since the second-order lambda calculus is impredicative, admitting a form of self-application, not all terms are typable in our calculus. The universe reconstruction algorithm checks whether or not a given term is typable in our calculus, and if it is typable, then it constructs a term in our calculus with the minimal universes.

- We develop an optimization algorithm of type-directed compilation based on finitary polymorphism. For this purpose we introduce, for each type attribute, a universe name lattice whose elements correspond to individual attribute values. Using this static structure, the optimization algorithm is able to statically evaluate (some of) reconstructed universes to attribute values and to suppress type attribute passing.

- We extend the optimization algorithm to separate compilation. In addition, we present a simple code-duplication specialization of polymorphic library functions based on the type information with finitary polymorphism. Compared with existing ad-hoc approach, our method allows to select a specialized version even in the body of a polymorphic function.

- We apply these results to the SML# compiler, and develop an optimizer. The optimizer transforms a typed intermediate language to our calculus with finitary polymorphism, and implement the optimization algorithm (except for separate compilation extension) and the library specialization algorithm. The implementation is available in the artifact accompanied with this paper. See the README file in the artifact for the details.

- We evaluate the effectiveness of our implementation on a set of benchmark programs. The evaluation shows that 83.79% of type attribute abstractions are eliminated, and achieves the average of 15.28% speed-up of compiled code. These results demonstrate the significance and effectiveness of our method.

The rest of the paper is organized as follows. Section 2 defines the calculus with finitary polymorphism. Section 3 shows type soundness theorem with respect to an operational semantics that models type-passing implementation. Section 4 defines a variant of the calculus with closed universe constraints that is useful in developing the universe reconstruction algorithm shown in the next section. Section 5 presents an algorithm that converts (a typable subset of) the standard second-order lambda calculus to the calculus with finitary polymorphism. Section 6 develops an optimization method for type-directed compiler based on the proposed calculus and the universe reconstruction algorithm. Section 7 presents an extension of the optimization algorithm to separate compilation. It also proposes a simple code-duplication specialization of polymorphic
library functions based on finitary polymorphism. Section 8 describes the implementation and reports performance results. Section 9 compares this work with related works. Section 10 concludes the paper with suggestions for further investigations.

2 THE CALCULUS WITH FINITARY POLYMORPHISM

This section defines the second-order calculus with finitary polymorphism.

Let TV be a given countably infinite set of type variables ranged over by \( t \). We also use the following meta variables: \( b \) for atomic types, \( \sigma \) for types, \( c^b \) for constants of type \( b \), \( U \) for type universes or simply universes, \( M \) for terms, and \( l \) for position labels. Position labels are used for identifying the locations of type applications. They are not essential for typing derivation, but they turn out to be useful not only for code analysis and understanding but also for reconstructing universes that we shall develop in Section 5.

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

\[
M ::= \ c^b \mid x \mid \lambda x : M \mid M M \mid \Lambda t : U.M \mid (M \sigma)^l \mid (M, M) \mid \#1 M \mid \#2 M \\
\sigma ::= b \mid t \mid \forall t : U.\sigma \mid \sigma \to \sigma \mid \sigma \times \sigma
\]

This is the standard definition of the second-order lambda calculus except that in type abstraction term \( \Lambda t :: U.M \) and universal type \( \forall t :: U.\sigma \), the universe of \( t \) is explicitly specified with \( U \) defined below. \((M \sigma)^l\) is the type application at (unique) position \( l \).

The intended denotation of a type universe \( U \) in \( \Lambda t :: U.M \) and \( \forall t :: U.\sigma \) is the set of types over which type variable \( t \) ranges. The standard practice in defining a predicative calculus is to follow the general framework of ramified type hierarchy [Russell 1908] and to construct a hierarchy of universes and an associated set of type variables, inductively. Although this approach restricts typable terms to be predicative, the range of each type variables remain infinite. Our purpose is to define a calculus that tracks the exact set of type instances of each type abstraction. For this purpose, we need to introduce type universes explicitly as terms roughly of the form \( \{l_1 : \sigma_1, \ldots, l_n : \sigma_n\} \) indicating the fact that the type abstraction term is applied to types \( \sigma_i \) at positions \( l_i \) (\( 1 \leq i \leq n \)).

To define a calculus with a universe term construction mechanism, we consider a universe constraint of a type variable as a property assigned to a type variable and introduce a universe environment \( \mathcal{U} \) of type variables. We also define type instances, ranged over by \( \rho \), that include types with a universe assignment of type variable of the form \( \sigma(t :: U) \). This syntax denotes a type \( \sigma \) whose type variable \( t \) ranges over the universe \( U \). This form of instance is needed for a type variable to be used as an instance of another polymorphic function. The syntax for universe environments, type instances, and universes are given below.

\[
\mathcal{U} ::= \emptyset \mid \mathcal{U}(t :: U) \\
\rho ::= \sigma \mid \rho(t :: U) \\
U ::= \{l : \rho, \ldots, l :: \rho\}
\]

We write \( \text{FTV}(X) \) for the set of free type variables that occurs in any syntactic object \( X \) containing type variables. Their definitions are obtained by inductively extending the following phrases according to the structure of syntax of \( X \).

\[
\text{FTV}(t) = \{t\} \\
\text{FTV}(\forall t :: U.\sigma) = \text{FTV}(U) \cup (\text{FTV}(\sigma) \setminus \{t\}) \\
\text{FTV}(\rho(t :: U)) = \text{FTV}(U) \cup (\text{FTV}(\rho) \setminus \{t\})
\]

In this calculus, type variables appearing in syntactic objects and judgments must be consistently assigned their universe. We call this property well-formedness. We write \( \vdash \mathcal{U} \) if \( \mathcal{U} \) is well-formed.
We also write \( \mathcal{U} \vdash \sigma \), \( \mathcal{U} \vdash \rho \), and \( \mathcal{U} \vdash \sigma \) for \( \sigma \), \( \rho \), and \( \mathcal{U} \) are well-formed under \( \mathcal{U} \). They are defined by the following (well-founded mutually inductive) rules:

- **Well-formed universe environments.**
  \[
  \vdash \emptyset \\
  \vdash \mathcal{U} \quad t \notin \text{dom}(\mathcal{U}) \\
  \vdash \mathcal{U}[t : U]
  \]

- **Well-formed types, instances, and universes under \( \mathcal{U} \).**
  \[
  \mathcal{U} \vdash \sigma \quad \text{if } \text{FTV}(\sigma) \subseteq \text{dom}(\mathcal{U}) \\
  \mathcal{U} \vdash \rho \quad \text{if } \text{FTV}(\rho) \subseteq \text{dom}(\mathcal{U}) \\
  \mathcal{U} \vdash \mathcal{U} \quad \text{if } \text{FTV}(\mathcal{U}) \subseteq \text{dom}(\mathcal{U})
  \]

To define a type system of the calculus, we need the following two auxiliary relations:

- **\( \mathcal{U} \vdash \rho_1 = \rho_2 \) for \( \rho_1 \) is equal to \( \rho_2 \) under a well-formed universe environment \( \mathcal{U} \).**
  \[
  \mathcal{U} \vdash \sigma = \sigma \\
  t : U \in \mathcal{U} \\
  \mathcal{U} \vdash \rho_1 = \rho_2 \\
  \mathcal{U} \vdash \rho_1 = \rho_2(t : U)
  \]

- **\( \mathcal{U} \vdash l : \sigma \in U \) for \( \sigma \) is a member of universe \( U \) at \( l \) under well-formed universe environment \( \mathcal{U} \).**
  \[
  l : \rho \in U \\
  \mathcal{U} \vdash \sigma = \rho \\
  \mathcal{U} \vdash l : \sigma \in U
  \]

Let \( \mathcal{T} \) be a type assignment that is a function from a finite set of variables to types. \( \mathcal{T} \) is well-formed under \( \mathcal{U} \), written \( \mathcal{U} \vdash \mathcal{T} \), if \( \text{FTV}(\mathcal{T}) \subseteq \text{dom}(\mathcal{U}) \). We write \( \mathcal{T}[x : \sigma] \) for the extension of \( \mathcal{T} \) with \( x \) assigned to \( \sigma \). We assume the standard bound variable convention and assume \( x \notin \text{dom}(\mathcal{T}) \) in this notation.

With these syntactic entities, we define the type system of the calculus as a proof system to derive a typing judgment of the form \( \mathcal{U}, \mathcal{T} \vdash M : \sigma \). The set of typing rules is given in Figure 1.

This is a refinement of the type system of the second-order lambda calculus, where each type abstraction is explicitly constrained to a finite universe. This constraint eliminates the impredicative property of the second-order lambda calculus, such as the term

\[
(\lambda x : (\forall t. t \to t). (x (\forall t. t \to t)) x) (\Lambda t. \lambda y : t.y)
\]

where the second-order type \( \forall t. t \to t \) abstracts over the set of all types including itself. Owing to this property, the typability of our calculus is equivalent to a predicative second-order type system as in [Leivant 1991], where the set of types are divided into a predefined set of type universes.
The feature that distinguishes our type system from these ramified type systems is the mechanism to construct arbitrary finite sets of type universes. For our purpose of compiler optimization, this property is essential for representing the exact type-instantiation information involved in polymorphic types. As an example, consider the following term in the second-order lambda calculus (with location annotations):

\[ \emptyset \vdash (\lambda X : \forall t.t \to t. \lambda Y : \forall s.s \to s.(X \; \text{int}^1.1, \; (((X \; \forall s.s \to s)^2 Y) \; \text{int}^1.2)) \]

\[ (\Lambda t.\lambda x : t.x) \]

\[ (\Lambda s.\lambda y : s.y) \]

\[ : \text{int} \to \text{int} \]

This computes a pair \((1, 2)\) using two polymorphic identity functions \((\Lambda t.\lambda x : t.x)\) and \((\Lambda s.\lambda y : s.y)\). The first one is instantiated to \text{int} at \(l_1\) and to \(\forall s.s \to s\) at \(l_2\), and the second one is instantiated to \text{int} at \(l_3\). This term is typed in our calculus as follows:

\[ \emptyset, \emptyset \vdash (\lambda X : \forall t :: [l_1 : \text{int}, l_2 : \forall s :: [l_3 : \text{int}].s \to s].t \to t. \lambda Y : \forall s :: [l_3 : \text{int}].s \to s.(X \; \text{int}^1.1, (((X \; \forall s :: [l_3 : \text{int}].s \to s)^2 Y) \; \text{int}^1.2)) \]

\[ (\Lambda :: [l_1 : \text{int}, l_2 : \forall s :: [l_3 : \text{int}].s \to s].\lambda x : t.x) \]

\[ (\Lambda :: [l_3 : \text{int}].\lambda y : s.y) \]

\[ : \text{int} \to \text{int} \]

As seen in this example, all the type applications, including those of the terms involving higher-order functions such as the type application at \(l_3\), are precisely traced and represented at the definition of type abstraction.

To precisely trace all the type instantiation in the context of higher-order terms with bound and free type variables, the type system introduces instance terms with bound type variables of the form \(\rho(t :: U)\). Figures 2 and 3 show two typical examples involving type application to bound type variables. To facilitate easy understanding, these examples show the corresponding terms in the second-order lambda calculus. In the example in Figure 2, the polymorphic function \(Y\) is applied to type \(t \times t\) containing type variable \(t\). Since the occurrence of this type application is in the scope of the binding \(\Lambda t\) of \(t\), \(t\) is properly assigned with its universe. However, the flexibility of the second-order lambda calculus also allows application to a type containing a type variable outside its scope as shown in Figure 3. The term in the second-order lambda calculus does not involve any impredicative nature, and therefore it should be typable in a predicative calculus including ours. The subtlety is that the universe of the type abstraction \(\Lambda t\) needs to include \(s \times s\) with type variable \(s\). However, \(s\) is bound outside of the scope of \(\Lambda t\), and therefore there must be some mechanism to associate its universe that is consistent with its binding. To properly deal with this situation, we introduced instance types with bound type variables of the form \(\rho(t :: U)\). In this example, to apply \(X\) to \(s \times s\), the universe of \(t\) contains \(s \times s(s :: [a : \text{int}, b : \text{bool}])\).

### 3 OPERATIONAL SEMANTICS AND TYPE SOUNDNESS

This calculus intends to model type-passing implementation of polymorphic languages including ML-style languages. For this purpose, we define an operational semantics as a call-by-value big-step evaluation relation. In addition to the standard operational semantics of the (untyped) lambda calculus with value environment, ranged over by \(E\), a type abstraction term in this calculus is implemented as a function that takes an instance type. To model this, we introduce type substitutions, ranged over by \(S\), and define the evaluation relation as a relation of the form

\[ S, E \vdash M \Downarrow v \]
(in the second-order calculus)
\[ \Theta, \emptyset \vdash (\lambda X : \forall t.t \to \text{int} \times (t \times t). ((X \text{ int})^a 1, (X \text{ bool})^b \text{ true})) \]
\[ (\Lambda t. \lambda x : t. \]
\[ (\lambda Y : \forall ss \times s \to s \times s. ((Y \text{ int})^c 1, (Y \text{ t} \times t)^d (x, x))) \]
\[ (\Lambda s. (X s \times s)^c) \]
\[ (\lambda x : t.x) \]
\[ : (\text{int} \times (\text{int} \times \text{int})) \times (\text{int} \times (\text{bool} \times \text{bool})) \]  

Fig. 2. Typing example: type variable application within its scope

(in the second-order calculus)
\[ \Theta, \emptyset \vdash (\lambda X : \forall t.t \to t. \]
\[ (\lambda Y : \forall ss \times \to s \times s. ((Y \text{ int})^a 1, (Y \text{ bool})^b \text{ true})) \]
\[ (\Lambda s. (X s \times s)^c) \]
\[ (\lambda x : t.x) \]
\[ : (\text{int} \times \text{int}) \times (\text{bool} \times \text{bool}) \]  

Fig. 3. Typing example: type variable application outside of the scope

indicating the fact that \( M \) evaluates to value \( v \) under type substitution \( S \) and variable environment \( E \). The set of values, value environments, and type substitutions are defined as follows:

\[ v ::= c^b \mid \text{cls}(S, E, x, M) \mid \text{tcls}(S, E, t::U, M) \mid \text{Wrong} \]
\[ E ::= \{x : v, \ldots, x : v\} \]
\[ S ::= [\sigma/t, \ldots, \sigma/t] \]

\( \text{cls}(S, E, x, M) \) is a function closure with the formal parameter \( x \) and \( \text{tcls}(S, E, t::U, M) \) is a type function closure with the formal type parameter \( t \). \( \text{Wrong} \) represents the runtime error. In the following, we write \( E[x : v] \) and \( S[\sigma/t] \) for the extension of \( E \) and \( S \) provided that \( x \notin \text{dom}(E) \) and \( t \notin \text{dom}(S) \).

To model syntactic constraints on type instantiation, we define the following semantic relations.

- \( S \models t \in \sigma \in U \) (\( \sigma \) is a member of \( U \) at \( t \) under \( S \))

\[
\frac{l : \rho \in U \quad S \models \sigma = \rho}{S \models l : \sigma \in U}
\]
\[
S, E \vdash c^b \downarrow c^b \quad S, E \vdash x : v \vdash x \downarrow v
\]

\[
S, E \vdash \lambda x : \sigma_1. M_1 \downarrow \text{cls}(S, E, x, M) \quad S, E \vdash \Lambda t :: U. M \downarrow \text{tcls}(S, E, t :: U, M)
\]

\[
S, E \vdash M_2 \downarrow \text{tcls}(S_0, E_0, t :: U, M_0) \quad S, E \models l : \sigma_0 \in S_0(U) \quad S_0[S_0(t)/t], E_0 \vdash M_0 \downarrow v
\]

\[
S, E \vdash \text{cl}(S_0, E_0, x, M_0) \quad S, E \vdash M_2 \downarrow v_0 \quad S_0, E_0(x : v_0) \vdash M_0 \downarrow v
\]

\[
S, E \vdash (M_1.M_2) \downarrow v \quad S, E \vdash \{v_1, v_2\} \quad (i \in \{1, 2\})
\]

Fig. 4. Evaluation Relation

- \( S \models \rho_1 = \rho_2 \) (\( \rho_1 \) is equivalent to \( \rho_2 \) under \( S \))

\[
S(\sigma_1) = S(\sigma_2) \quad S \models \sigma_1 = \sigma_2
\]

\[
S' \models l : S(t) \in S(U) \quad (\text{for some } S') \quad S \models \rho_1 = \rho_2
\]

Using these relations, the set of evaluation relation is given in Figure 4. These rules should be taken with the implicit rules indicating the fact that if some of the conditions specified as a semantic relation or a value pattern are not satisfied, then the evaluation yields \textit{Wrong}. This relation faithfully models instance-constrained type-passing operational semantics.

We now show that the type system is sound with respect to the operational semantics. We first define semantic typing relations. We write \( \models v : \sigma \) to indicate that a value \( v \) has a type \( \sigma \), and write \( \models E : \mathcal{T} \) to indicate that \( E \) is a model of \( \mathcal{T} \). The definitions of these relations are given below:

- \( \models v : \sigma \) if one of the following holds:
  - \( \models c^b : b \)
  - \( \models \text{cls}(S, E, x, M) : \sigma_1 \rightarrow \sigma_2 \) if there are some \( \mathcal{U}, \mathcal{T}, \sigma_1, \) and \( \sigma_2 \) such that \( \models S : \mathcal{U}, \models E : S(\mathcal{T}), S(\sigma_i : \rightarrow \sigma_i') = \sigma_1 \rightarrow \sigma_2, \) and \( \mathcal{U}, \mathcal{T} \vdash \lambda x : \sigma_i.M : \sigma_i \rightarrow \sigma_i' \).
  - \( \models \text{tcls}(S, E, t :: U, M) : \forall t :: U. \sigma \) if there are some \( \mathcal{U}, \mathcal{T}, \) and \( \sigma' \) such that \( \models S : \mathcal{U}, \models E : S(\mathcal{T}), S(\forall t :: U'. \sigma') = \forall t :: U. \sigma, \) and \( \mathcal{U}, \mathcal{T} \vdash \lambda t :: U'.M : \forall t :: U'. \sigma' \).

- \( \models E : \mathcal{T} \) if \( \text{dom}(E) = \text{dom}(\mathcal{T}) \) and \( \models E(x) : \mathcal{T}(x) \) for any \( x \in \text{dom}(E) \).

A substitution \( S \) is a model of \( \mathcal{U} \), written \( \models S : \mathcal{U} \), if this is derived by the following rules:

\[
\models \emptyset : \emptyset \quad \models S[\mathcal{U}] \quad \models S[t :: U] \quad \models S[\sigma/t] : \mathcal{U}[t :: U]
\]

The following relates the syntactic constraint on instantiation to the semantic counterpart.

**Lemma 3.1.** If \( \mathcal{U} \vdash l : \sigma \in U \) and \( \models S : \mathcal{U} \) then \( \models l : \sigma \in U \)

**Proof.** We first show

if \( \mathcal{U} \vdash \rho_1 = \rho_2 \) and \( \models S : \mathcal{U} \), then \( \models \rho_1 = \rho_2 \)
by induction on the derivation of \( \mathcal{U} \vdash \rho_1 = \rho_2 \). The base case is trivial. Suppose \( \rho_2 = \rho'_2(t :: U) \) such that \( t :: U \in \mathcal{U} \) and \( \mathcal{U} \vdash \rho_1 = \rho'_2 \). By the induction hypothesis, \( \models \rho_1 = \rho'_2 ; t :: U \in \mathcal{U} \) and \( \models S : \mathcal{U} \) implies \( S' \models l' : S(t) \in S(U) \) for some \( S' \) and \( l' \). Then we have \( S \models \rho_1 = \rho'_2(t :: U)(\equiv \rho_2) \) by...
definition of this relation. To prove the lemma, suppose \( \Gamma \vdash l : \sigma \in U \). Then there is some \( l : \rho \in U \) such that \( \Gamma \vdash \sigma = \rho \). Therefore, we have \( S \models \sigma = \rho \). By definition, we have \( S \models l : \sigma \in U \). □

Using this, we now show the following.

**Theorem 3.2.** If \( \Gamma, \tau \vdash M : \sigma \), \( S \models \Gamma, \tau \), \( = E : S(\tau) \), and \( S, E \vdash M \Downarrow \nu \), then \( \models \nu : S(\sigma) \).

**Proof.** This is proved by induction on the length of evaluation relation \( S, E \vdash M \Downarrow \nu \). We proceed by cases on the structure of \( M \).

Case \( M \equiv x \). The typing relation and evaluation relation must be of the forms:

- \( \Gamma, \tau \vdash x : \sigma \), and
- \( S, E[x : v] \vdash x \Downarrow \nu \).

By assumption, \( \models E[x : v] : S(\tau \vdash x : \sigma) \). Thus, we have \( \models \nu : S(\sigma) \), as desired.

Case \( M \equiv (M_0 \sigma_0)^t \). The typing relations and evaluation relations must have been:

- \( \Gamma, \tau \vdash M_0 : \forall t :: U. \sigma_1 \),
- \( \Gamma, \tau \vdash (M_0 \sigma_0)^t : [\sigma_0/t] \sigma_1 \),
- \( \Gamma \vdash l : \sigma_0 \in U \),
- \( S, E \vdash M_0 \Downarrow \nu_0 \), and
- \( S, E \vdash (M_0 \sigma_0)^t \Downarrow \nu \).

By the induction hypothesis, \( \models \nu_0 : S(\forall t :: U. \sigma_1) \) and therefore \( \nu_0 \) must be of the form \( tcls(S', E', t :: U'. M') \). By the value typing, there are some \( \Gamma', \tau' \), and \( \sigma' \) such that \( \models \Gamma' : \forall t :: U'. \sigma' \), \( \models \tau' : U'. M' :: \forall t :: U'. \sigma' \), and \( S(\forall t :: U. \sigma') = S(\forall t :: U. \sigma_1) \). Since \( \Gamma \vdash l : \sigma_0 \in U \), we have \( S \models l : \sigma_0 \in U \) by Lemma 3.1. Consequently, \( \nu \) is derived by the evaluation relation \( S'(\sigma_0)/t \), \( E' \vdash M' \Downarrow \nu \). By the typing rule, \( \models (M_0 \sigma_0)^t : \forall t :: U. \sigma' \). Since \( S \models l : \sigma_0 \in U \) implies \( S \models l : S(\sigma_0) \in S(U) \) by the definition of this relation. Since \( S(\forall t :: U. \sigma') = S(\forall t :: U. \sigma_1) \) where \( t \) can be assumed to be fresh, we have \( S'(U') = S(U) \) and \( S'(\sigma') = S(\sigma_1) \). Then we have \( \models S'(\sigma_0)/t : U \Downarrow U \). By the induction hypothesis, \( \models \nu : S'(\sigma_0)/t(\sigma') \). Since \( S'(\sigma') = S(\sigma_1) \), we have \( \models \nu : S(\sigma_0)/t(\sigma') \), as desired.

Case \( M \equiv \Lambda t :: U. M \). The typing relation and evaluation relation must be of the forms:

- \( \Gamma, \tau \vdash \Lambda t :: U. M : \forall t :: U. \sigma \), and
- \( S, E \vdash \Lambda t :: U. M \Downarrow \nu \).

By the evaluation relation, \( \nu = tcls(S, E, t :: U, M) \). By the value typing, \( \models tcls(S, E, t :: U, M) : S(\forall t :: U. \sigma) \).

Other cases are standard. □

**4 THE CALCULUS WITH CLOSED UNIVERSES**

The predicative nature of the type system allows us to define an alternative presentation of the calculus with closed type universes. By the definition of a model of a universe environment \( \Gamma \), it is easily checked, by induction of the construction, that a universe assigned to a type variable has a straightforward set theoretical denotation under set theoretical interpretations of each base type, function type constructor, and type quantification over a given universe. From this observation, it is straightforward to define the syntax for closed universes, i.e., those that do not contain free type variables. The idea is to attach a universe to each type variable appearing in a universe and consider it as a universe variable, and do not consider a free type variable. The set of closed instances, ranged over by \( \pi \), closed universes, ranged over by \( C \), and closed instance environments, ranged over by \( E \),

---

**Proc. ACM Program. Lang., Vol. 2, No. ICFP, Article 81. Publication date: September 2018.**
are defined below.

\[ \pi ::= b \mid t^C \mid \forall t :: C. \pi \mid \pi \rightarrow \pi \mid \pi \times \pi \]

\[ C ::= \{ l : \pi, \ldots, l : \pi \} \]

\[ \emptyset ::= \emptyset \mid \emptyset t :: C \]

Let \( \emptyset \) be a type assignment in this calculus.

Type equivalence under \( \emptyset \), written \( \emptyset \vdash \sigma = \pi \), is redefined to be the syntactic extension of the following clause:

\[ \emptyset \vdash t = t^C \quad (\text{if } \emptyset(t) = C) \]

Intuitively, \( t^C \) denotes the set of instances of \( C \), but it acts as a type variable \( t \) when it is used to check type instance membership. The other definitions remain the same as before.

We refer to the resulting calculus as \( \Lambda^C \) and the calculus defined in Section 2 as \( \Lambda^U \). It is easily seen that for any typing in \( \Lambda^U \), there is a corresponding typing in \( \Lambda^C \). The definition of \( C \) in \( \Lambda^C \) is superficially more general than \( U \) in \( \Lambda^U \) since, for example, \( C \) may contain an instance of the form \( l : (\pi_1 \times \pi_2) \) with different \( C_1 \) and \( C_2 \). This extra generality is vacuous since \( \emptyset \vdash t \times t = t^C_1 \times t^C_2 \) will never hold in any \( \emptyset \) and therefore such instance will never be used in type application and can be eliminated without affecting typability. Indeed, the two calculi are equivalent in typability in the sense of the following proposition.

**Proposition 1.** For any structure \( X \) containing types, let erase(\( X \)) be the structure obtained from \( X \) by erasing universe annotations. Let \( \Gamma \vdash M : \sigma \) be any typing judgment in the second-order lambda calculus. There is a typing \( \subseteq, \emptyset \vdash M_1 : \sigma_1 \) in \( \Lambda^U \) such that erase(\( M_1 \)) = \( M \), erase(\( \emptyset \)) = \( \Gamma \), and erase(\( \sigma_1 \)) = \( \sigma \) if and only if there is a typing \( \emptyset \vdash M_2 : \sigma_2 \) in \( \Lambda^C \) such that erase(\( M_2 \)) = \( M \), erase(\( \emptyset \)) = \( \Gamma \), erase(\( \sigma_2 \)) = \( \sigma \)

Figure 5 shows the terms in \( \Lambda^C \) that correspond to those shown in Figure 2 and Figure 3. Note that type variables in both cases including the first one in the scope of universe bound are uniformly represented in the form of \( t^C \). The resulting terms are notationally cumbersome. However, this presentation is useful for developing a universe reconstruction algorithm, to which we turn.

## 5 Universe Reconstruction

In this section, we develop a universe reconstruction algorithm, which takes a term in the second-order lambda calculus and returns a term in \( \Lambda^C \) presented in the previous section if it is typable,
and reports failure if it is not typable. Our strategy is to reduce the universe reconstruction problem to a simple variant of the type inference problem of the polymorphic record calculus with kinded unification presented in [Ohori 1995].

A universe of the form
\[ \ldots, l : \pi, \ldots \]
in the type system of our calculus is naturally regarded as a record with location labels \( l \) as field labels and instance types \( \sigma \) as field values, and a type application
\[ (M \sigma)^l \]
of a universe-bound polymorphic term \( M \) to type \( \sigma \) corresponds to selecting the \( l \) field from the universe of the polymorphic type of \( M \), which must be of the form
\[ \forall t : \ldots, l : \pi, \ldots. \tau \]
such that \( C \vdash \sigma = \pi \). This observation leads to the following strategy for universe reconstruction:

- We regard universes \( C \) as record terms and the condition \( (l : \pi) \in C \) associated with typing constraint \( C \vdash l : \sigma \in C \) of type application \( (M \sigma)^l \) as a field selection term.
- From this correspondence, we represent universe constraints of the type system of \( \Lambda^C \) as terms of a typed record calculus whose typability is equivalent to conformity of universe constraints.
- From a typing derivation of the second-order lambda calculus with fresh universe variables, we extract record terms that represent universe constraints, and infer a principal typing of the term. The solution substitution of a principal typing represents the minimal universes that satisfy the universe constraint.

We first define a simple record calculus. The terms of the record calculus represent universe constraints of type variables in \( \Lambda^C \). To reflect this role, the set \( TV \) of type variables and types (\( \sigma \)) of \( \Lambda^C \) are respectively taken as variables and terms (algebraic datatypes) of the record calculus. For these terms, the type system of the record calculus assigns the types of the record calculus that represent universe constraints in \( \Lambda^C \). To define such a type system, we introduce a set \( UV \) of universe variables (ranged over by \( \alpha \)) as the set of type variables of the record calculus, and introduce type terms as constant types in the record calculus. The definition of the record calculus is given in Figure 6. \#l \( r \) selects \( l \) field from record (variable) \( r \). \( r = r \) and \((r; \cdots ; r)\) are equality check and sequential evaluation, respectively. Note that, for our purpose, record expressions and complete record types are not needed.

This calculus is a subset of the polymorphic record calculus given in [Ohori 1995], for which we have a type inference algorithm that computes a principal typing of any typable term. The following definitions are taken from [Ohori 1995]. A substitution \( S \) is well-formed under a kind assignment \( \mathcal{K} \) if for any \( t \in \text{dom}(S), \text{FTV}(S(t)) \subseteq \text{dom}(\mathcal{K}) \). A kinded substitution is a pair \((\mathcal{K}, S)\) of a kind assignment \( \mathcal{K} \) and a substitution \( S \) that is well-formed under \( \mathcal{K} \). The kind assignment \( \mathcal{K} \) in \((\mathcal{K}, S)\) specifies kind constraints of the result of the substitution. The following theorem is extracted from the corresponding type inference theorem shown in [Ohori 1995].

**Theorem 5.1 (Record type inference).** There is an algorithm \( WK \) such that for any term \( r \), kind assignment \( \mathcal{K} \), and type assignment \( \Sigma \), if \( WK(\mathcal{K}, \Sigma, e) = (\mathcal{K}', S, \tau) \), then the following properties hold:

1. \((\mathcal{K}', S)\) respects \( \mathcal{K} \) and \((\mathcal{K}', S(\Sigma)) \vdash e : \tau \),
2. if \( \mathcal{K}_0, S_0(\Sigma) \vdash e : \tau_0 \) for some \((\mathcal{K}_0, S_0)\) such that \((\mathcal{K}_0, S_0)\) respects \( \mathcal{K} \), then there is some \( S' \) such that \((\mathcal{K}_0, S')\) respects \( \mathcal{K}' \), \( \tau_0 = S'(\tau) \), and \( S_0(\Sigma) = S'(S(\Sigma)) \).

If \( WK(\mathcal{K}, \Sigma, e) \) fails, then there is no \((\mathcal{K}_0, S_0)\) and \( \tau_0 \) such that \((\mathcal{K}_0, S_0)\) respects \( \mathcal{K} \) and \((\mathcal{K}_0, S_0(\Sigma)) \vdash e : \tau_0 \).
Kinds \( (k) \), types \( (\tau) \), terms \( (\alpha) \), kind environments \( (\mathcal{K}) \).

\[
\begin{align*}
\tau & \ ::= \: \sigma \mid \top \mid \tau \mid \tau \ 
\end{align*}
\]

\[
\begin{align*}
\mathcal{K} & \ ::= \ \{ \alpha_1 : k_1, \ldots , \alpha_n : k_n \} \\
\Sigma & \ ::= \ \{ t_1 : \alpha_1, \ldots , t_n : \alpha_n \}
\end{align*}
\]

Kinding rules

\[
\begin{align*}
\mathcal{K} \{ \alpha :: \{ l : \tau, \ldots , l : \tau, \ldots \} \} & \vdash \alpha :: \{ l : \tau, \ldots , l : \tau \}
\end{align*}
\]

Typing rules

\[
\begin{align*}
\mathcal{K}, \Sigma \vdash b : b & \quad \mathcal{K}, \Sigma \vdash t : \alpha \text{ (if } \Sigma(t) = \alpha) & \quad \mathcal{K}, \Sigma \vdash \sigma : \delta \quad \mathcal{K}, \Sigma \vdash \forall t. \sigma : \forall t. \delta \\
\mathcal{K}, \Sigma \vdash \sigma_1 : \delta_1(i = \{1,2\}) & \quad \mathcal{K}, \Sigma \vdash \sigma_1 : \delta_1(i = \{1,2\}) & \quad \mathcal{K}, \Sigma \vdash \sigma_1 \times \sigma_2 : \delta_1 \times \delta_2 \\
\mathcal{K}, \Sigma \vdash r : \tau_1 & \quad \mathcal{K}, \Sigma \vdash \tau_1 :: \{ l : \tau_2 \} & \quad \mathcal{K}, \Sigma \vdash \tau_1 :: \{ l : \tau_2 \} & \quad \mathcal{K}, \Sigma \vdash r_1 : \tau & \mathcal{K}, \Sigma \vdash r_2 : \tau \\
\mathcal{K}, \Sigma \vdash \#l \ r : \tau_2 & \quad \mathcal{K}, \Sigma \vdash r_1 = r_2 : \text{unit} \\
\mathcal{K}, \Sigma \vdash (r_1; \cdots ; r_n) : \text{unit}
\end{align*}
\]

Fig. 6. A Simple Record Calculus

\[
\begin{align*}
\{ \alpha : \{ a : \text{int}, b : \text{bool} \}, \beta : \{ c : \text{int}, d : \alpha \times \alpha \} \}, \{ t : \alpha, s : \beta \} \\
\vdash (\#a \ t = \text{int}; \#b \ t = \text{bool}; \#c \ s = \text{int}; \#d \ s = t \times t) : \text{unit}
\end{align*}
\]

\[
\begin{align*}
\{ \alpha : \{ a : \text{int}, b : \text{bool} \}, \beta : \{ c : \alpha \times \alpha \} \}, \{ s : \alpha, t : \beta \} \\
\vdash (\#a \ t = \text{int}; \#b \ t = \text{bool}; \#c \ t = s \times s) : \text{unit}
\end{align*}
\]

Fig. 7. Principal Typing Examples

Figure 7 shows two examples of principal typings that represent universe constraints associated with type applications in the example terms of Figure 5. From these examples, it should be clear that the record calculus is expressive enough to reconstruct universe annotations.

Using the algorithm \( \mathcal{WK} \), we define the universe reconstruction algorithm that takes a typing derivation of the second-order lambda calculus and returns the terms with universe annotation. Let \( \Gamma \vdash M : \sigma \) be a given term. We can assume that this typing is closed, and does not contain free type variables. The algorithm consists of the following steps:

1. We convert the giving typing \( \Gamma \vdash M : \sigma \) to a typing that satisfies the bound type variable condition, i.e., all bound type variables are distinct, and construct a typing derivation. A typing derivation is represented as a term where all sub-terms have type annotations.
(2) Let $TV(M)$ be the set of bound type variables in $M$. Construct the initial kind and type environment as follows.

$$\Sigma = \{t : \alpha \mid t \in TV(M), \alpha \text{ fresh}\}$$

$$\mathcal{K} = \{\alpha_1 : \emptyset, \ldots, \alpha_n : \emptyset\}$$

(3) Inductively traverse the reconstructed type annotated term $M : \sigma$ and generate terms of the record calculus as follows.

- If $M$ is a type application $((M : \forall t. \sigma_1) \sigma)^t : \sigma_2$, then generate the term $\#_t t = \sigma$.
- If $M$ is an application $(M_1 : \sigma_1 \rightarrow \sigma_2)(M_2 : \sigma_3)$, then $\sigma_1$ and $\sigma_3$ only differ in the name of bound type variables. For each pair of corresponding types $(\forall t. \sigma, \forall s. \sigma')$ in $\sigma_1$ and $\sigma_3$, generate term $t = s$.

Generate term $r$ as $(r_1; \ldots; r_n)$ by combining all the generated terms $r_1, \ldots, r_n$.

(4) Apply the type inference algorithm $\mathcal{WK}$ to $(\mathcal{K}, \Sigma, r)$. If it fails then the term is not typable in $\Lambda^C$. We describe the treatment of failure in Section 8. Suppose $\mathcal{WK}(\mathcal{K}, \Sigma, r) = (\mathcal{K}', S, \tau)$. Let $\{t_1, \ldots, t_n\} = dom(\Sigma)$. Let $S(\Sigma) = \{t_1 : \alpha^1, \ldots, t_n : \alpha^n\}$. Let $\{\alpha_1, \ldots, \alpha_j\}$ be the set of distinct universe variables of $\{\alpha^1, \ldots, \alpha^n\}$. For each $\alpha$ in $\{\alpha_1, \ldots, \alpha_j\}$, let $C_{\alpha}$ be the term obtained from $\mathcal{K}'(S(\Sigma(t_j)))$ by substituting each $\alpha$ with $\alpha C_{\alpha}$. Let $\Gamma, M, \sigma$ and $\tau$ be the assignment, term, and type of $\Lambda^C$ that are obtained from $\Gamma, M$, and $\sigma$ by performing the following substitution.

- Let $\forall t$ and $\Delta t$ be type abstractions in $\Gamma$ or $M$. Let $\alpha = S(\Sigma(t))$. Replace $\forall t$ with $\forall \alpha :: C_{\alpha}$, and $\Delta t$ with $\Delta t$.

Substitute each type variable $t$ with $S(\Sigma(t))$.

The resulting typing $\emptyset, \Gamma \vdash \overline{M} : \overline{\sigma}$ is a provable typing in $\Lambda^C$.

Figure 8 shows an example of universe reconstruction.

6 OPTIMIZING TYPE-DIRECTED COMPILATION

Besides theoretical interests, our main purpose of developing the calculus with finitary polymorphism is to use it as a typed intermediate representation of a type-directed compiler of a polymorphic language. In this section, we present an optimization method for a type-directed compiler of an ML-style polymorphic language with type dependent features. Note that the calculus is a second-order calculus and therefore can be used for optimizing various polymorphic languages not limited to ML.

We based our development on the general structure of compilation method presented in [Ohori 1995], which is perhaps the first proposal in literature (originally presented in [Ohori 1992]) of a type-theoretical framework for type-directed compilation considered in this study. The rationale behind this choice is the following. Firstly, in addition to its original target of record polymorphism, the framework is uniformly applicable for compiling out various advanced features including native (tag-free) data representation which subsumes tag-free GC, and a simple form of overloading. Secondly, this framework is implemented in SML# [SML#-Project-Team 2018], an open-source full scale Standard ML compiler. Indeed, SML# supports record polymorphism, native data representation, and a simple form of overloading based on this framework. The SML# compiler provides us a suitable basis to implement and evaluate the optimization method we shall present in this section.

A concrete definition of a source language depends, of course, on the type dependent features the source language supports. As suggested in [Ohori 1999], it should be possible to define an abstract model of ML-style polymorphic calculi with type dependent features. For our purpose of presenting optimization method, it is sufficient to describe the general structure of type-directed compilation method using some examples. Here we use native data representation and record polymorphism explained in Introduction as examples of type dependent features.
(1) Given typing:

$$\emptyset \vdash (\lambda x : \forall t.t \rightarrow t. \quad (\lambda y : s.s \times s \rightarrow s \times s.((y \int^a) (1, 2), (y \ boolean^b) (true, false)))$$

$$\Lambda u. (x u x u)$$

$$\Lambda v. \lambda x : v.x : (int \times int) \times (boolean \times boolean)$$

(2) Typing derivation (type annotated term):

$$\emptyset \vdash (\lambda x : \forall t.t \rightarrow t. \quad (\lambda y : s.s \times s \rightarrow s \times s.((y \int^a) (1, 2), (y \ boolean^b) (true, false)))$$

$$\emptyset \vdash (\forall s.s \rightarrow s \times s) \rightarrow (int \times int) \times (boolean \times boolean) \quad (\lambda u. (x : \forall t.t \rightarrow t. u u u u)$$

$$\emptyset \vdash (\forall t.t \rightarrow t) \rightarrow (int \times int) \times (boolean \times boolean) \quad \Lambda v. \lambda x : v.x : (v.v \rightarrow v)$$

$$\emptyset \vdash (int \times int) \times (boolean \times boolean)$$

(3) Generated type and kind environments in the record calculus.

$$\Sigma = \{s : \alpha, t : \beta, u : \gamma, v : \delta\}$$

$$\mathcal{K} = \{\alpha : \emptyset, \beta : \emptyset, \gamma : \emptyset, \delta : \emptyset\}$$

(4) The generated term in the record calculus.

$$r = (#a s = int; \#b s = boolean; u = s; \#c t = u x u; \; v = t)$$

(5) Solution of the type inference

$$\forall \mathcal{K}(\mathcal{K}, \Sigma, r) = \{\alpha : [[a : int, b : boolean]], \beta : [[c : \alpha \times \alpha]], \; [\alpha / \gamma, \beta / \delta]\}$$

(6) The result

$$\emptyset, \emptyset \vdash (\lambda x : \forall \beta. \forall \beta : [c : \alpha^{a:primitive,b:boolean} \times \alpha^{a:primitive,b:boolean}], \beta \rightarrow \beta$$

$$\emptyset, \emptyset \vdash (\lambda y : \forall \alpha. \forall \alpha : [\alpha : int, b : boolean], \alpha \times \alpha \rightarrow \alpha \times \alpha.$$ 

$$\emptyset, \emptyset \vdash (\forall \int^a (1, 2), \forall \boolean^b (true, false)))$$

$$\emptyset, \emptyset \vdash (\forall \alpha \times \alpha)$$

$$\emptyset, \emptyset \vdash \lambda \alpha : [\alpha : int, b : boolean].(x \alpha \times \alpha)$$

$$\emptyset, \emptyset \vdash \lambda \beta : [c : \alpha^{a:primitive,b:boolean} \times \alpha^{a:primitive,b:boolean}], \lambda x : \beta.$$ 

$$\emptyset, \emptyset \vdash (int \times int) \times (boolean \times boolean)$$

Fig. 8. A universe reconstruction example

A type-directed compilation in the style of [Ohori 1995] consists of the following phases:

(1) Kinded type reconstruction.

The first step is to perform kinded type inference on an untyped source term and converts it to an explicitly typed term. In the case of a polymorphic function, the inferred polymorphic type contains kind constraints on type variables, which represent the type attributes to be passed at runtime for realizing type-dependent semantics. We write those polymorphic functions with kinded type abstraction in the following syntax

$$\Lambda \sharp K. \lambda x : t. M : \forall t\sharp K. t \rightarrow \sigma$$

where $$\Lambda \sharp K$$ is kinded type abstraction with kind $$K$$.

For example, if a term $$e$$ of type $$t$$ is used to construct a tuple in native data representation, then $$t$$ is kinded with SIZE and TAG, indicating that size and tag information of $$t$$ must be
passed at runtime. After kinded type reconstruction, an untyped term \( \lambda x. (x, 3.14) \) becomes
\[
\Lambda t[#(\text{TAG}, \text{SIZE})]. \lambda x : t. (x, 3.14) : \forall t[#(\text{TAG}, \text{SIZE})]. t \rightarrow t * \text{real}.
\]

If the polymorphic record operation \(#\text{age}\) is applied to a term of type \( t \), then \( t \) is kinded with a record kind \( \{\text{age} : s\} \) indicating that the age offset value must be passed at runtime. After kinded type reconstruction, an untyped term \( \lambda x. #\text{age} \ x \) becomes
\[
\Lambda t[#(\text{age} : s)]. \lambda x : t. #\text{age} \ x : \forall s. \forall t[#(\text{age} : s)]. t \rightarrow s.
\]

(2) Type-directed transformation.
This phase compiles each type dependent polymorphic term into a combination of primitive operations using type attribute information. In the case of natural data representation, a tuple constructor is compiled to code that allocates a heap block of appropriate size, updates its fields with element values, and creates a tag vector indicating whether each field is a pointer or not. In the case of record polymorphism, a polymorphic field selector such as \(#\text{age} \ e\) is compiled to code that computes the integer offset corresponding to the age field of the record type of \( e \) and issues the memory load instruction from the record with the computed integer offset.

This phase consists of the following two steps.
(a) Type-attribute term generation.
The compiler generates type-attribute terms, denoted here by \( \text{attTerm}(\tau) \), to represent the necessary attribute values, and uses them to compile terms involving polymorphic constructors. In the case of pair constructors, its attribute values include \( \text{sizeTerm}(t_i) \) for the size of each field and \( \text{tagTerm}(t_i) \) for the tag of each field. In the case of field selection, the necessary value is \( \text{offsetTerm}(l)(\tau) \) for the offset of \( l \) in the record type \( \tau \).
(b) Type-attribute passing transformation.
If these type-attribute terms are inside of a polymorphic function, then they are of the form \( \text{attTerm}(t) \) that cannot be evaluated at compile time. In such cases, the type-directed compiler performs type-attribute passing transformation by introducing singleton type \( \text{attTy}(t) \) for \( \text{attTerm}(t) \) and converting a polymorphic function \( f = \Lambda t. M \) and its type application \( f \tau \) to \( \Lambda t. \lambda x : \text{attTy}(t). M \) and \( f \tau \ \text{attTerm}(\tau) \).

For example, type reconstructed term
\[
\text{let pair} = \Lambda t[#(\text{age} : s)]. \lambda x : t. #\text{age} \ x : \forall s. \forall t[#(\text{age} : s)]. t \rightarrow s
\]
\[
\text{in pair} \ \{\text{age} : \text{int}, \text{id} : \text{int}\} \ \{\text{age} = 21, \text{id} = 101\}
\]
is converted to
\[
\text{let pair} = \Lambda t[#(\text{age} : s)]. \lambda l : \text{offset}(\text{age})(t). \lambda x : t. x[l]
\]
\[
\quad : \forall s. \forall t[#(\text{age} : s)]. \text{offset}(\text{age}, t) \rightarrow t \rightarrow s
\]
\[
\text{in pair} \ 0 \ \{\text{age} : \text{int}, \text{id} : \text{int}\} \ \{21, 101\}.
\]
In this case, the type attribute value \( \text{offsetTerm}(\text{age})(\{\text{age} : \text{int}, \text{id} : \text{int}\}) \) is statically reduced to 0.

The main target of optimization is to reduce the type-attribute passing overhead generated in phase 2-b above. The optimizer takes a type-reconstructed kinded polymorphic function such as the above and converts it to a universe annotated term using the universe reconstruction algorithm presented in the previous section. After universe reconstruction, a kinded polymorphic function is typically of the form
\[
\Lambda t#K :: C. \lambda x : t. M
\]
where \( C \) represent the set of instances of \( t \). From the definition of \( \Lambda^C \), \( C \) always represents a finite set of closed types. Location labels \( l : \pi \) in \( C \) is useful in program analysis but they are not used in optimization. Based on these properties, in the following development, we sometimes regard \( C \) as a finite set of closed types and use set notation. For example, \( \{a : \{k:int, c:bool\} \times s|d:real\} \) represents the set \( \{int \times real, bool \times real\} \). In such case, we sometimes write \( C = \{int \times real, bool \times real\} \).

For the term \( \Lambda#K::C,\lambda x:t.M \), there are two possibilities of optimization.

1. The first is the obvious case where the set \( C \) represents either empty or a singleton \( \{\tau\} \). The former indicates that the function is never applied, and is subject to dead code elimination. In the latter case, the polymorphic term can be specialized to \( \lambda x : \tau.M \).

2. Even when \( C \) represents a set of multiple elements, some attribute \( attTerm(t) \) can still be statically computed. A simple example is \( tag(t) \) where \( C = \{int, real\} \). Since the native representation of \( int \) and \( real \) are both non pointers, \( tagTerm(int) \) and \( tagTerm(real) \) are both \( unboxed \), and therefore \( attTerm(t) \) is statically evaluates to \( unboxed \). This situation also happens for computed attributes. In the case of record polymorphism, \( offsetTerm(age)(t) \) for \( C = \{(age : int, name : string), (age : int, distiller : string, region : string)\} \) is statically computed to 0, since in both cases, the age offset is 0 under the canonical representation of heap allocated records whose components are sorted according to their labels.

These observations lead us to construct an optimization method by lifting static attribute computation on individual types to sets of types, i.e. \( universes \), and representing type-attribute terms as \( attTerm(C) \) so that attribute values are computed by universe information \( C \). For example, if \( tag\_unboxed \) denotes the set of types whose representation is \( unboxed \) and \( C \subseteq tag\_unboxed \), then \( tagTerm(t) = unboxed \).

Let \( ATT \) be the set of type attributes used in the type-directed compiler. Our optimization method consists of defining data structures called \( universe name lattices \) and two optimizing transformations \( type abstraction specialization \) and \( universe-based attribute-passing optimization \). The details of the method are described below.

1. Construction of \( universe name lattices on the powerset of types \).
   Let \( att \in ATT \) be any type attribute, and let \( \{v_1, \ldots, v_n, \ldots\} \) be the set of all possible values of \( att \). For each \( v_i \), we introduce a universe name \( att\_v_i \) for the set of types \( \{\tau | attTerm(\tau) = v_i\} \). Then the set \( \{att\_v_1, \ldots, att\_v_n, \ldots\} \) denotes a set of subsets of types that partitions the set of all types. Let \( att\_unknown \) and \( att\_EMPTY \) be the special universe names denoting the set of all types, and the empty set, respectively. By this construction, \( att\_v_1, \ldots, att\_v_n, \ldots, att\_unknown, att\_EMPTY \) form a flat lattice with the set inclusion ordering of their denotation, with the top element \( att\_unknown \) and the bottom element \( att\_EMPTY \). We call this lattice \( L_{att} \).

In the case of natural data representation, the set of relevant attributes are \( \{tag, size\} \), and the corresponding sets of possible values are \( \{boxed, unboxed\} \) and \( \{1, 2, 4, 8\} \) (in bytes), respectively. For \( tag \), the universe name \( tag\_boxed \) denotes the set of all types whose runtime representation is a pointer, and \( tag\_unboxed \) denotes the set of all types whose runtime representation is a non-pointer. In the case of \( size \) attribute, the universe name \( size\_8 \), for example, denotes the set of all types whose runtime size is 8 bytes, including \( real \) and all the heap allocated types.

We note that both the set of relevant attributes and the corresponding attribute values may be unbounded. For example, in the case of record polymorphism, there are countably many relevant attributes of the form \( offset(l) \) for each label \( l \). Moreover, for each attribute \( offset(l) \), the set of possible values are integer offsets of label \( l \), which is unbounded. Accordingly, \( L_{offset(l)} \) consists of an infinite set of universe names \( offset(l)_0, offset(l)_4, offset(l)_8, \ldots \).
each of which denotes the set of record types containing \( l \) field at the specified offset. For example, the universe name \( \text{offset} (\text{age}) \cdot 0 \) in \( L \cdot \text{offset} (\text{age}) \) denotes the set of record types containing age filed at offset 0.

(2) **Attribute value computation with universe name lattices.**

Let \( \text{att} \) be any type attribute, and let \( L _ { \text{att}} \) be the universe name lattice of \( \text{att} \) defined above. By the construction, for any subset \( C \) of types, there is a unique minimal element \( \text{att} _ * \) in \( L _ { \text{att}} \) such that \( C \subseteq \text{att} _ * \), and this set \( \text{att} _ * \) is statically computed.

For example, in \( L _ { \text{tag}} \), \( \text{att} _ * \) is \( \text{att} _ { \text{unboxed}} \) for \( C = \{ \text{int, bool} \} \). In \( L _ { \text{offset} (\text{age})} \) of record polymorphism, \( \text{offset} (\text{age}) _ * \) for \( C = \{ \{ \text{age} : \text{int}, \text{name} : \text{string} \}, \{ \text{age} : \text{int}, \text{distiller} : \text{string}, \text{region} : \text{string} \} \} \) is \( \text{offset} (\text{age}) _ 0 \).

(3) **Type abstraction specialization.**

Let \( \Lambda \# K :: C . M \) be the type abstraction term obtained by the composition of type reconstruction and universe reconstruction. The optimizer transforms this term according to \( C \) as follows:

(a) **Dead-function elimination:** If \( C \) is empty or only contains instances that involve type variables annotated with empty bound, then the term is converted to \( \Lambda t :: \text{EMPTY}.M \). This function is never applied and therefore this is dead code. The case of empty \( C \) is obvious by construction of our type system. The case where \( C \) contains a type \( \tau \) with \( s ^ 0 \) means that this function is applied to \( \tau \) inside of a term of the form \( \Lambda s :: \text{EMPTY}.M' \). Since the term \( \Lambda s :: \text{EMPTY}.M' \) is not applied to any type, \( M' \) is not executed. Therefore, a type application to a type \( \tau \) containing \( s ^ 0 \) is not executed. This term is subject to elimination in a later phase.

(b) **Monomorphic specialization:** If \( C \) is a singleton set \( \{ \tau \} \), then it is converted to \( M[\tau / t] \), and the corresponding type applications are eliminated.

(c) **Universe-name-based attribute computation:** If \( C \) contain multiple elements, then for each type attribute \( \text{att}_i \in ATT \) of the set of \( ATT \) represented by the kind \( K \), we compute the unique minimal universe name \( \text{att}_i _* \) in the attribute lattice \( L _ { \text{att}_i} \) that covers \( C \), and convert the term to \( \Lambda t :: (\text{att}_1 _* , \ldots , \text{att}_n _* ).M \).

(4) **Universe-name-based attribute-passing optimization.**

After the above transformation, type abstraction is of the form \( \Lambda t :: (\text{att}_1 _* , \ldots , \text{att}_n _* ) \). The conventional type-attribute-passing compilation phase converts a polymorphic function of the form \( \lambda x : t . M_0 \) where \( M_0 \) requires \( \text{att}_i (t) \) to \( \lambda A : \text{attTy}(t) . \lambda x : t . M_0 \), and the corresponding type application \( M \tau \) to \( M \tau \cdot v_i \) if \( \text{attTerm} (\tau) = v_i \) for some value \( v_i \), otherwise \( M \tau \cdot A \) to propagate the extra type attribute parameter \( A \). This phase is refined so that \( \lambda A : \text{attTy}(t) _i \) is inserted only when \( \text{att}_i _* \) is \( \text{att}_i _{\text{unknown}} \). In other cases, attribute value \( \text{attTerm}(t) _i \) statically evaluates to \( v_i \), and type attribute variable \( A : \text{attTy}(t) _i \) will never be referenced.

We claim that the optimization presented in this section achieves the optimal result in attribute abstraction and application for closed and predicative programs in the sense that it only inserts attribute abstraction if the target function is applied to more than one type that has different attribute values. One limitation of this method is that it does not perform optimization of external functions. This is related to the issue of generalizing our method to separate compilation, which we shall discuss later.

7 EXTENSION TO SEPARATE COMPILATION

The formalism and our optimization method presented in previous sections do not address the issue of separate compilation. The calculus with finitary polymorphism defined in Section 2 and the universe reconstruction algorithm presented in Section 5 assume that the set of all the call sites of a polymorphic function are available. This assumption no longer hold in separate compilation.
In our implementation described later, we mark type abstractions of polymorphic functions that are exported to other compilation units as EXTRN, whose universe name is always `att_unknown` for any attribute `att`.

In this section, we present an extension of our method to separate compilation.

### 7.1 Optimizing a Separate Compilation System

In a separate compilation system, such as the one implemented in SML#, each source file `S` is associated with an interface file `I` consisting of the following declarations:

- **Require interfaces** \( \{ I_1, \ldots, I_n \} \), which specifies the set of interface files of other compilation units that are used in the source file `S`.
- **Provide typings** \( \{ x_1 : \sigma_1, \ldots, x_m : \sigma_m \} \), which specifies the set of identifiers with their types that are provided to other compilation units.

Using these interface declarations, separate compilation is done in the following steps:

1. Statically evaluate all the require interfaces and construct a static environment.
2. Statically evaluate the source file under the constructed static environment to obtain the static typing of the set of identifiers to be exported, and checks that the set of typings conforms to the provide typings.
3. If the above static evaluations are successful, generate the code of the source program.

To extend our optimization method to this form of a separate compilation system, we extend the interface file `I` of a source file `S` with the following additional declarations:

- **Usedby interfaces** \( \{ I'_1, \ldots, I'_k \} \), which specifies the set of interface files of other compilation units that require `S`.
- **Require universes** \( \{ x_1 : \forall t_1 :: C_1, \ldots, x_n : \forall t_n :: C_n \} \), which specifies, for each polymorphic function declaration \( x_i : \forall t_i, \sigma_i \) in a required interface \( I_j \), the upper bound \( C_i \) of type abstraction in the type of \( x_i \).
- **Provide universes** \( \{ x_1 : \forall t_1 :: C_1, \ldots, x_m : \forall t_m :: C_m \} \), which specifies, for each polymorphic function declaration \( x_i : \forall t_i, \sigma_i \) in the provide typings, the upper bound \( C_i \) of type abstraction in the type of \( x_i \).

With these additional declarations, our optimization method is incorporated in a separate compilation system by adding the following phases after the static elaboration phases (step (1) and (2)) of the above separate compilation process:

1. For each exported identifier, compute the union of the set of require universes declared in all the usedby interfaces, and used this universe as the initial universe in the universe reconstruction algorithm.
2. When the universe reconstruction is completed, the compiler checks that the computed universe is a subset of the declared provide universe for each provide identifier.
3. After these steps, the optimization algorithm described in Section 6 is applied.

The first two steps also verify the consistency of require universe and provide universe declarations.

There is no fundamental difficulty for the programmer to specify their conservative estimate. Specifying the set of usedby interfaces can be regarded as a part of system design. Require universes and provide universes are local to the program `S` and therefore the programmer should be able to estimate them. Admittedly however, it is cumbersome to hand specify these declarations. Another possibility is to generate them automatically by the compiler when the set of all the source files have been written. This scenario is also feasible for separate compilation. Since these additional declarations are for optimization, each source file can still be separately type-checked and compiled.
without these declaration. When additional performance is desired, then the developer asks the compiler to generate these declarations and re-compiles the entire source files.

Developing a user-friendly yet light-weight and effective method is a subject of further investigation.

7.2 Providing Specialized Polymorphic Library Functions

While the above extension should be effective for separately compiled set of source programs, it may not be practical for general purpose library such as List and Array structures in the Standard ML Basis Library. The functions in those libraries are heavily used in most of programs. One ad-hoc solution to these polymorphic library functions is to provide specialized copies of commonly used instance types. For example, for a commonly used library function map : \( \forall t. \forall s. (t \to s) \to t \ list \to s \ list \), the compiler can provide the following three specialized copies:

- \( \text{map\_boxed} : \forall t :: (\langle \text{tag\_boxed} \rangle). \forall s :: (\langle \text{tag\_boxed} \rangle)(t \to s) \to t \ list \to s \ list \)
- \( \text{map\_4} : \forall t :: (\langle \text{tag\_unboxed}, \text{size\_4} \rangle). \forall s :: (\langle \text{tag\_unboxed}, \text{size\_4} \rangle).(t \to s) \to t \ list \to s \ list \)
- \( \text{map\_8} : \forall t :: (\langle \text{tag\_unboxed}, \text{size\_8} \rangle). \forall s :: (\langle \text{tag\_unboxed}, \text{size\_8} \rangle).(t \to s) \to t \ list \to s \ list \)

on the assumption that these three are commonly used special cases. At the call site, an appropriate version is chosen according to the reconstructed universe annotation. Our universe reconstruction based on finitary polymorphism enables us to choose a specialized version even within a polymorphic function. For example, if map is used inside of a polymorphic function as in

```haskell
fun foo L = map (fn x => (x,x)) L
```

and our universe reconstruction algorithm infers the type \( \forall r :: \{\text{string}, \text{int} \ast \text{int}\}. r \ list \to r \ list \), then map\_boxed is safely chosen.

The approach looks similar to an ad-hoc approach of providing monomorphic libraries such as those for floating point arrays adopted in various ML compilers, and MLton-style polymorphism elimination by duplication. It should be noted the following important differences. First, providing monomorphic version does not solve the problem of natural data representation. Second, in the conventional ad-hoc approach, monomorphic versions are only applicable to monomorphic types, and cannot be used inside of polymorphic code even when it is only applied to monomorphic types.

Different from MLton-style complete code duplication, our specialized version are optional one for optimization. It is always possible to use the fully polymorphic version. This has a significant implication in polymorphic functions with multiple type variables. For example, support we have a library function of type \( \forall t. \forall s. \forall r. \forall u. \forall v. t \to s \to r \to u \to v \to t \ast s \ast r \ast u \ast v \). If the polymorphism version is not available and each type variables has 3 instances then \( 3^5 = 243 \) copies of specialized version are required for separately compiling this function.

8 IMPLEMENTATION AND EVALUATION

We implemented the optimization method described in Section 6 and the library specialization described in Subsection 7.2, and evaluated the performance. Implementation of separate compilation extension described in Subsection 7.1 is our on-going work, which we would like to report in future version.

The implementation is done by extending the SML# compiler version 3.4.0 with our optimization method. The resulting optimizing SML# compiler implements three compilation modes: one without optimization, one with type attribute abstraction optimization only, and the one with type attribute abstraction optimization and the library code specialization.

In this section, we describe the implementation, discuss some implementation issues, and present the performance evaluation results.
8.1 Structure of the Target Compiler
The SML# compiler separately compiles one source file into an object file using interface files of other sources specified through _require declarations. The compilation phases relevant to the implementation are the following:

- **Kinded type reconstruction.**
  This phase performs type inference and reconstructs an explicitly typed second-order term with kind annotations indicating the need of type information to compile polymorphic constructs in later phases. These kind annotations are used in later phases to type-directed compilation of the corresponding polymorphic constructs. In most of the functions, tag and size attributes are required and hence the kind information to indicate their needs are implicit.

- **Type-directed transformation.**
  This phase translates the explicitly typed second-order terms into a low-level calculus by compiling out polymorphic constructs using type information by inserting the attribute abstractions and applications when type-attribute values are required. This phase consists of a series of compile steps for polymorphic constructs including record operations, heap block allocations, and function calls.

8.2 Implementation of the Optimization Algorithm
In this implementation, we selected the type attributes _tag, size_ (for natural data representation) and _offset(l)_ (for record polymorphism) as our target of optimization.

The implementation is done by inserting a new optimization phase after the kinded type reconstruction, and modifying the type-directed transformation phase so that it uses universe names computed by the new compilation phase. The new optimization phase performs the following three steps:

1. **Universe reconstruction.**
   This step translates the explicitly typed second-order lambda calculus obtained by the type reconstruction phase into the term of the calculus with finitary polymorphism by performing the universe reconstruction algorithm. It largely follows the algorithm described in Section 5 with the following additional refinement:
   - The algorithm assigns a fresh universe variable only to each type abstraction. The type environment _T_ in our formal development corresponds to the set of external variables imported through the _require_ interface declarations. Our current implementation does not incorporate the separate compilation extension described in Subsection 7.1; we mark them _EXTRN_ and exclude them from the universe reconstruction.
   - Re-implementation of record inference with kinded unification. The SML# compiler contains the algorithm _WK_ for record type inference with kinded unification. However, the algorithm is tightly coupled in the type reconstruction phase and we find it difficult to reuse it. Hence, we have re-implemented a simpler version for the terms of the form _# r = σ_ and _x = y_, which are the only terms generated by our algorithm.
   - Treatment of failure. The universe reconstruction algorithm fails when it detects cycles in the instance-universe relation through the occur check in the kinded unification. This indicates that we cannot construct a finite universe. For example, by exploiting functors, some rank-2 polymorphic terms as in

   \[
   \text{let } F = \lambda x : (\forall t. t \rightarrow t). \Lambda s. x \in F (F (\Lambda t. \lambda x : t.x))
   \]
may be constructed and make the universe reconstruction fail. In this case, instead of stopping translation, the algorithm proceeds by marking the type variables involving a cycle as ERROR, which are later treated in the same way as EXTRN.

- Treatment of polymorphic data constructors. Polymorphic data constructors such as `nil` in ML do not contain explicit type abstraction \( \Lambda t \), but they are also subject to type-directed compilation. So we treat each occurrence of polymorphic data constructors as a new definition of a type abstraction term.

The result of this step is an annotation of each type abstraction with one of ERROR, EXTRN, EMPTY, MONO(\( \sigma \)), or a universe bound \( C \). ERROR and EXTRN are explained above. EMPTY indicates that the type abstraction will never be instantiated. MONO(\( \sigma \)) indicates that \( \sigma \) is the only instance of this type abstraction.

(2) Universe evaluation.
In this step, the algorithm evaluates the universe bound annotation \( C \) and generates one of the following universe name annotation for \( tag \), \( size \), and \( offset(l) \) attributes:

\[
\begin{align*}
tag & : \{ tag_{\text{unknown}}, tag_{\text{boxed}}, tag_{\text{unboxed}} \} \\
size & : \{ size_{\text{unknown}}, size_1, size_2, size_4, size_8 \} \\
offset(l) & : \{ offset(l)_{\text{unknown}} \} \cup \{ offset(l)_n \mid n \in \text{int} \}
\end{align*}
\]

These universe names are elements of universe name lattices \( L_{tag} \), \( L_{size} \), and \( L_{offset(l)} \), which we have explained in Section 6.

(3) Type-attribute passing optimization.
After the previous step, each type abstraction is marked with one of the annotations: ERROR, EXTRN, EMPTY, MONO(\( \sigma \)), or a vector of universe names of the form \( \langle att_1_{\cdot v_1}, \ldots, att_n_{\cdot v_n} \rangle \). Those marked with ERROR or EXTRN are not subject to specialization. For the other three cases, the optimizer performs the following in the type-directed compilation phase:

(a) Eliminating EMPTY abstractions as dead code.
The optimizer performs a simple dead code elimination for the type abstraction marked as EMPTY. The form \( \Lambda t :: \text{EMPTY}.M \) will never be applied to any types and therefore they are dead code. Since unused terms are still bound to variables, the optimizer replaces the terms with a dummy value.

(b) Converting MONO(\( \sigma \)) abstractions to monomorphic functions.
If a type abstraction is marked with MONO(\( \sigma \)), the optimizer eliminates this type abstraction and the corresponding type applications, and substitutes the type variable with the type \( \sigma \).

(c) Type abstraction elimination and term specialization of \( \langle att_1_{\cdot v_1}, \ldots, att_n_{\cdot v_n} \rangle \) abstraction.
The remaining cases are our main target of optimization. If the reconstructed universe contain more than one elements, then for each type attribute \( att_i \), the universe evaluation annotates a universe name \( u_{v_i} \). In this case, the optimizer performs universe-name-based attribute-passing optimization described in Section 6. The type directed compilation phase of SML# is modified so that it inserts type attribute abstractions only when the corresponding universe name is \( att_{\text{unknown}} \). This change is reflected to the type of the polymorphic function, which suppresses the corresponding attribute value application at the call site. So additional elimination of the type application term is automatically done by the type-directed compilation phase of the original SML# compiler.
8.3 Implementation of Specialized Polymorphic Library Functions

We have implemented the library specialization presented in Subsection 7.2 for the Standard ML Basis Library. We have taken the following strategies:

1. Specialize all the polymorphic functions that are not optimized by the optimization algorithm described above.
2. For a function that contains multiple type abstraction, all the type variables are restricted to the same universe.
3. For each polymorphic function $F$, we make the three specialized copies $F_1$, $F_2$, and $F_3$ by specializing the type abstractions to $C_1 = \{\text{TAG} : \text{tag\_boxed}, \text{SIZE} : \text{size\_8}\}$, $C_2 = \{\text{TAG} : \text{tag\_unboxed}, \text{SIZE} : \text{size\_4}\}$, and $C_3 = \{\text{TAG} : \text{tag\_unboxed}, \text{SIZE} : \text{size\_8}\}$, respectively.

Under these strategy, the implementation is straightforwardly done by performing the following after universe reconstruction and universe evaluation:

- If the type of a polymorphic function contains type abstraction with $\{\text{TAG} : \text{tag\_unknown}, \text{SIZE} : \text{size\_unknown}\}$, then the three specialized copies with the above universe names are generated. This step is enabled by a compilation switch.
- When the polymorphic function is instantiated to some type $\tau$, then the optimizer checks whether $C \vdash \tau \in C_i$ ($i \in 1, 2, 3$) or not, and selects $F_i$ if $C \vdash \tau \in C_i$ and $F_i$ is defined, otherwise it selects the original $F$.

With this extension, we have compiled the Standard ML Basis Library with the code duplication specialization switch turned on, and obtained a compiler that compiles user source files without code duplication but uses specialized library functions whenever possible.

8.4 Performance Evaluation

Using the implementation, we evaluated the effectiveness of the method by comparing the performance of the optimized code against the unoptimized code.

For the benchmark programs, we used the standard micro benchmark set of Standard ML distributed with Standard ML compilers including SML#, the Standard ML Basis Library that is statically linked with all the benchmark programs, and the SML# compiler itself. The SML# compiler consists of 210 separately compiled Standard ML source files. We used this entire set of source files as a benchmark (referred to as "smlsharp" below), i.e. we compile the entire SML# compiler with our optimizing SML# compiler to generate three versions. The specialization statistics shown below is the sum of those of all the SML# source files. As an input data to smlsharp benchmark, we used a match compiler source file of the SML# compiler, which contains 1,665 lines of Standard ML code.

The optimizer collects the following data:

| Attribute  | Description                                      |
|------------|--------------------------------------------------|
| total      | the total number of type abstractions             |
| mono       | the number of type abstractions with a singleton universe |
| deadcode   | the number of type abstractions with the EMPTY universe |
| boxed      | the number of specializations with $\text{tag\_boxed}$ |
| unboxed    | the number of specializations with $\text{tag\_unboxed}$ |
| size       | the number of specializations with $\text{size\_n}$ for any $n$ |
| offset     | the number of specializations with $\text{offset(l)\_n}$ for any $n$ |

Table 1 shows the specialization statistics of optimization. The benchmarks "coresml", "gcbench", and "mandelbrot" do not contain any type abstractions. So we omit them in the following comparisons. Figure 9 shows the summary of the statistics. It shows for each type attribute the amount of attribute abstraction elimination relative to the expected number of attribute abstraction for the unoptimized fully polymorphic code. Since the benchmarks do not use record polymorphism,
Table 1. Statistics of the benchmark programs

| benchmark          | total | mono | deadcode | boxed | unboxed | size | offset | unchanged |
|--------------------|-------|------|----------|-------|---------|------|--------|-----------|
| barnes_hut2        | 21    | 19   | 0        | 0     | 1       | 1    | 0      | 0         |
| boyer              | 23    | 23   | 0        | 0     | 0       | 0    | 0      | 0         |
| count_graphs       | 36    | 26   | 4        | 0     | 2       | 0    | 0      | 0         |
| cpstk              | 3     | 3    | 0        | 0     | 0       | 0    | 0      | 0         |
| diviter            | 8     | 5    | 0        | 1     | 0       | 1    | 0      | 2         |
| divrec             | 7     | 4    | 0        | 1     | 0       | 0    | 0      | 2         |
| fft                | 1     | 0    | 1        | 0     | 0       | 0    | 0      | 0         |
| gcbench            | 0     | 0    | 0        | 0     | 0       | 0    | 0      | 0         |
| knuth_bendix       | 64    | 48   | 6        | 6     | 0       | 0    | 0      | 4         |
| lexgen             | 27    | 14   | 1        | 1     | 0       | 1    | 0      | 11        |
| life               | 25    | 23   | 0        | 1     | 0       | 1    | 0      | 1         |
| logic              | 3     | 3    | 0        | 0     | 0       | 0    | 0      | 0         |
| mandelbrot         | 0     | 0    | 0        | 0     | 0       | 0    | 0      | 0         |
| nqueen             | 2     | 2    | 0        | 0     | 0       | 0    | 0      | 0         |
| nucleic            | 1111  | 995  | 116      | 0     | 0       | 0    | 0      | 0         |
| perm9              | 7     | 7    | 0        | 0     | 0       | 0    | 0      | 0         |
| puzzle             | 4     | 1    | 0        | 1     | 2       | 3    | 0      | 0         |
| ratio_regions      | 20    | 17   | 0        | 0     | 3       | 3    | 0      | 0         |
| ray                | 12    | 11   | 1        | 0     | 0       | 0    | 0      | 0         |
| simple2            | 62    | 50   | 6        | 0     | 2       | 0    | 0      | 4         |
| smllyacc           | 227   | 149  | 3        | 15    | 8       | 23   | 0      | 52        |
| tsp                | 4     | 4    | 0        | 0     | 0       | 0    | 0      | 0         |
| vliw               | 126   | 82   | 10       | 11    | 1       | 13   | 0      | 21        |
| smlsharp           | 4324  | 3373 | 139      | 158   | 9       | 162  | 0      | 645       |
| basis              | 488   | 148  | 5        | 15    | 0       | 15   | 0      | 320       |

**total**: the total number of type abstractions; **mono**: the number of singleton universes; **deadcode**: the number of EMPTY universe; **boxed**: the number of specializations with tag_boxed; **unboxed**: the number of specializations with tag_unboxed; **size**: the number of specializations with size_n for any n; **offset**: the number of specializations with offset(l)_n for any n.

the offset abstraction is expected to be always zero. So the expected number of type attribute abstractions for each type abstraction is 2 for \{tag, size\}.

From these data, we conclude that 83.79% of attribute abstractions are eliminated in total. This is quite satisfactory results. Among them, 75.81% is by monomorphic specialization (MONO(\(\sigma\))), 4.42% is by dead code elimination (EMPTY), 1.80% is by tag attribute specialization (tag_boxed and tag_unboxed), and 1.76% is by size attribute specialization (size_n).

To verify that optimization with offset(l)_n universe names works as expected, we have tested several small programs that use record polymorphism and have checked offset attribute abstractions are eliminated.

Figure 11 shows code size reduction (measured by the size of compiled object files). They show that, on average, the optimization achieves 12.04% code size reduction.

To evaluate the execution speed improvement, for each benchmark, we used the three compilation mode of our optimizing compiler described above and prepared the unoptimized code (original) the code with type attribute abstraction optimization (polytyelim), and the code with type attribute abstraction optimization and the library code specialization (polytyelim + codedup). We then measured the execution time of each code 5 times and take average. Figure 10 shows
the measurement results with error bars. They show that, on average, the optimization achieves 10.35% speed-up without library specialization, and 15.28% speed-up with library specialization. The average code size reduction (without code duplication) is 12.04%. Table 2 shows the code size increase of the Basis Library owing to code duplication specialization. It shows that the size of the entire Basis Library becomes 1.86 times larger. Since polymorphic library specialization makes 3 extra copies for each unoptimized polymorphic function, we think this overhead is reasonable.

These results confirm the effectiveness of our optimization method. In particular, 83.79% reduction of extra type attribute abstractions is significant and satisfactory. We also note that effectiveness of our method is verified not only by small benchmarks but also by a large practical system, the SML# compiler.
Fig. 11. Object size reduction

Table 2. Code duplication overhead of the Basis Library

|                        | original | polytyelim | polytyelim+codedup |
|------------------------|----------|------------|-------------------|
| Basis Library (in byte)| 2321448  | 2265968    | 3818664           |
| Basis Library (in ratio)| 1       | 0.976      | 1.645             |

9 RELATED WORKS

One of the technical contributions of the present paper is the development of finitary polymorphism as a second-order calculus where type abstraction has an explicit universe constraint. Owing to this constraint, our calculus is predicative, and it shares some properties with the predicative variant of the second-order lambda calculus [Leivant 1989, 1991]. To define a predicative type system, these systems assume a predefined hierarchy of type universes in the spirit of Russell’s ramified theory of types [Russell 1908] and Martin-Löf’s predicative intuitionistic type theory [Martin-Löf 1975]. A distinguishing feature of our type system is that every type abstraction is finitary (as an operation on types). Predicativity is achieved by finite type universe terms and the typing rule for membership relation. We believe that this is the first successful attempt to define a predicative second-order calculus without assuming a predefined hierarchy of type universes.

The contention behind our development is that any practical polymorphism is essentially finitary and therefore a calculus that represents finitary polymorphism can be used to represent the exact property of polymorphic functions. On representing finitary properties of polymorphic functions, our type system has some similarity to type systems with intersection types [Barbanera et al. 1995; Coppo et al. 1981], including recent works of non-idempotent intersection types [Bucciarelli et al. 2017]. Precise relationship between our finitary polymorphism based on second-order type abstraction and intersection types would be an interesting topic of further investigation. Our current observation is that our type system with finitary type abstraction can directly represent type-directed compilation of polymorphic programs including those that involve nested type abstractions and partial type applications.
On static analysis of polymorphic functions, our approach also has some similarity with flow analysis of the lambda calculus Shivers [1991]. Some authors have investigated optimization of ML-style implicit polymorphism by combining intersection types and flow analysis [PALSBERG and PAVLOPOULOU 2001; Siskind 1999; WELLS et al. 2002]. As we noted in Introduction, our purpose is to develop a formalism and optimization method for type-directed compilation, which realizes various advanced features over conventional ML polymorphism. Since type-directed compilation relies on explicit type attribute passing, those approaches based on flow analysis with intersection types do not seem to be directly applicable. However, it would be an interesting question whether or not universe reconstruction to second-order finitary polymorphism can be reduced to a form of flow analysis with intersection types.

Our motivation of developing a calculus of finitary polymorphism is to optimize type-directed computation based on this property. We have targeted compilation of record polymorphism [Ohori 1995] and natural data representation [Nguyen and Ohori 2006] implemented in SML# compiler. In addition, type-directed compilation has been extensively investigated. Tag-free garbage collection [Tolmach 1994] is a basic technique underlying natural data representation. Intentional type analysis [Harper and Morrisett 1995] can be regarded as a uniform generalization of record compilation. Type classes in Haskell is another important feature realized by type-directed compilation. In the simplest form, implementation of type classes in Haskell based on “dictionary passing” [Jones 1994] has the structure similar to record compilation. A number of advanced features of type classes require more elaborate attribute passing. Our method should also be effective for those type-directed compilation.

Optimizing polymorphism by specialization is a classical research topic and several approaches have been presented. In addition to Weeks [2006] of MLton mentioned in Introduction, Hall et al. [1996]; Jones [1995]; Peterson and Jones [1993]; Tolmach and Oliva [1998] also optimize polymorphism through specialization. The new technical contribution original to our work is to provide a systematic solution to this classical problem based on a framework of finitary polymorphism. We have shown that our method provides an optimal solution to this problem in the sense that lambda abstraction for extra attribute passing is only inserted when the function is applied to more than two types that differ in the values of attributes. In a wider perspective, our work is related to representation optimization of polymorphic languages including polymorphism with unboxed values [Jones and Launchbury 1991; Leroy 1992], and flexible representation analysis [Shao 1997]. We believe that our calculus can be applied to these optimization methods to perform more accurate analysis and therefore to obtain more effective results.

10 CONCLUSIONS AND FURTHER INVESTIGATIONS

To develop a type-theoretical method for optimizing type-directed compilation of polymorphic languages, we defined a predicative second-order lambda calculus with finitary polymorphism, where each type abstraction is explicitly constrained to a finite type universe, and showed the type soundness with respect to type-passing operational semantics. Different from a calculus with stratified type universes, type universes of the calculus are terms that represent the exact finite set of instance types. For this calculus, we developed a universe reconstruction algorithm that takes a term of the standard second-order lambda calculus, checks if the term is typable with finitary polymorphism, and, if typable, constructs a term in the calculus of finitary polymorphism. Based on these results, we presented a type-based optimization method for polymorphic languages with several advanced features that require type-passing operational semantics, and implemented the method in the SML# compiler, and evaluated its effectiveness. The performance results show that the method eliminates 83.79% of type abstractions, and the execution speed of the optimized code.

Proc. ACM Program. Lang., Vol. 2, No. ICFP, Article 81. Publication date: September 2018.
Atsushi Ohori, Katsuhiro Ueno, and Hisayuki Mima

increases by 15.28% on average. These results demonstrates the significance and effectiveness of our method.

This is the first step toward establishing a type-theoretical optimizing method for type-directed compilation of polymorphic languages, and there are a number of further investigations. An important future topic is to extend our implementation to separate compilation. To implement the optimization method for separate compilation system presented in Subsection 7.1, we need to sort out various engineering issues on how to specify additional interface declarations, especially the usedby declarations and require universes. For this purpose, we should develop some tools and programming environments that assist the programmer to specify those declarations.

ACKNOWLEDGMENTS

The authors thank anonymous reviewers for a number of constructive comments, which were useful for improving the presentation of the paper. The first author’s work was partially supported by JSPS KAKENHI Grant Numbers JP25280019 and JP18K11233. The second author’s work was partially supported by JSPS KAKENHI Grant Number JP15K15964.

REFERENCES

F. Barbanera, M. Dezani-Ciancaglini, and U. Deliguoro. 1995. Intersection and Union Types: Syntax and Semantics. Information and Computation 119, 2 (1995), 202 – 230. https://doi.org/10.1006/inco.1995.1086

Antonio Bucciarelli, Delia Kesner, and Daniel Ventura. 2017. Non-idempotent intersection types for the Lambda-Calculus. Logic Journal of the IGPL 25, 4 (2017), 431–464. https://doi.org/10.1093/jigpal/jzw018

Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. 1981. Functional Characters of Solvable Terms. Math. Log. Q. 27, 2-6 (1981), 45–58. https://doi.org/10.1002/malq.19810270205

Martin Elsmar. 1998. Polymorphic equality – No tags required. In Types in Compilation, Xavier Leroy and Atsushi Ohori (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 136–155.

Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. 1996. Type Classes in Haskell. ACM Trans. Program. Lang. Syst. 18, 2 (March 1996), 109–138. https://doi.org/10.1145/227699.227700

Robert Harper and Greg Morrisett. 1995. Compiling Polymorphism Using Intensional Type Analysis. In Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL ’95). ACM, New York, NY, USA, 130–141. https://doi.org/10.1145/199448.199475

Mark P. Jones. 1994. A Theory of Qualified Types. Sci. Comput. Program. 22, 3 (1994), 231–256. https://doi.org/10.1016/0167-6423(94)90005-0

Mark P. Jones. 1995. Dictionary-free overloading by partial evaluation. LISP and Symbolic Computation 8, 3 (01 Sep 1995), 229–248.

Simon L. Peyton Jones and John Launchbury. 1991. Unboxed values as first class citizens in a non-strict functional language. In Functional Programming Languages and Computer Architecture, John Hughes (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 636–666.

Daniel Leivant. 1989. Stratified Polymorphism (Extended Summary). In Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS ’89), Pacific Grove, California, USA, June 5-8, 1989. IEEE, 39–47. https://doi.org/10.1109/LICS.1989.39157

Daniel Leivant. 1991. Finitely stratified polymorphism. Information and Computation 93, 1 (1991), 93 – 113. Selections from 1989 IEEE Symposium on Logic in Computer Science.

Xavier Leroy. 1992. Unboxed Objects and Polymorphic Typing. In Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL ’92). ACM, New York, NY, USA, 177–188. https://doi.org/10.1145/143165.143205

Martin-Löf. 1975. An Intuitionistic Theory of Types: Predicative Part. In Logic Colloquium, H. E. Rose and J. C. Shepherdson (Eds.). North Holland, Amsterdam.

Huu-Duc Nguyen and Atsushi Ohori. 2006. Compiling ML Polymorphism with Explicit Layout Bitmap. In Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP ’06). ACM, New York, NY, USA, 237–248. https://doi.org/10.1145/1140335.1140364

Atsushi Ohori. 1992. A Compilation Method for ML-style Polymorphic Record Calculi. In Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL ’92). ACM, New York, NY, USA, 154–165. https://doi.org/10.1145/143165.143200

Proc. ACM Program. Lang., Vol. 2, No. ICFP, Article 81. Publication date: September 2018.
