Type System for Specializing Polymorphism[⋆]

Atsushi Ohori[⋆⋆]

Research Institute for Mathematical Sciences
Kyoto University
Sakyo-ku, Kyoto 606 JAPAN
E-Mail: ohori@kurims.kyoto-u.ac.jp

Abstract. Flexibility of programming and efficiency of program execution are two important features of a programming language. Unfortunately, however, there is an inherent conflict between these features in design and implementation of a modern statically typed programming language. Flexibility is achieved by high-degree of *polymorphism*, which is based on generic primitives in an abstract model of computation, while efficiency requires optimal use of low-level primitives specialized to individual data structures. The motivation of this work is to reconcile these two features by developing a mechanism for specializing polymorphic primitives based on static type information. We first present a method for *coherent* transformation of an ML style polymorphic language into an explicitly typed calculus. We then analyze the existing methods for compiling record calculus and unboxed calculus, extract their common structure, and develop a framework for type based specialization of polymorphism.

1 Introduction

The term "polymorphism" means *generic nature* of a program. A program is polymorphic if it behaves uniformly over values of various different types. A typical example is the identity function

$$id \equiv \lambda x.x$$

which has the same behavior for all types and can therefore be applied to values of any type. Another example is field selection from a labeled record

$$f \equiv \lambda x.x{\cdot}l$$

which extracts the l field from a labeled record (passed through an argument x.) This function behaves uniformly for any record type containing an l field.

[⋆] **Appeaered in Proc. Theoretical Aspect of Computer Software, LNCS 1281, pages 107 - 137, 1997 as an invited paper.**

[⋆⋆] Partly supported by the Japanese Ministry of Education Grant-in-Aid for Scientific Research on Priority Area 275: "advanced databases".

An important achievement in type theory of programming languages is the development of polymorphic type systems where generic nature of a program such as above is cleanly represented as a term having a polymorphic type. In Girard-Reynolds type system [7, 29], the function *id* is given the following type

$$id : \forall t.t \to t$$

representing the polymorphic nature of *id*. This form of polymorphism is embodied in the type system of ML [17]. Its practical usefulness has been widely recognized and an ML-style polymorphic type system has been adopted in a number of modern polymorphic programming languages including Standard ML [18] and Haskell [12]. Recent studies of *record polymorphism* enable us to represent polymorphic functions operating on labeled record structures. Using the type system of [25], the field selection function f above is given the following polymorphic type

$$f \ : \ \forall t_1{::}U.\forall t_2{::}\{\!\{l : t_1\}\!\}.\, t_2 \to t_1$$

where type variables t_1 and t_2 are constrained with a *kind*, i.e., a type of types. $t_1{::}U$ means that t_1 ranges over all types, while $t_2{::}\{\!\{l : t_1\}\!\}$ denotes the constraint that t_2 ranges only over those record types containing an l field of type t_1. For example, if t_1 is instantiated to *string* then t_2 can be instantiated to $\{l : string, m : int\}$ but not $\{a : int, b : bool\}$. By this kind constraint, the above type correctly represents the polymorphic nature of the function f above, i.e., f takes a value of any record type containing an l field and returns a value of that field. We call the typing mechanism refined with kinds *kinded typing.* Due to this refinement, ML-style parametric polymorphism can scale up to large and complicated practical software development such as database programming, where labeled data structures are essential [3].

It is highly desirable to develop an efficient programming language that supports flexible polymorphic typings. However, polymorphism inherently conflicts with run-time efficiency of programs – another essential feature of programming languages. The source of polymorphism is the generic nature of primitive operations in an abstract computation model on which a high-level programming language is based. On the other hand, run-time efficiency is achieved through optimal use of low-level primitives specialized to individual data structures. To achieve run-time efficiency of a program, it is essential to compile high-level primitive operations into optimized low-level code. This is a straightforward process for a statically typed monomorphic language. Since the data structure is statically determined, the compiler can compile high-level generic primitives into specialized low-level code depending on the argument type determined by the context in which it is used. A language with a polymorphic type system, however, this is a difficult problem.

To understand the problem let us analyze the two examples of polymorphic functions given above. The function $\lambda x.x$ is polymorphic because the variable binding mechanism is inherently generic. There are several ways of describing lambda binding mechanism. In the lambda calculus, it is represented by the following rewrite rule

$$(\lambda x.e_1)\ e_2 \Longrightarrow [e_2/x]e_1$$

where $[e_2/x]e_1$ denotes the term obtained from e_1 by substituting e_2 for all free occurrences of x. The implicit assumption underlying this operation is that any values denotable by terms have the same size and therefore can be freely substituted for a variable irrespective of their types. A more machine-oriented presentation of variable binding is the mechanism of de Bruijn indexes [5], which correspond to indexes into an array of values. Again, it is assumed that all denotable values have the same size so that they can be stored in an array. In an actual computer hardware, however, values have various different sizes, and therefore generic variable binding is impossible. A similar situation exists for record polymorphism. The function $\lambda x.\ x{\cdot}l$ is polymorphic because the field selection operation $e{\cdot}l$ is a generic operation in the underlying operational semantics, which is the following.

$$\{l_1 = e_1, \ldots, l_i = e_i, \ldots, l_n = e_n\}{\cdot}l_i \implies e_i$$

However, no currently available general purpose computer architecture efficiently supports such operation.

In a statically typed monomorphic language, this problem is solved by *statically specializing* a program. The compiler generates optimized code specialized to the type of the actual data. For example, a monomorphic identity function such as

$$\lambda x : real.x$$

is compiled to the code that receives a double word floating point data and returns the same data. Similarly, in a language with simple static type system, records are represented as vectors, and a monomorphic field selecting function such as

$$\lambda x : \{Name : string, Age : int\}.x{\cdot}Name$$

is compiled to efficient code that performs indexing into a vector consisting of a string and an integer. These specializations are source of efficiency of statically typed programming languages.

Unfortunately, these apparently effective optimizations are not directly applicable to polymorphic languages. Since the type of the data actually passed to a polymorphic function differs and cannot be determined at the time of compiling the function, it is difficult to specialize it statically. The conventional solution to this problem is simply to give up those optimizations, and to implement polymorphic operations directly using far less efficient data representation and costly run-time type-checking. There have been some efforts to reduce the run-time cost due to inefficiency of data representation required by polymorphic primitives, including representation optimization for polymorphic bindings [27, 16] and efficient data representation for associative access for labeled records [28]. These methods show some positive results, but polymorphic primitives are still inefficient compared to the corresponding specialized monomorphic operations.

To develop an efficient polymorphic language suitable for serious software development, we should overcome this problem and develop a systematic method to specialize polymorphic operations into efficient code. Such a method should be a refinement of conventional techniques of compiling monomorphic languages so that monomorphic programs should be as efficient as conventional implementation of monomorphic languages. In our previous works, we have developed two such methods. One is a compilation method for a polymorphic record calculus [23, 25], which specializes polymorphic operations on labeled records and labeled variants to efficient index operations. The other is an unboxed semantics for ML polymorphism [24], which specializes polymorphic variable binding to efficient size sensitive variable binding. These two methods appear to exhibit common structures in specialization of ML-style polymorphic languages. In particular, both exploit type information to perform appropriate specialization. The purpose of the present work is to analyze these two methods, to identify the crucial issues in program specialization, and to develop a framework for type based specialization of polymorphism. We hope that the framework presented here will serve as a type-theoretical basis for compiling polymorphic functions into efficient code.

The work reported here is still preliminary. In particular, although we are confident that our framework can be used in practice, this should be justified through full-scale compiler construction. We are currently developing a rigorous operational semantics suitable for actual implementation and are planing to perform experimentation in a compiler construction project we have been conducting jointly with Oki Electric. The author intends to report a fuller account of the framework with implementation techniques in near future.

The rest of this paper is organized as follows. In Sect. 2, we describe the general structure of type based specialization of polymorphism. Sect. 3 defines the skeleton structure of the source language. The first step of specializing an ML style polymorphic language is to recover type information that represents the polymorphic nature of a given program. There is one subtle problem in this process. That is the problem of "coherence" in type reconstruction. Sect. 4 analyzes this problem and provides a solution. In Sect. 5, we analyze the two specialization methods mentioned above and extract the general structure for type based specialization. Sect. 6 gives a framework for specialization. We first define an implementation calculus where specialization of generic primitives can be represented as terms. We then develop a translation scheme from the source calculus into the implementation calculus. This development is parameterized with polymorphic primitives so that it can be applied to various polymorphic primitives. Sect. 7 describes implementation strategy for the implementation calculus. Finally, Sect. 8 concludes the paper with suggestions of future work.

2 General Structure of Specialization

The source language we are interested in is an ML-style polymorphic programming language. Polymorphism in such a language is based on *implicit typing.* The programmer writes code without any type annotation, and the type system automatically *infers* its most general type that represents the generic nature of the program.

In conventional implementation of ML-style programming languages such as [1, 15], type inference is done only for static check of type consistency; after type-checking is done, the type information is thrown away and compilation is done based on the syntactic structure of raw terms (with few exceptions of special primitives such as polymorphic equality.) However, the approach of type based specialization we are advocating makes crucial use of type information in compiling programs. There are two natural approaches to exploit type information. One is to refine type inference algorithm directly so that it not only infers types but also performs specialization as well. The other approach is to use an intermediate language that encodes all the type information obtained by type inference process. The second approach yields a cleaner and more manageable system by separating specialization from type inference. So we adopt the second approach. The type based specialization is therefore structured into the following two phases.

1. Type Reconstruction.
   This phase performs type inference on a given raw term, and constructs an explicitly typed term that encodes all the inferred type information. This phase can be regarded as the combination of type inference of an implicitly typed source language, denoted here by λ^{ml}, and transformation of λ^{ml} into an explicitly typed calculus, denoted here by Λ^{ml}.
2. Specialization.
   This phase transforms a term of Λ^{ml} into a term of an implementation calculus, which we call Λ^{impl}, by specializing polymorphic primitives using type information. Λ^{impl} is a model of a low-level language that can be implemented efficiently without using type information at run-time. The implicit calculus λ^{impl} obtained from Λ^{impl} by forgetting type information models efficient rum-time execution.

Our goal is to provide a framework for both of the phases so that it can be used for various polymorphic primitives. Fig. 1 shows the relationship between the calculi considered in this paper.

$$\begin{tikzcd}
\begin{array}{c}\Lambda^{impl}\\ \text{Implementation Calculus}\end{array} \arrow[d, "\text{(type erase)}"] & \begin{array}{c}\Lambda^{ml}\\ \text{Explicitly Typed Calculus}\end{array} \arrow[l, "\text{(specialize)}"'] \\
\begin{array}{c}\lambda^{impl}\\ \text{Execution Model}\end{array} & \begin{array}{c}\lambda^{ml}\\ \text{Source Calculus}\end{array} \arrow[u, "\text{(type encoding)}"']
\end{tikzcd}$$

Fig. 1. Relationship Among the Calculi.

Although a type inference problem with various advanced polymorphic primitives is often a difficult and delicate problem, this process does not affect the subsequent specialization. On the other hand, a proper definition of Λ^{ml} and reconstruction of a Λ^{ml} term from a given λ^{ml} typing is crucial in type based specialization. So we do not consider the type inference problem of λ^{ml} and simply assume that a typing derivation of λ^{ml} is given by some type inference process, but we include a careful analysis on Λ^{ml} and reconstruction of Λ^{ml} terms from λ^{ml} typing derivations.

3 The Source Language

The source calculus, λ^{ml}, is to serve as a model for various polymorphic languages. This calculus should be regarded as a family of calculi parameterized by various polymorphic constructs. In this section, we only consider the following Core ML terms.

$$e ::= c^b \,|\, x \,|\, \lambda x.e \,|\, e\ e \,|\, let\ x = e\ in\ e$$

c^b stands for constants of base type b, and $let\ x = e_1\ in\ e_2$ is ML's polymorphic let construct. The actual source language is obtained by adding the desired terms constructors including those that represent polymorphic primitives. The type system of λ^{ml} defined in this section is general enough to represent various polymorphic primitives. Sect. 5 demonstrates that both the record calculus and the unboxed calculus can be represented as a special case of λ^{ml}.

Terms are considered module renaming of bound variables. In what follows, we adopt the usual "bound variable convention", i.e., we assume that bound variables are distinct and are different from any free variables, and that this property is preserved by substitution. We shall implicitly make this assumption for all the languages containing variable binding mechanisms we shall consider in this paper, including polymorphic types.

A type system for Core ML was given by Damas and Milner [4]. In their account, the set of types is stratified into the set of *monotypes* (ranged over by τ) and the set of *polytypes* (ranged over by σ) as follows.

$$\tau = b \,|\, t \,|\, \tau \to \tau$$
$$\sigma = \tau \,|\, \forall t.\sigma$$

We write $[\tau_1/t_1, \ldots, \tau_n/t_n]\tau$ for the type obtained from τ by substituting τ_i for each occurrence of t_i in τ. This operation is extended to other objects containing types. The set of free type variables of τ is denoted by $FTV(\tau)$. In [4], the type system is given as a proof system to derive a typing of the form

$$\mathcal{T} \vdash e : \sigma$$

where $\mathcal{T}$ is a *type assignment*, which is a mapping from a finite set of variables to polytypes. We write $\mathcal{T}\{x : \sigma\}$ for the type assignment $\mathcal{T}'$ such that $dom(\mathcal{T}') = dom(\mathcal{T}) \cup \{x\}$, $\mathcal{T}'(x) = \sigma$, and for $y \in dom(\mathcal{T}), y \neq x$, $\mathcal{T}'(y) = \mathcal{T}(y)$. The set of typing rules is given in Fig. 2.

$$(\textsc{var})\quad \mathcal{T}\{x:\sigma\} \vdash x : \sigma \qquad\qquad (\textsc{const})\quad \mathcal{T} \vdash c^b : b$$

$$(\textsc{app})\quad \frac{\mathcal{T} \vdash e_1 : \tau_1 \to \tau_2 \quad \mathcal{T} \vdash e_2 : \tau_1}{\mathcal{T} \vdash e_1\ e_2 : \tau_2} \qquad\qquad (\textsc{abs})\quad \frac{\mathcal{T}\{x:\tau_1\} \vdash e_1 : \tau_2}{\mathcal{T} \vdash \lambda x.e_1 : \tau_1 \to \tau_2}$$

$$(\textsc{gen})\quad \frac{\mathcal{T} \vdash e : \sigma}{\mathcal{T} \vdash e : \forall t.\sigma} \quad \text{if } t \notin FTV(\mathcal{T}) \qquad\qquad (\textsc{inst})\quad \frac{\mathcal{T} \vdash e : \forall t.\sigma}{\mathcal{T} \vdash e : [\tau/t](\sigma)}$$

$$(\textsc{let})\quad \frac{\mathcal{T} \vdash e_1 : \sigma \quad \mathcal{T}\{x:\sigma\} \vdash e_2 : \tau}{\mathcal{T} \vdash let\ x = e_1\ in\ e_2 : \tau}$$

Fig. 2. Damas-Milner Type System of Core ML

In this paper, we make one refinement to Damas-Milner type system by placing *kind* constraint on type variables. Intuitively, a kind denotes a set of monotypes that share some property. For example, a record kind $\{\!\{l : \tau\}\!\}$ denotes the set of all (monomorphic) record types containing the field $l : \tau$. The purpose of introducing kind constraints is twofold.

- It enables us to represent various polymorphic primitives.
- It also enables us to keep track of the set of type variables used in typing derivations. This will become crucial when we perform type based specialization.

Required kinds depend on the form of polymorphism we would like to support. Instead of assuming some particular kinds, we consider the source calculus λ^{ml} as a family of calculi parameterized by a kind structure. For this, we assume that there is a given set $Kind$ of kinds (ranged over by k). On this set, we assume the following general properties.

- It contains a special constant kind U denoting the set of all monotypes. Type variables having U correspond to those of ML.
- In addition to constant kinds, we allow a kind to contain monotypes. This generality is necessary for representing various polymorphic operations such as those found in a polymorphic record calculus.
- The set of kinds is closed under consistent conjunction. This is needed to represent a polymorphic function that invokes more than one polymorphic primitives. For example, in $\lambda x.(x \cdot l, x \cdot m)$, the type of x is kinded by a record kind $\{\!\{l : t_1, m : t_2\}\!\}$ which is a conjunction of two atomic record kinds $\{\!\{l : t_1\}\!\}$ and $\{\!\{m : t_2\}\!\}$.

Any type variables in λ^{ml} must be kinded by a *kind assignment* (ranged over by $\mathcal{K}$) which is a list of pairs of a type variable and a kind of the form

$$\{t_1{::}k_1, \ldots, t_n{::}k_n\}$$

such that $t_1, \ldots, t_n$ are pairwise distinct. In what follows, we use the notation $\langle a \rangle$ for a list of elements of the form $a_1, \ldots, a_n$ when the exact sequence is not important. For example, we sometimes write $\{\langle t{::}k \rangle\}$ for a kind assignment.

For a kind assignment $\mathcal{K}$, we write $dom(\mathcal{K})$ for the set of type variables that are assigned a kind by $\mathcal{K}$. We write $\mathcal{K}\{t{::}k\}$ for the kind assignment obtained by adding the pair $t{::}k$ to $\mathcal{K}$ provided that $t \notin dom(\mathcal{K})$, and we write $\mathcal{K}\mathcal{K}'$ for the kind assignment obtained by concatenating $\mathcal{K}$ and $\mathcal{K}'$ provided that $dom(\mathcal{K}) \cap dom(\mathcal{K}') = \emptyset$. The empty kind assignment is denoted by $\emptyset$.

We say that a type τ is *well formed under* $\mathcal{K}$, written $\mathcal{K} \vdash \tau$, if $FTV(\tau) \subseteq dom(\mathcal{K})$. We extend this relation to other structures containing types, including kinds. Since we allow a kind to contain type variables, a kind assignment must also satisfy a well-formedness condition. We write $\vdash \mathcal{K}$ to denote that kind assignment $\mathcal{K}$ is well formed. The following rules define this property.

$$\vdash \emptyset$$

$$\frac{\vdash \mathcal{K} \quad \mathcal{K} \vdash k \quad t \notin dom(\mathcal{K})}{\vdash \mathcal{K}\{t{::}k\}}$$

In what follows, we implicitly assume that kind assignments appearing various formula are well formed.

The type system we shall define depends on *kinding judgments* of the form

$$\mathcal{K} \vdash \tau :: k$$

denoting the fact that type τ has kind k under kind assignment $\mathcal{K}$. This relation of course depends on the set of kinds and their intended meanings. Here, we assume that for a given set of kinds, there is an associated kinding relation satisfying the following properties.

1. If $\mathcal{K} \vdash \tau :: k$ then $\mathcal{K} \vdash \tau$ and $\mathcal{K} \vdash k$.
2. If $\vdash \mathcal{K}\{t{::}k\}\mathcal{K}'$ and $\mathcal{K} \vdash \tau :: k$ then $\vdash \mathcal{K}([\tau/t]\mathcal{K}')$.
3. If $\mathcal{K}\{t{::}k\}\mathcal{K}' \vdash \tau_0 :: k_0$ and $\mathcal{K} \vdash \tau :: k$ then $\mathcal{K}([\tau/t]\mathcal{K}') \vdash [\tau/t]\tau_o :: [\tau/t]k_0$.

These properties are standard ones that should be satisfied by most of kinding structures.

With the introduction of kinds, a type may depends on other type variables than its own free type variables. This requires us to generalize the notion of free type variables. The set of *essentially free type variables* of τ under $\mathcal{K}$, written $EFTV(\mathcal{K}, \tau)$, is the smallest set satisfying the following conditions.

- $FTV(\tau) \subseteq EFTV(\mathcal{K}, \tau)$.
- for each $t \in EFTV(\mathcal{K}, \tau)$, if $(t{::}k) \in \mathcal{K}$ then $FTV(k) \subseteq EFTV(\mathcal{K}, \tau)$.

The set of polytypes of λ^{ml} is refined to be the set of types of the form.

$$\forall(t_1{::}k_1, \ldots, t_n{::}k_n).\tau$$

The type system of λ^{ml} is defined as a proof system to derive a typing of the form

$$\mathcal{K}, \mathcal{T} \vdash e \,:\, \tau$$

$$(\textsc{const})\quad \mathcal{K},\mathcal{T} \vdash c^b : b \quad \text{if } \mathcal{K} \vdash \mathcal{T}$$

$$(\textsc{var})\quad \frac{\begin{array}{l}\mathcal{K} \vdash \mathcal{T}\{x : \forall(t_1{::}k_1,\ldots,t_n{::}k_n).\tau\} \\ \mathcal{K} \vdash \tau_i :: [\tau_1/t_1,\ldots,\tau_{i-1}/t_{i-1}]k_i \ (1 \leq i \leq n)\end{array}}{\mathcal{K},\mathcal{T}\{x : \forall(t_1{::}k_1,\ldots,t_n{::}k_n).\tau\} \vdash x : [\tau_1/t_1,\ldots,\tau_n/t_n]\tau}$$

$$(\textsc{app})\quad \frac{\mathcal{K},\mathcal{T} \vdash e_1 : \tau_1 \to \tau_2 \quad \mathcal{K},\mathcal{T} \vdash e_2 : \tau_1}{\mathcal{K},\mathcal{T} \vdash e_1\ e_2 : \tau_2}$$

$$(\textsc{abs})\quad \frac{\mathcal{K},\mathcal{T}\{x : \tau_1\} \vdash e_1 : \tau_2}{\mathcal{K},\mathcal{T} \vdash \lambda x.e_1 : \tau_1 \to \tau_2}$$

$$(\textsc{let})\quad \frac{\mathcal{K}\{\langle t{::}k\rangle\},\mathcal{T} \vdash e_1 : \tau_1 \quad \mathcal{K},\mathcal{T}\{x : \forall(\langle t{::}k\rangle).\tau_1\} \vdash e_2 : \tau_2}{\mathcal{K},\mathcal{T} \vdash \mathit{let}\ x = e_1\ \mathit{in}\ e_2 : \tau_2}$$
$$\text{if } \langle t\rangle \cap FTV(\mathcal{T}) = \emptyset.$$

Fig. 3. The Type System of λ^{ml}

denoting the property that e has type τ under type assignment $\mathcal{T}$ and kind assignment $\mathcal{K}$. The set of typing rules is given in Fig. 3.

Typings are closed under kind respecting type substitution, as shown in the following.

Lemma 1. *If $\mathcal{K}\{t{::}k\}\mathcal{K}',\mathcal{T} \vdash e : \tau$ and $\mathcal{K} \vdash \tau_0 :: k$ then $\mathcal{K}([\tau_0/t]\mathcal{K}'),[\tau_0/t]\mathcal{T} \vdash e : [\tau_0/t]\tau$.*

Proof is by induction on e.

We say that a type variable is *free in a typing derivation* if it appears in some $\mathcal{T}$ or τ in the typing derivation tree, and if it is not discharged by a let rule. Its inductive definition can easily be given. The following property holds for free type variables.

Proposition 1. *The set of all free type variables of any typing derivation of a typing $\mathcal{K},\mathcal{T} \vdash e : \tau$ is contained in $dom(\mathcal{K})$.*

This can be proved by induction on typing derivations. It should be noted that the set of free type variables in a derivation cannot be determined by a typing. To see this, consider the following ML typing in Damas-Milner system.

$$\emptyset \vdash (\lambda x.1)\ (\lambda y.y) : \mathit{int}$$

Although no type variable appears in this typing, a typing derivation may contain free type variables in types of x and y.

To properly deal with the problem of reconstruction of explicitly typed terms, we introduce another layer on top of typings as a model of compilation units. A *declaration* (ranged over by d) is a list of pairs of a variable and a term of the form

$$\{x_1 = e_1,\ldots,x_n = e_n\}$$

such that variables are pairwise distinct. We write $d\{x = e\}$ for the extension of d with $x = e$. The typing rules for declarations are given below.

$$\vdash \emptyset : \emptyset$$

$$\frac{\vdash d : \mathcal{T} \quad \{\langle t::k\rangle\}, \mathcal{T} \vdash e \,:\, \tau}{\vdash d\{x = e\} : \mathcal{T}\{x : \forall(\langle t::t\rangle).\tau\}}$$

Note that if $\vdash d : \mathcal{T}$ then $FTV(\mathcal{T}) = \emptyset$. Using this property, we can prove the following.

Lemma 2. *If* $\vdash \{x_1 = e_1, \ldots, x_n = e_n\} : \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ *then*

$$\emptyset, \emptyset \vdash let\ x_1 = e_1\ in\ \cdots\ let\ x_n = e_n\ in\ ()\ :\ unit$$

where $()$ *is a constant of base type unit.*

From this correspondence, we see that declarations are merely shorthand for nested let expressions. However, this notion allows us to model incremental or separate compilation of ML.

4 Reconstructing Explicitly Typed Terms

As observed by Harper and Mitchell [10], a typing derivation of Core ML corresponds to a typing of an explicitly typed term. For the Damas and Milner's type system, the corresponding set fof explicitly typed terms, called XML terms, is given by the following grammar

$$M ::= c^b \mid x \mid M\ M \mid \lambda x : \tau.M \mid let\ x : \sigma = M\ in\ M \mid \Lambda t.M \mid M\ \tau$$

where $\Lambda t.M$ is type abstraction and $M\ \tau$ is type application. The type system of this explicitly typed language is obtained from that of Damas-Milner type system by replacing the untyped terms in each typing rule with the corresponding typed terms. The following is due to Harper and Mitchell [10].

Theorem 1. *There is a one-to-one correspondence between the set of ML typing derivations and the set of XML typings.*

Because of this property, Damas-Milner calculus is often regarded as “syntactic shorthands” for XML.

There is, however, one subtle problem in this intuitive view. For a given Damas-Milner typing, there are in general infinitely many typing derivations. As a result, a given Damas-Milner typing corresponds to infinitely many XML terms. Furthermore, we have shown in [22] that some typing corresponds to provably unequal XML terms, as explained below. Consider the following Damas-Milner typing.

$$\{x : \forall t.(t \to t) \to b\} \vdash (x\ (\lambda y.y))\ :\ b$$

The following XML terms all correspond to the above typing.

$$
\begin{array}{l}
\{x : \forall t.(t \to t) \to b\} \vdash ((x\ int)\ (\lambda y : int.y))\ :\ b \\
\{x : \forall t.(t \to t) \to b\} \vdash ((x\ bool)\ (\lambda y : bool.y))\ :\ b \\
\{x : \forall t.(t \to t) \to b\} \vdash ((x\ t')\ (\lambda y : t'.y))\ :\ b
\end{array}
$$

But since the terms are in normal form and therefore they are provably unequal. Using the terminology of [2], we can say that reconstruction from Damas-Milner typings to XML terms cannot be *coherent*. For analysis of typing properties of a program, this failure of coherence does not cause any problem, since the typing properties of a term are completely determined by the typing. However, this is potentially a serious problem in type based specialization, since, in specialization, all the subterms must also be properly specialized according to their types.

An essentially same example and the related observation were re-stated in [13], and the problem of coherence was discussed. The solution suggested in [13] is simply to restrict terms to be a subset of legal ML terms to avoid above failure of coherence. However, in [22] we already showed that there is a coherent type reconstruction base on a simple semantics of ML polymorphism using only monotypes. In [10], Harper and Mitchell also observed that translation of closed terms is coherent. Those results suggest that by appropriate refinement, we can achieve coherent type reconstruction for a Damas-Milner style calculus. In this section, we provide a complete solution to this problem by defining an explicitly typed second-order calculus Λ^{ml} as a refinement of Harper and Mitchell's XML, and giving a coherent transformation from λ^{ml} to Λ^{ml}.

4.1 The Explicit Calculus: Λ^{ml}

In addition to kinding, we define Λ^{ml} to have the following properties.

- Type abstraction and type application are restricted to let expressions and variables, respectively.
- Type substitution is combined with term substitution.

These properties make Λ^{ml} typings more closely correspond to λ^{ml} typing derivations.

The set of terms of Λ^{ml} is given by the following syntax.

$$M ::= c^b \mid (x\ \langle\tau\rangle) \mid M\ M \mid \lambda x : \tau.M \mid let\ x : \sigma = \Lambda(\langle t{::}k\rangle).M\ in\ M$$

To define the reduction relation for Λ^{ml}, we generalize substitution by combining it with type instantiation. We write $[(t_1, \ldots, t_n).M/x]N$ for the term obtained from N by substituting $[\tau_1/t_1, \ldots, \tau_n/t_n]M$ for $(x\ \tau_1 \ldots \tau_n)$. Its inductive definition is obtained by extending the following cases according to the structure of the term.

$$
\begin{array}{l}
[(t_1, \ldots, t_n).M/x](x\ \tau_1 \ldots \tau_n) = [\tau_1/t_1, \ldots, \tau_n/t_n]M \\
[(t_1, \ldots, t_n).M/x](y\ \tau_1 \ldots \tau_m) = (y\ \tau_1 \ldots \tau_m) \quad (\text{if } x \not\equiv y)
\end{array}
$$

We write $M \longrightarrow M'$ if M' is obtained from M by applying one of the following reduction axioms to some subterm of M.

$$(\beta) \quad (\lambda x : \tau.M_1)\ M_2 \Longrightarrow [().M_2/x]M_1$$

$$(let) \quad let\ x : \sigma = \Lambda(\langle t{::}k\rangle).M_1\ in\ M_2 \Longrightarrow [(\langle t\rangle).M_1/x]M_2$$

The reduction relation $M \longrightarrow\!\!\!\!\rightarrow M'$ is the reflexive transitive relation of the one-step reduction relation $M \longrightarrow M'$.

The kinding relation is the same as that of λ^{ml}. The set of typing rules for Λ^{ml} is given in Fig. 4.

$$(\textsc{const}) \quad \mathcal{K}, \mathcal{T} \vdash c^b : b \qquad \text{if } \mathcal{K} \vdash \mathcal{T}$$

$$(\textsc{var}) \quad \frac{\begin{array}{l}\mathcal{K} \vdash \mathcal{T}\{x : \forall(t_1{::}k_1, \ldots, t_n{::}k_n).\tau\} \\ \mathcal{K} \vdash \tau_i :: [\tau_1/t_1, \ldots, \tau_{i-1}/t_{i-1}]k_i \ (1 \leq i \leq n)\end{array}}{\mathcal{K}, \mathcal{T}\{x : \forall(t_1{::}k_1, \ldots, t_n{::}k_n).\tau\} \vdash (x\ \tau_1 \cdots \tau_n) : [\tau_1/t_1, \ldots, \tau_n/t_n]\tau}$$

$$(\textsc{app}) \quad \frac{\mathcal{K}, \mathcal{T} \vdash M_1 : \tau_1 \rightarrow \tau_2 \qquad \mathcal{K}, \mathcal{T} \vdash M_2 : \tau_1}{\mathcal{K}, \mathcal{T} \vdash M_1\ M_2 : \tau_2}$$

$$(\textsc{abs}) \quad \frac{\mathcal{K}, \mathcal{T}\{x : \tau_1\} \vdash M_1 : \tau_2}{\mathcal{K}, \mathcal{T} \vdash \lambda x : \tau_1.M_1 : \tau_1 \rightarrow \tau_2}$$

$$(\textsc{let}) \quad \frac{\mathcal{K}\{\langle t{::}k\rangle\}, \mathcal{T} \vdash M_1 : \tau_1 \qquad \mathcal{K}, \mathcal{T}\{x : \forall(\langle t{::}k\rangle).\tau_1\} \vdash M_2 : \tau_2}{\mathcal{K}, \mathcal{T} \vdash let\ x : \forall(\langle t{::}k\rangle).\tau_1 = \Lambda(\langle t{::}k\rangle).M_1\ in\ M_2 : \tau_2}$$
$$\text{if } \{\langle t\rangle\} \cap FTV(\mathcal{T}) = \emptyset$$

Fig. 4. The Type System of Λ^{ml}

For this calculus, the following two substitution lemmas hold.

Lemma 3. *If $\mathcal{K}\{t{::}k\}\mathcal{K}', \mathcal{T} \vdash M : \tau$ and $\mathcal{K} \vdash \tau_0 :: k$ then $\mathcal{K}([\tau_0/t]\mathcal{K}'), [\tau_0/t]\mathcal{T} \vdash [\tau_0/t]M : [\tau_0/t]\tau$.*

Lemma 4. *If $\mathcal{K}, \mathcal{T}\{x : \forall(\langle t{::}k\rangle).\tau_0\} \vdash M_1 : \tau$, $\mathcal{K}\{\langle t{::}k\rangle\}, \mathcal{T} \vdash M_2 :: \tau_0$, and $\langle t\rangle \cap FTV(\mathcal{T}) = \emptyset$ then $\mathcal{K}, \mathcal{T} \vdash [(\langle t\rangle).M_2/x]M_1 : \tau$.*

Both can be proved by induction on M. Using these properties, we can prove the following subject reduction theorem.

Theorem 2. *If $\mathcal{K}, \mathcal{T} \vdash M : \tau$ and $M \longrightarrow\!\!\!\!\rightarrow N$ then $\mathcal{K}, \mathcal{T} \vdash N : \tau$.*

Similarly to λ^{ml}, the set of declarations (ranged over by D) of $\Lambda^{ml}$are defined as lists of pairs of a typed variable and a term of the form

$$\{x_1 : \sigma_1 = \Lambda(\langle t_1\rangle).M_1, \ldots, x_n : \sigma_n = \Lambda(\langle t_n\rangle).M_n\}$$

such that variables are pairwise distinct. We write $D\{x : \sigma = \Lambda(\langle t\rangle).M\}$ for the extension of D with $x : \sigma = \Lambda(\langle t\rangle).M$. The typing relation for declarations is given as follows.

$$\vdash \emptyset : \emptyset$$

$$\frac{\vdash D : \mathcal{T} \quad \{\langle t{::}k\rangle\}, \mathcal{T} \vdash M \ : \ \tau}{\vdash D\{x : \forall(\langle t{::}k\rangle).\tau = \Lambda(\langle t{::}k\rangle).M\} : \mathcal{T}\{x : \forall(\langle t{::}k\rangle).\tau\}}$$

4.2 Coherent Type Reconstruction

We now show that the type reconstruction for λ^{ml}declarations is coherent.

For a Λ^{ml} term M, $erase(M)$ is the λ^{ml} term obtained from M by erasing all the type information. Its inductive definition is given below.

$$\begin{aligned}
erase(c^b) &= c^b \\
erase((x\ \langle\tau\rangle)) &= x \\
erase(\lambda x : \tau.M) &= \lambda x.erase(M) \\
erase(M_1\ M_2) &= erase(M_1)\ erase(M_2) \\
erase(let\ x : \sigma = \Lambda(\langle t{::}k\rangle).M_1\ in\ M_2) &= let\ x = erase(M_1)\ in\ erase(M_2)
\end{aligned}$$

The following lemma is crucial in establishing the coherence.

Lemma 5. *Let $\mathcal{T}$ be a type assignment that does not contain polytype, and M_1, M_2 be terms in normal form. If $\mathcal{K}_1, \mathcal{T} \vdash M_1 \ : \ \tau$, $\mathcal{K}_2, \mathcal{T} \vdash M_2 \ : \ \tau$, and $erase(M_1) \equiv erase(M_2)$ then $M_1 \equiv M_2$.*

Proof. Since $\mathcal{T}$ does not contain polytype, M_1, M_2 are both simply typed terms. Then the lemma is proved similarly to the corresponding result in [22]. □

Since Λ^{ml} can be regarded as a predicative subset of the second-order lambda calculus, Λ^{ml} is strongly normalizing. Then Theorem 2 and Lemma 5 imply the following.

Corollary 1. *If $\mathcal{T}$ does not contain polytype, $\mathcal{K}_1, \mathcal{T} \vdash M_1 \ : \ \tau$, $\mathcal{K}_2, \mathcal{T} \vdash M_2 \ : \ \tau$, and $erase(M_1) \equiv erase(M_2)$ then M_1 and M_2 are equal.*

As a special case of this, all the Λ^{ml} typings of the form $\mathcal{K}', \emptyset \vdash M \ : \ \tau$ that corresponds to the same λ^{ml} typing $\mathcal{K}, \emptyset \vdash e \ : \ \tau$ have the same meaning. This means that the translation of a complete program is guaranteed to be coherent. A naive strategy is therefore to compile a complete program at once. Unfortunately, this strategy does not support incremental compilation such as top-level interactive loop implemented in most of functional languages, since in this case a unit of compilation is necessarily an open term containing free variables having a polytype. We solve this problem by regarding a declaration as a compilation unit. Lemma 2 shows that a declaration $d\{x = e\}$ has the property that both d and $d\{x = e\}$ correspond to a closed term. Then by regarding declaration as a compilation unit, we obtain a complete solution for coherent type reconstruction. Corollary 1 guarantees that for a declaration d in λ^{ml}, all possible declarations in Λ^{ml} corresponding to d have the same meaning. Furthermore, this does not place any restriction on ML terms, and this allows incrementally compile each element of a declaration. We claim that this is a faithful model for most of ML implementations.

4.3 Canonical Λ^{ml} Terms

The previous result establishes that we can freely choose any Λ^{ml} term corresponding to a given λ^{ml} declaration. As an intermediate term for specialization we shall develop later, we need to choose a "canonical" one. Λ^{ml} terms corresponding to the same λ^{ml} declatation differ only in the type annotations, so this amounts to choosing canonical type annotations.

To define a desirable canonical type annotations, we make the following additional convention on a given kind structure.

> For any closed kind k there is a unique closed type τ_k such that $\emptyset \vdash \tau_k :: k$.

Note that this condition does not place any constraint on a kind structure, since we can always extend the set of monotypes to include a (virtual) base type τ_k to satisfy this condition.

We say that a free type variable in a typing derivation of $\mathcal{K}, \mathcal{T} \vdash e : \tau$ in λ^{ml} is *vacuous* if it does not appear in $EFTV(\mathcal{K}, \mathcal{T}) \cup EFTV(\mathcal{K}, \tau)$. Although translation is coherent, existence of vacuous type variables causes a problem in specialization since we cannot determine type attributes for some subterms. The following property guarantees that for any typing there is a typing derivation that does not contain vacuous type variables.

Corollary 2. *If $\mathcal{K}, \mathcal{T} \vdash e : \tau$ then $\mathcal{K}', \mathcal{T} \vdash e : \tau$ for some $\mathcal{K}'$ such that $dom(\mathcal{K}') = EFTV(\mathcal{K}, \mathcal{T}) \cup EFTV(\mathcal{K}, \tau)$.*

This is an immediate consequence of the above assumption on kinding and Lemma 1. Among possible Λ^{ml} terms corresponding a given λ^{ml} declaration, we choose one that does not contain vacuous type variables. The translation algorithm for declaration is now given inductively as follows.

1. The translation of $\emptyset$ is $\emptyset$.
2. The translation of $d\{x = e\}$ is obtained as follows. From d, we obtain a declaration D and its type assignment $\mathcal{T}$. From a typing derivation for $\mathcal{K}, \mathcal{T} \vdash e : \tau$, compute a typing $\mathcal{K}, \mathcal{T} \vdash M : \tau$. Let $t_1, \ldots, t_n$ be the set of vacuous type variables, let $\mathcal{K} = \{t_1{::}k_1, \ldots, t_n{::}k_n\}\{\langle t{::}k \rangle\}$, and let $k'_i = [\tau_{k'_1}/t_1, \ldots, \tau_{k'_{i-1}}/t_{i-1}]k_i$. Then the declaration for $d\{x = e\}$ is

$$\begin{aligned} D\{x &: \forall(\langle t{::}[\tau_{k'_1}/t_1, \ldots .\tau_{k'_n}/t_n]k\rangle).[\tau_{k'_1}/t_1, \ldots .\tau_{k'_n}/t_n]\tau\} \\ &= \Lambda(\langle t{::}[\tau_{k'_1}/t_1, \ldots .\tau_{k'_n}/t_n]k\rangle).[\tau_{k'_1}/t_1, \ldots .\tau_{k'_n}/t_n]M\} \end{aligned}$$

The coherence result guarantees that we can safely perform the substitution $[\tau_{k'_1}/t_1, \ldots .\tau_{k'_n}/t_n]$. Moreover, the resulting Λ^{ml} declaration contains no free type variable, and is therefore suitable for subsequent specialization.

5 Examples of Specialization and Their Analysis

As we mentioned in Introduction, we have developed compilation methods for polymorphic record operations [25] and for efficient lambda binding [24]. This section gives simplified accounts of these two methods and analyzes their common structures. For simplicity of presentation, in the following explanation, we only show implicitly typed calculi.

5.1 Compilation of Polymorphic Record Operations

The Source Language. Here we only consider the following minimal set of terms

$$e ::= c^b \,|\, x \,|\, \lambda x.e \,|\, e\ e \,|\, let\ x = e\ in\ e \,|\, \{l = e, \ldots, l = e\} \,|\, e{\cdot}l$$

where l stands for a given set of labels. The sets of kinds and monotypes are given by the syntax

$$\begin{aligned} \tau &::= b \,|\, t \,|\, \tau \rightarrow \tau \,|\, \{l : \tau, \ldots, l : \tau\} \\ k &::= U \,|\, \{\!\{l : \tau, \ldots, l : \tau\}\!\} \end{aligned}$$

where $\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$ is a labeled record type and $\{\!\{l_1 : \tau_1, \ldots, l_n : \tau_n\}\!\}$ is a record kind denoting the set of record types containing the fields $l_1 : \tau_1, \ldots, l_n : \tau_n$. The kind $\{\!\{l_1 : \tau_1, \ldots, l_n : \tau_n\}\!\}$ is regarded as shorthand for the conjunction of the atomic record kinds $\{\!\{l_1 : \tau_1\}\!\}, \ldots, \{\!\{l_n : \tau_n\}\!\}$.

The typing rule for polymorphic field selection is given as follows.

$$(\textsc{dot})\ \frac{\mathcal{K}, \mathcal{T} \vdash e\ :\ \tau_1 \quad \mathcal{K} \vdash \tau_1 :: \{\!\{l : \tau_2\}\!\}}{\mathcal{K}, \mathcal{T} \vdash e{\cdot}l\ :\ \tau_2}$$

The following is an example of typing.

$$\{t_1{::}U, t_2{::}\{\!\{l : t_1\}\!\}\}, \emptyset \vdash \lambda x.x{\cdot}l\ :\ t_2 \rightarrow t_1$$

Combining with kinded let polymorphism, this term can be used as a term having the polymorphic type $\forall(t_1{::}U, t_2{::}\{\!\{l : t_1\}\!\}).t_2 \rightarrow t_1$. The same mechanism can also be used to represent polymorphic field update and polymorphic variants.

The Implementation Language and Specialization. The idea of specializing polymorphic field selection operation is to decompose it to an indexing operation and an index value. If $\mathcal{I}$ denotes the index value of the label l in the type of e, then labeled field selection $e{\cdot}l$ is translated to index operation $e[\mathcal{I}]$.

The set of indexes is given by the following syntax.

$$\mathcal{I} ::= n \,|\, I$$

where n denotes natural numbers (used as index values) and I denotes a set of *index variables.* The set of terms of the implementation calculus is given by the syntax

$$C ::= x \,|\, c^b \,|\, \lambda x.C \,|\, C\ C \,|\, \{C, \ldots, C\} \,|\, C\,[\mathcal{I}] \,|\, let\ x = C\ in\ C \,|\, \delta I.C \,|\, C\ \mathcal{I}$$

where $\{C, \ldots, C\}$ is a vector representation of a labeled record, $C[\mathcal{I}]$ is index expression, $\delta I.C$ is *index abstraction*, and $C\ \mathcal{I}$ is *index application*.

The set of monotypes of the implementation calculus is given by the following syntax.

$$\tau ::= t \,|\, b \,|\, \tau \to \tau \,|\, \{l : \tau, \cdots, l : \tau\} \,|\, index(l, \tau) \Rightarrow \tau$$

$index(l, \tau)$ is an *index type* denoting the index value corresponding to l in the record type τ, and $index(l, \tau) \Rightarrow \tau$ denotes functions that take the index value denoted by $index(l, \tau)$ and return a value of type τ.

Since index values (values denoted by types of the form $index(l, \tau)$) are always static, we do not treat $index(l, \tau)$ as a first-class type, but instead, introduce a different static judgments for index values. An *index assignment* $\mathcal{L}$ is a function from a finite set of index variables to index types. We write $\mathcal{L} \vdash \mathcal{I} \,:\, index(l, \tau)$ if $\mathcal{I}$ is the index value of l in type τ. We assume that the fields in a record type are sorted by labels. This relation is given by the following rules.

$$\mathcal{L} \vdash i \,:\, index(l_i, \{l_0 : \tau_0, \ldots, l_i : \tau_i, \ldots, l_n : \tau_n\})$$

$$\mathcal{L}\{I : index(l, \tau)\} \vdash I \,:\, index(l, \tau)$$

Some of typing rules are given in Fig. 5.

$$\text{(IABS)}\quad \frac{\mathcal{K}, \mathcal{L}\{I : index(l, \tau_1)\}, \mathcal{T} \vdash C_1 \,:\, \tau_2}{\mathcal{K}, \mathcal{L}, \mathcal{T} \vdash \lambda I.C_1 \,:\, index(l, \tau_1) \Rightarrow \tau_2}$$

$$\text{(IAPP)}\quad \frac{\mathcal{K}, \mathcal{L}, \mathcal{T} \vdash C \,:\, index(l, \tau_1) \Rightarrow \tau_2 \quad \mathcal{L} \vdash \mathcal{I} \,:\, index(l, \tau_1)}{\mathcal{K}, \mathcal{L}, \mathcal{T} \vdash C\ \mathcal{I} \,:\, \tau_2}$$

$$\text{(RECORD)}\quad \frac{\mathcal{K}, \mathcal{L}, \mathcal{T} \vdash C_i \,:\, \tau_i \quad (1 \le i \le n)}{\mathcal{K}, \mathcal{L}, \mathcal{T} \vdash \{C_1, \cdots, C_n\} \,:\, \{l_1 : \tau_1, \cdots, l_n : \tau_n\}}$$

$$\text{(INDEX)}\quad \frac{\mathcal{K}, \mathcal{L}, \mathcal{T} \vdash C_1 \,:\, \tau_1 \quad \mathcal{K} \vdash \tau_1 :: \{\!\{l : \tau_2\}\!\} \quad \mathcal{L} \vdash \mathcal{I} \,:\, index(l, \tau_1)}{\mathcal{K}, \mathcal{L}, \mathcal{T} \vdash C_1\,[\mathcal{I}] \,:\, \tau_2}$$

Fig. 5. Some of Typing Rules of the Polymorphic Record Calculus

The following theorem is proved in [25].

Theorem 3. *There is a type-preserving and behavior-preserving translation algorithm from the source calculus to the implementation calculus.*

As an example, the field selection function

$$\{t_1{::}U, t_2{::}\{\!\{l{:}t_1\}\!\}\}, \emptyset \vdash \lambda x.x{\cdot}l \,:\, t_2 \to t_1$$

is translated to the following typing.

$$\{t_1{::}U, t_2{::}\{\!\{l{:}t_1\}\!\}\}, \emptyset, \emptyset \vdash \lambda I.\lambda x.x[I] \,:\, index(l, t_2) \Rightarrow t_2 \to t_1$$

When this function is let-bound and used for some instance type, the necessary index value is inserted. For example, from

$$let\ f = \lambda x.x{\cdot}l\ in\ (f\ \{l = ''a'', m = 2\},\ f\ \{a = true, l = ''b''\})$$

the translator will produce the following code

$$let\ f = \lambda I.\lambda x.x[I]\ in(f\ 0\ \{''a'', 2\},\ f\ 1\ \{true, ''b''\})$$

under the assumption that a labeled record is encoded as a vector of values sorted by labels, and the first entry has index 0.

5.2 Unboxed Semantics for ML Polymorphism

The Source Language. Here we only consider the set of raw terms of Core ML.

As mentioned in Introduction, the crucial information for unboxed semantics is the size of values. If the size of the argument is known, then lambda abstraction (and other size sensitive operations such as second projection) can be implemented efficiently without requiring that all the data must be boxed values (pointers to heap allocated values.) We therefore need to keep track of those type variables whose size information is needed. To achieve this, we introduce a constant kind S denoting those types whose size information is needed.

Among the Core ML terms, lambda abstraction and let expression are those whose efficient compilation requires size information. This property is represented by the following typing rules.

$$(\textsc{abs})\quad \frac{\mathcal{K}, \mathcal{T}\{x : \tau_1\} \vdash e_1\ :\ \tau_2 \qquad \mathcal{K} \vdash \tau_1 :: S}{\mathcal{K}, \mathcal{T} \vdash \lambda x.e_1\ :\ \tau_1 \to \tau_2}$$

$$(\textsc{let})\quad \frac{\mathcal{K}\{\langle t{::}k\rangle\}, \mathcal{T} \vdash e_1\ :\ \tau_1 \qquad \mathcal{K} \vdash \tau_1 :: S \qquad \mathcal{K}, \mathcal{T}\{x : \forall(\langle t{::}k\rangle).\tau_1\} \vdash e_2\ :\ \tau_2}{\mathcal{K}, \mathcal{T} \vdash let\ x = e_1\ in\ e_2\ :\ \tau_2}$$
$$\text{if } \langle t\rangle \cap FTV(\mathcal{T}) = \emptyset$$

Other typing rules are standard.

For example, under this type system, the identity function $\lambda x.x$ is given the following typing.

$$\{t{::}S\}, \emptyset \vdash \lambda x.x\ :\ t \to t$$

The Implementation Language and Specialization. The idea of specializing a boxed polymorphic operation is to decompose it into a pair of an unboxed operation and a size information. If $\mathcal{I}$ denotes the size of x, then generic lambda abstraction $\lambda x.e$ can be translated to $\lambda^{\mathcal{I}}x.e$ where $\lambda^{\mathcal{I}}$ is lambda abstraction specialized to size $\mathcal{I}$. Different from generic lambda abstraction, this operation is implemented without requiring run-time objects to be boxed.

The syntax for terms denoting sizes (ranged over by $\mathcal{I}$) is the same as those for index values in the record calculus. The set of terms of the unboxed calculus is given by the syntax.

$$C ::= x \mid c^b \mid \lambda^{\mathcal{I}} x.C \mid C\ C \mid let^{\mathcal{I}}\ x = C\ in\ C \mid \delta I.C \mid C\ \mathcal{I}$$

The set of monotypes is given by the following syntax.

$$\tau ::= t \mid b \mid \tau \to \tau \mid size(\tau) \Rightarrow \tau$$

$size(\tau)$ is a *size type* denoting the size of values of type τ, and $size(\tau) \Rightarrow \tau$ denotes functions that take the size denoted by $size(\tau)$ and return value of type τ. We assume that the size of values of type τ is determined by the out-most type constructor of τ if τ is not a type variable, otherwise the size is undefined.

A *size type assignment* $\mathcal{L}$ is a function from a finite set of size variables to size types. We write $\mathcal{L} \vdash \mathcal{I} : size(\tau)$ if $\mathcal{I}$ denotes the size of type τ. This relation is given by the following rules.

$$\mathcal{L} \vdash n : size(\tau) \qquad \tau \text{ is not a type variable and its size is } n$$

$$\mathcal{L}\{I : size(\tau)\} \vdash I : size(\tau)$$

Some of typing rules are given in Fig. 6.

$$\text{(IABS)}\quad \frac{\mathcal{K}, \mathcal{L}\{I : size(\tau_1)\}, \mathcal{T} \vdash C_1 : \tau_2}{\mathcal{K}, \mathcal{L}, \mathcal{T} \vdash \lambda I.C_1 : size(\tau_1) \Rightarrow \tau_2}$$

$$\text{(IAPP)}\quad \frac{\mathcal{K}, \mathcal{L}, \mathcal{T} \vdash C : size(\tau_1) \Rightarrow \tau_2 \qquad \mathcal{L} \vdash \mathcal{I} : size(\tau_1)}{\mathcal{K}, \mathcal{L}, \mathcal{T} \vdash C\ \mathcal{I} : \tau_2}$$

$$\text{(ABS)}\quad \frac{\mathcal{K}, \mathcal{L}, \mathcal{T}\{x : \tau_1\} \vdash C_1 : \tau_2 \qquad \mathcal{K} \vdash \tau_1 :: S \qquad \mathcal{L} \vdash \mathcal{I} : size(\tau_1)}{\mathcal{K}, \mathcal{L}, \mathcal{T} \vdash \lambda^{\mathcal{I}} x.C_1 : \tau_1 \to \tau_2}$$

$$\text{(LET)}\quad \begin{array}{l} \mathcal{K}\{\langle t{::}k\rangle\}, \mathcal{T} \vdash C_1 : \tau_1 \qquad \mathcal{K} \vdash \tau_1 :: S \qquad \mathcal{L} \vdash \mathcal{I} : size(\tau_1) \\ \mathcal{K}, \mathcal{T}\{x : \forall(\langle t{::}k\rangle).\tau_1\} \vdash C_2 : \tau_2 \\ \hline \mathcal{T} \vdash let^{\mathcal{I}}\ x = C_1\ in\ C_2 : \tau_2 \\ \text{if } \langle t\rangle \cap FTV(\mathcal{T}) = \emptyset \end{array}$$

Fig. 6. Some of Typing Rules of the Unboxed Calculus

The following theorem is proved in [24].

Theorem 4. *There is a type-preserving translation algorithm from the source calculus to the implementation calculus.*

As an example, the identity function

$$\{t{::}S\}, \emptyset \vdash \lambda x.x : t \to t$$

is translated to the following typing.

$$\{t{::}S\}, \emptyset, \emptyset \vdash \lambda^1 I.\lambda^I x.x \;:\; size(t) \Rightarrow t \rightarrow t$$

When this function is let-bound and used for some instance type, the necessary size value is inserted. For example, from

$$let\ f = \lambda x.x\ in\ (f\ 3,\ f\ 3.14)$$

the translator will produce the following code.

$$let\ f = \lambda^1 I.\lambda^I x.x\ in(f\ 1\ 3,\ f\ 2\ 3.14)$$

where we assume that the size of a natural number is 1 and that of a floating point number is 2.

5.3 Analysis of Specializations

The two examples of type based specialization appear to share general structures. Close analysis of the two examples reveals the following common properties.

- Translation is mostly generic. Only small portion of code needs specialization. In the examples, the translations are generically done as follows

$$\lambda x.x{\cdot}l \longrightarrow \lambda x.x[\mathcal{I}]$$
$$\lambda x.x \longrightarrow \lambda^{\mathcal{I}} x.x$$

  by the type (kind) information of x, and only the offset value and the size value $\mathcal{I}$ need to be specialized.
- Kinding information determines what sort of information is required during specialization. For example, if a term e has the following typing in the record calculus,

$$\{t{::}\{\!\{l : bool, m : int\}\!\}\}, \emptyset \vdash e \;:\; t \rightarrow \tau$$

  then from the kinding, we know that e contains field selections of labels l and m from a value having type t. We can therefore translate this typing to the following typing in the implementation calculus by introducing two index variables as follows.

$$\{t{::}\{\!\{l : bool, m : int\}\!\}\}, \{I_1 : index(l,t), I_2 : index(m,t)\}, \emptyset \vdash C_e \;:\; t \rightarrow \tau$$

- Specialization information is statically computed by the type system from the type of the actual argument. For example, consider again the following expression

$$let\ f = \lambda x.x{\cdot}l\ in\ (f\ \{l = \text{"}a\text{"}, m = 2\},\ f\ \{a = true, l = \text{"}b\text{"}\})$$

  which is translated to the following code

$$let\ f = \lambda I.\lambda x.x[I]\ in(f\ 0\ \{\text{"}a\text{"}, 2\},\ f\ 1\ \{true, \text{"}b\text{"}\})$$

  The index value 0 and 1 is determined by the type system at the time of compilation.

- Computation of required specialization information is done by treating attribute values as types. The polymorphic field selection has the following polytype.

  $$\forall t_1{::}U.\forall t_2{::}\{\!\{l : t_1\}\!\}.index(l, t_2) \Rightarrow t_2 \to t_1$$

  When this function is used, the required specialization information is determined statically by the type $index(l, \tau)$ where τ is the actual instance type of t_2. For example, if this function is applied to a value of type $\{l : string, m : int\}$, then since 0 is the only value of the type $index(l, \{l : string, m : int\})$, the required value for I is determined to be 0. This treatment of values as types enables us to perform systematic specialization without resorting to run-time time analysis.

From these analysis, we observe the following common structure in type based specialization of polymorphism.

1. A polymorphic primitive is decomposed into a low-level generic operation and an attribute value that determines the behavior of the operation.
2. Polymorphism is recovered by introducing an abstraction mechanism over attributes.
3. The type system computes necessary attributes values statically by treating attribute values themselves as types.

By developing a typed implementation calculus supporting these features, and a type preserving translation scheme from the source language to the implementation language, it should be possible to apply the type based specialization exploited in the above two examples to a wide range of polymorphic primitives.

There are some similarities between type based specialization analyzed above and *intentional polymorphism* by Harper and Morrisett [11], which also exploits type information. Their framework is based on run-time type analysis, and therefore, as a type system, it is more general and can express various non parametric operations easily. However, it does not fully address the issue of specialization of polymorphism into efficient code. For example, if a type is passed at run-time and the run-time system performs type analysis, then the run-time system can certainly compute the index value of a label in a labeled record type, but this strategy is equivalent to searching a label at run-time and does not necessarily contribute to producing efficient code. As summarized above, one distinguishing feature of our type based specialization is that it statically computes those attributes of a type that are relevant for efficient execution, and only those relevant values are passed at run-time. This feature is crucial in compiling polymorphism. Developing a systematic method to achieve this feature is far from trivial.

6 A Framework for Type Based Specialization

6.1 The Source Language

The source language is the explicitly typed kinded calculus Λ^{ml} extended with polymorphic primitives. Some polymorphic primitive may only be represented as a term constructor. We therefore extend Λ^{ml} with polymorphic term constructors (ranged over by π). The typing of π is characterized by a kind k_π denoting the possible types of its argument, and the type constructor T_{k_π} of the result type. The source calculus is extended with the following typing rule

$$(\pi)\ \frac{\mathcal{K},\mathcal{T} \vdash M\ :\ \tau \quad \mathcal{K} \vdash \tau :: k_\pi}{\mathcal{K},\mathcal{T} \vdash \pi(M : \tau)\ :\ T_{k_\pi}(\tau)}$$

for each term constructor π. The type annotation is necessary to maintain the explicit typing of Λ^{ml}. We further assume that this is a rule scheme denoting all the rules obtained by instantiating k_π.

6.2 The Implementation Calculus : Λ^{impl}

As explained earlier, a polymorphic primitive $\pi(M : \tau)$ is compiled into a low-level generic operation C_π and an attribute value that determines the behavior of C_π according to the type of the argument. To represent such operation, we assume that for each atomic kind k_π introduced for a primitive operation π, there is an associated type attribute P_{k_π} such that for any type τ of kind k_π that is not a type variable, $P_{k_\pi}(\tau)$ denotes a unique attribute value. We write $|P_{k_\pi}(\tau)|$ for the unique value denoted by $P_{k_\pi}(\tau)$. If M has type τ and $c = |P_{k_\pi}(\tau)|$, then the polymorphic construct $\pi(M)$ is compiled to $C_\pi(C_M, c)$. We shall formalize this representation as a typing rule of Λ^{impl} below.

A key feature of the implementation calculus is to treat such a type attribute as a type denoting the attribute itself, and to introduce an abstraction mechanism over attributes. We call type attributes of the form $P_{k_\pi}(\tau)$ *attribute types* and use α for a meta variable ranging over attribute types. We introduce a set of *attribute variables* (ranged over by A). The set of attribute terms is given by the following syntax.

$$a ::= c \mid A$$

The set of terms of Λ^{impl} is given by the following syntax.

$$\begin{aligned} C ::= c^b \mid (x\ \langle\tau\rangle\ \langle a\rangle) \mid \lambda x : \tau.C \mid C\ C \mid C_\pi(C : \tau, a) \\ \mid \mathit{let}\ x : \sigma = \Lambda(\langle t{::}k\rangle).(\langle A : \alpha\rangle).C\ \mathit{in}\ C \end{aligned}$$

$(x\ \langle\tau\rangle\ \langle a\rangle)$ is a variable with type instantiation and attribute application. In $\mathit{let}\ x : \sigma = \Lambda(\langle t{::}k\rangle).(\langle A : \alpha\rangle).C_1\ \mathit{in}\ C_2$, type variables $\langle t\rangle$ and attribute variables $\langle A\rangle$ in C_1 are abstracted. We write $[a_1/A_1, \ldots, a_m/A_m]C$ for the term obtained from C by substituting a_i for A_i. Term substitution is refined to the operator that integrates type instantiation and attribute substitution. We write $[(t_1, \ldots, t_n).(A_1, \ldots, A_m).C_1/x]C_2$ for the term obtained from C_2 by substituting $[a_1/A_1, \ldots, a_m/A_m]([\tau_1/t_1, \ldots, \tau_n/t_n]C_1)$ for each occurrence of the form $(x\ \tau_1 \cdots \tau_n\ a_1 \cdots a_m)$. Its formal inductive definition is easily given.

The reduction axioms (other than those of C_π) are as follows.

$$(\beta)\quad (\lambda x : \tau.C_1)\ C_2 \Longrightarrow [().().C_2/x]C_1$$

$$(\textsc{let})\quad \mathit{let}\ x : \sigma = \Lambda(\langle t{::}k\rangle).(\langle A : \alpha\rangle).C_1\ \mathit{in}\ C_2 \Longrightarrow [(\langle t\rangle).(\langle A\rangle).C_1/x]C_2$$

The reduction relation of Λ^{impl} is defined as usual.

An *attribute type assignment* (ranged over by $\mathcal{L}$) is a mapping from a finite set of attribute variables to attribute types. We write

$$\mathcal{L} \vdash a\ :\ \alpha$$

when attribute value a has attribute type α under $\mathcal{L}$. This relation is defined by the following rules.

$$(\textsc{const})\quad \mathcal{L} \vdash c\ :\ P_k(\tau) \qquad \text{if } \tau \text{ is not a type variable and } |P_k(\tau)| = c$$

$$(\textsc{avar})\quad \mathcal{L}\{A : \alpha\} \vdash A\ :\ \alpha$$

The sets of monotypes and polytypes are given by the following syntax.

$$\begin{aligned}\tau &::= t \,|\, b \,|\, \tau \to \tau \,|\, T_k(\tau)\\ \sigma &::= \forall(\langle t{::}k\rangle).(\langle\alpha\rangle) \Rightarrow \tau\end{aligned}$$

Note that polymorphic type abstraction and attribute abstraction are combined in the definition of polytypes.

An attribute type assignment $\mathcal{L}$ is *well formed* under kind assignment $\mathcal{K}$, denoted by $\mathcal{K} \vdash \mathcal{L}$, if $FTV(\mathcal{L}) \subseteq dom(\mathcal{K})$. The type system of Λ^{impl} is defined as a proof system to derive the following form of judgments.

$$\mathcal{K}, \mathcal{L}, \mathcal{T} \vdash C\ :\ \tau$$

The set of typing rules is given in Fig. 7.

For this calculus, the following three substitution lemmas hold.

Lemma 6. *If* $\mathcal{K}\{t{::}k\}\mathcal{K}', \mathcal{L}, \mathcal{T} \vdash C\ :\ \tau$ *and* $\mathcal{K} \vdash \tau_0 :: k$ *then* $\mathcal{K}([\tau_0/t]\mathcal{K}'), [\tau_0/t]\mathcal{L}, [\tau_0/t]\mathcal{T} \vdash [\tau_0/t]C\ :\ [\tau_0/t]\tau$.

This is proved similarly to the corresponding proof of Λ^{ml} using the assumption we made that typings of polymorphic primitives are closed under type instantiation.

Lemma 7. *If* $\mathcal{K}, \mathcal{L}\{A_1 : \alpha_1, \ldots, A_n : \alpha_n\}, \mathcal{T} \vdash C\ :\ \tau$ *and* $\mathcal{L} \vdash a_i\ :\ \alpha_i$ $(1 \leq \imath \leq n)$ *then* $\mathcal{K}, \mathcal{L}, \mathcal{T} \vdash [a_1/A_1, \ldots, a_n/A_n]C\ :\ \tau$.

Proof is by simple induction on C.

Lemma 8. *If* $\mathcal{K}\{\langle t{::}k\rangle\}, \mathcal{L}\{\langle A\ :\ \alpha\rangle\}, \mathcal{T} \vdash C_2\ :\ \tau_0$*, no* $t' \in \langle t\rangle$ *occurs free in* $\mathcal{T} \cup \mathcal{L}$*, and* $\mathcal{K}, \mathcal{L}, \mathcal{T}\{x\ :\ \forall(\langle t{::}k\rangle).(\langle\alpha\rangle) \Rightarrow \tau_0\} \vdash C_1\ :\ \tau$*, then* $\mathcal{K}, \mathcal{L}, \mathcal{T} \vdash [(\langle t\rangle).(\langle A\rangle).C_2/x]C_1\ :\ \tau$.

This is proved by induction on C_1.

Using these lemma, we can show the subject reduction theorem by assuming that the additional axioms for C_π preserve typings.

$$(\textsc{const})\quad \mathcal{K},\mathcal{L},\mathcal{T} \vdash c^b : b \quad \text{if } \mathcal{K} \vdash \mathcal{T} \text{ and } \mathcal{K} \vdash \mathcal{L}$$

$$(\textsc{var})\quad \frac{\begin{array}{l}\mathcal{K} \vdash \tau_i :: [\tau_1/t_1,\ldots,\tau_{i-1}/t_{i-1}]k_i \\ \mathcal{L} \vdash a_i : [\tau_1/t_1,\ldots,\tau_n/t_n](\alpha_i) \\ \mathcal{K} \vdash \mathcal{T}\{x : \forall(t_1{::}k_1,\ldots,t_n{::}k_n).(\alpha_1,\ldots,\alpha_m) \Rightarrow \tau_0\} \\ \mathcal{K} \vdash \mathcal{L}\end{array}}{\begin{array}{l}\mathcal{K},\mathcal{L},\mathcal{T}\{x : \forall(t_1{::}k_1,\ldots,t_n{::}k_n).(\alpha_1,\ldots,\alpha_m) \Rightarrow \tau_0\} \\ \quad \vdash (x\ \tau_1 \cdots \tau_n\ a_1 \cdots a_m) : [\tau_1/t_1,\ldots,\tau_n/t_n]\tau_0\end{array}}$$

$$(\textsc{app})\quad \frac{\mathcal{K},\mathcal{L},\mathcal{T} \vdash C_1 : \tau_1 \to \tau_2 \quad \mathcal{K},\mathcal{L},\mathcal{T} \vdash C_2 : \tau_1}{\mathcal{K},\mathcal{L},\mathcal{T} \vdash C_1\ C_2 : \tau_2}$$

$$(\textsc{abs})\quad \frac{\mathcal{K},\mathcal{L},\mathcal{T}\{x : \tau_1\} \vdash C_1 : \tau_2}{\mathcal{K},\mathcal{L},\mathcal{T} \vdash \lambda x : \tau_1.C_1 : \tau_1 \to \tau_2}$$

$$(C_\pi)\quad \frac{\mathcal{K},\mathcal{L},\mathcal{T} \vdash C : \tau \quad \mathcal{K} \vdash \tau :: k_\pi \quad \mathcal{L} \vdash a : P_{k_\pi}(\tau)}{\mathcal{K},\mathcal{L},\mathcal{T} \vdash C_\pi(C : \tau, a) : T_{k_\pi}(\tau)}$$

$$(\textsc{let})\quad \frac{\begin{array}{l}\mathcal{K}\{\langle t{::}k\rangle\},\mathcal{L}\{\langle A : \alpha\rangle\},\mathcal{T} \vdash C_1 : \tau_1 \\ \mathcal{K},\mathcal{L},\mathcal{T}\{x : \forall(\langle t{::}k\rangle).(\langle\alpha\rangle) \Rightarrow \tau_1\} \vdash C_2 : \tau_2\end{array}}{\mathcal{K},\mathcal{L},\mathcal{T} \vdash let\ x : \forall(\langle t{::}k\rangle).(\langle\alpha\rangle) \Rightarrow \tau_1 = \Lambda(\langle t{::}k\rangle).(\langle A : \alpha\rangle).C_1\ in\ C_2 : \tau_2}$$
$$\text{if no } t_i \in \langle t\rangle \text{ is in } FTV(\mathcal{T}) \cup FTV(\mathcal{L})$$

Fig. 7. The Type System of the Implementation Calculus Λ^{impl}

6.3 Type Based Specialization

We now present type based specialization of polymorphism as an algorithm to transform Λ^{ml} typings into Λ^{impl} typings. The key idea is to combine the transformation from the Λ^{ml} typing

$$(\pi)\quad \frac{\mathcal{K},\mathcal{T} \vdash M : \tau \quad \mathcal{K} \vdash \tau :: k_\pi}{\mathcal{K},\mathcal{T} \vdash \pi(M : \tau) : T_{k_\pi}(\tau)}$$

to the Λ^{impl} typing

$$(C_\pi)\quad \frac{\mathcal{K},\mathcal{L},\mathcal{T} \vdash C : \tau \quad \mathcal{K} \vdash \tau :: k_\pi \quad \mathcal{L} \vdash a : P_{k_\pi}(\tau)}{\mathcal{K},\mathcal{L},\mathcal{T} \vdash C_\pi(C : \tau, a) : T_{k_\pi}(\tau)}$$

with appropriate type abstraction and attribute abstraction for polymorphic let expression.

We first define the relationship between types of Λ^{ml} and those of Λ^{impl}. Let $\sigma = \forall(t_1{::}k_1,\ldots,t_n{::}k_n).\tau$ be a polytype of Λ^{ml}. In general each kind k_i is a conjunction of atomic kinds $k_i^1,\ldots,k_i^l$. We say that $t{::}k$ is an atomic kinding if k is an atomic kind, and we say that an atomic kinding $t'{::}k'$ is in $t_1{::}k_1,\ldots,t_n{::}k_n$ if there is some $t_i{::}k_i$ such that $t' = t_i$ and k' is one of conjuncts of k_i. Let $t'_1{::}k'_1,\ldots,t'_m{::}k'_m$ be the set of all atomic kindings in $t_1{::}k_1,\ldots,t_n{::}k_n$. The Λ^{impl} type $(\sigma)^*$ corresponding to σ is the following type.

$$(\sigma)^* = \forall(t_1{::}k_1,\ldots,t_n{::}k_n).(P_{k'_1}(t'_1),\ldots,P_{k'_m}(t'_m)) \Rightarrow \tau$$

We extend this relation to type assignment. For a type assignment $\mathcal{T}$ of Λ^{ml}, $(\mathcal{T})^*$ is the type assignment of Λ^{impl} such that $dom((\mathcal{T})^*) = dom(\mathcal{T})$, and $(\mathcal{T})^*(x) = (\mathcal{T}(x))^*$ for all $x \in dom(\mathcal{T})$.

Let $\langle t{::}k \rangle$ be the set of all atomic kindings in a kind assignment $\mathcal{K}$. We define the attribute type assignment $\mathcal{L}_{\mathcal{K}}$ induced by $\mathcal{K}$ as

$$\mathcal{L}_{\mathcal{K}} = \{\langle A : P_k(t) \rangle\}$$

where $\langle A \rangle$ is a set of fresh attribute variables.

The compilation algorithm is given in Fig. 8 as an algorithm $\mathcal{C}$ that takes $\mathcal{L}_{\mathcal{K}}, (\mathcal{T})^*$ and M and computes a term of the implementation calculus. Since $\mathcal{L}_{\mathcal{K}}$ has the property that it has an element $A : P_k(t)$ for each atomic kinding $t{::}k$ in $\mathcal{K}$, each attribute value a mentioned in the algorithm is uniquely determined, and therefore $\mathcal{C}$ is a deterministic algorithm.

$$\begin{aligned}
\mathcal{C}(\mathcal{L}_{\mathcal{K}}, (\mathcal{T})^*, (x\ \tau_1 \cdots \tau_n)) = {} & \text{let } (\forall(t_1{::}k_1, \ldots, t_n{::}k_n).(\alpha_1, \ldots \alpha_m) \Rightarrow \tau = (\mathcal{T})^*(x) \\
& \qquad \alpha_i' = [\tau_1/t_1, \ldots, \tau_n/t_n]\alpha_i \\
& \qquad a_i = \begin{cases} c & \text{if } |\alpha_i'| = c \\ A & \text{if } |\alpha_i'| \text{ is undefined and } (A : \alpha_i') \in \mathcal{L}_{\mathcal{K}} \end{cases} \\
& \text{in } (x\ \tau_1 \cdots \tau_n\ a_1 \cdots a_m)
\end{aligned}$$

$$\mathcal{C}(\mathcal{L}_{\mathcal{K}}, (\mathcal{T})^*, c^b) = c^b$$

$$\mathcal{C}(\mathcal{L}_{\mathcal{K}}, (\mathcal{T})^*, \lambda x : \tau.M) = \lambda x : \tau.\mathcal{C}(\mathcal{L}_{\mathcal{K}}, (\mathcal{T})^*\{x : \tau\}, M)$$

$$\mathcal{C}(\mathcal{L}_{\mathcal{K}}, (\mathcal{T})^*, M_1\ M_2) = \mathcal{C}(\mathcal{L}_{\mathcal{K}}, (\mathcal{T})^*, M_1)\ \mathcal{C}(\mathcal{L}_{\mathcal{K}}, (\mathcal{T})^*, M_2)$$

$$\begin{aligned}
\mathcal{C}(\mathcal{L}_{\mathcal{K}}, (\mathcal{T})^*, \pi(M : \tau)) = {} & \text{let } C_1 = \mathcal{C}(\mathcal{L}_{\mathcal{K}}, (\mathcal{T})^*, M) \\
& \qquad a = \begin{cases} c & \text{if } |P_{k_\pi}(\tau)| = c \\ A & \text{if } |P_{k_\pi}(\tau)| \text{ is undefined and } (A : P_{k_\pi}(\tau))) \in \mathcal{L}_{\mathcal{K}} \end{cases} \\
& \text{in } C_\pi(C_1 : \tau, a)
\end{aligned}$$

$$\begin{aligned}
& \mathcal{C}(\mathcal{L}_{\mathcal{K}}, (\mathcal{T})^*, let\ x : \forall(\langle t{::}k \rangle).\tau_1 = \Lambda(\langle t{::}k \rangle).M_1\ in\ M_2) \\
& \quad = \text{let } \forall(\langle t{::}k \rangle).(\alpha_1, \ldots, \alpha_m) \Rightarrow \tau_1 = (\forall(\langle t{::}k \rangle).\tau_1)^* \\
& \qquad\qquad A_1, \ldots, A_m \text{ fresh} \\
& \qquad\qquad C_1 = \mathcal{C}(\mathcal{L}_{\mathcal{K}}\{A_1 : \alpha_1, \ldots A_m : \alpha_m\}, (\mathcal{T})^*, M_1) \\
& \qquad\qquad C_2 = \mathcal{C}(\mathcal{L}_{\mathcal{K}}, (\mathcal{T}\{x : \forall(\langle t{::}k \rangle).\tau_1\})^*, M_2) \\
& \quad\ \text{in } let\ x : \forall(\langle t{::}k \rangle).(\alpha_1, \ldots, \alpha_m) \Rightarrow \tau_1 = \Lambda(\langle t{::}k \rangle).(A_1 : \alpha_1, \ldots, A_m : \alpha_m).C_1\ in\ C_2
\end{aligned}$$

Fig. 8. The Compilation Algoritm from Λ^{ml} to Λ^{impl}

This algorithm preserves typings in the following sense.

Theorem 5. *If $\mathcal{K}, \mathcal{T} \vdash M : \tau$ is a typing in Λ^{ml} and $\mathcal{C}(\mathcal{L}_{\mathcal{K}}, (\mathcal{T})^*, M) = C$ then $\mathcal{K}, \mathcal{L}_{\mathcal{K}}, (\mathcal{T})^* \vdash C : \tau$ is a typing in Λ^{impl}.*

Proof is by induction on the structure of M.

The correctness of the type based specialization presented above can be shown by setting up a form of logical relation between terms of Λ^{ml} and those of Λ^{impl} having related types. (See [20] for a survey on logical relations and their applications.) Here we outline the general structure of the correctness proof as a generalization of the result show in [25].

The actural logical relation depends on the type constructors T_{k_π} introduced for each primitive operations and the semantics used for asserting equivalence of behavior of terms. Here we assume that there is a typed axiomatic semantics of the form $\mathcal{K}, \mathcal{T} \vdash M = N : \tau$ of Λ^{ml} and $\mathcal{K}, \mathcal{L}, \mathcal{T} \vdash C_1 = C_2 : \tau$ of Λ^{impl} induced by the axioms (β), $(\textsc{let})$ and the additional axioms for polymorphic term constructors π.

For a closed monotype τ of Λ^{ml}, define the sets $term^\tau$ and $Term^\tau$ of closed terms of Λ^{ml} and Λ^{impl}, respectively, as follows.

$$term^\tau = \{M | \emptyset, \emptyset \vdash M : \tau\}$$
$$Term^\tau = \{C | \emptyset, \emptyset, \emptyset \vdash C : \tau\}$$

The strategy for establishing the preservation of the behavior of a program is to define a type indexed family of relations $\{R^\tau \subseteq term^\tau \times Term^\tau\}$ that is sufficiently strong to ensure that the related terms have the same behavior. Let $(M, C) \in R^\tau$. A proper definition for base types and function types is as follows.

- if $\tau = b$ then for any c^b, $\emptyset, \emptyset \vdash M = c^b : b \Longleftrightarrow \emptyset, \emptyset, \emptyset \vdash C = c^b : b$.
- if $\tau = \tau_1 \rightarrow \tau_2$ then for any $(M_1, C_1) \in R^{\tau_1}$, $(M\ M_1, C\ C_1) \in R^{\tau_2}$.

This relation is extended to closed polytypes as follows.

Let $\sigma = \forall(t_1{::}k_1, \ldots, t_n{::}k_n).\tau$ and $\forall(t_1{::}k_1, \ldots, t_n{::}k_n).(\alpha_1, \ldots, \alpha_m) \Rightarrow \tau = (\sigma)^*$. Define R^σ as follows.

$$((t_1, \ldots t_n).M,\ (t_1, \ldots t_n).(A_1, \ldots A_m).C) \in R^\sigma$$

if the following conditions hold: for any types $\tau_1, \ldots, \tau_n$ such that $\emptyset \vdash \tau_i :: [\tau_1/t_1, \ldots, \tau_{i-1}/t_{i-1}]k_i$,

$$\begin{array}{l}([\tau_1/t_1, \ldots, \tau_n/t_n]M, [c_1/A_1, \ldots, c_m/\alpha_m]([\tau_1/t_1, \ldots, \tau_n/t_n]C)) \\ \quad \in\ R^{[\tau_1/t_1, \ldots, \tau_n/t_n]\tau}\end{array}$$

where $c_i = |[\tau_1/t_1, \ldots, \tau_n/t_n]\alpha_i|$.

Note that elements of R^σ are not pairs of terms but those of values in the ranges of substitutions. Based on this extension, the relation is extended to the relations on term substitutions of Λ^{ml} and those of Λ^{impl}. We use η for a meta variable for term substitutions of Λ^{ml} and Λ^{impl}. Let $\mathcal{T}$ be a closed type assignment of Λ^{ml}. Define $R^{\mathcal{T}}$ as follows.

$$(\eta_1, \eta_2) \in R^{\mathcal{T}} \Longleftrightarrow \forall x \in dom(\mathcal{T}).(\eta_1(x), \eta_2(x)) \in R^{\mathcal{T}(x)}$$

To establish the correctness of compilation, it is sufficient to prove the following property.

Suppose $\mathcal{K}, \mathcal{T} \vdash M : \tau$ be any Λ^{ml} typing, and $\mathcal{C}(\mathcal{L}_{\mathcal{K}}, (\mathcal{T})^*, M) = C$. Let $\mathcal{K} = \{t_1{::}k_1, \ldots, t_n{::}k_n\}$, and let $\mathcal{L}_{\mathcal{K}} = \{A_1 : \alpha_1, \ldots, A_m : \alpha_m\}$. For any $\tau_1, \ldots, \tau_n$ such that $\emptyset \vdash \tau_i :: [\tau_1/t_1, \ldots, \tau_{i-1}/t_{i-1}]k_i$, for any $(\eta_1, \eta_2) \in R^{[\tau_1/t_1,\ldots,\tau_n/t_n]\mathcal{T}}$,

$$(\eta_1([\tau_1/t_1, \ldots, \tau_n/t_n]M),\ \eta_2([a_1/A_1, \ldots, a_m/A_m]([\tau_1/t_1, \ldots, \tau_n/t_n]C))) \\ \in R^{[\tau_1/t_1,\ldots,\tau_n/t_n]\tau}$$

where $a_i = |[\tau_1/t_1, \ldots, \tau_n/t_n]\alpha|$.

We believe that R can be defined for most of the standard type constructors introduced for polymorphic primitives, and this property can be proved. See [25] for an example of such a proof for polymorphic record and variants.

7 Implementation Strategy

Λ^{impl} is a model for a language that can be implemented efficiently without using type information at run-time. The evaluation model of Λ^{impl} is therefore the calculus obtained by erasing all the type annotation from Λ^{impl}, which we call λ^{impl}. The set of terms of λ^{impl} is given by the following syntax.

$$e ::= c^b \mid (x\ a_1 \cdots a_m) \mid \lambda x.e \mid e\ e \mid C_\pi(e, a) \mid let\ x = \Lambda(A_1, \ldots, A_m).e\ in\ e$$

where we erase type information but not specialization information. The operational semantics that is the closest to programming language implementation is perhaps natural semantics [14]. A natural semantics is usually given by a set of evaluation relation of the form

$$E \vdash e \Downarrow v$$

denoting the property that term e evaluates to value v under run-time environment E, which is a mapping from variables to values. Here we describe an extension to this form of semantics to λ^{impl}, which would give an implementation strategy for λ^{impl}.

The set of values of λ^{impl} contains the following

$$v ::= c^b \mid cls(\lambda x.e, E) \mid \Lambda(\langle\alpha\rangle).v \mid a$$

where $cls(\lambda x.e, E)$ is a function closure and $\Lambda(\langle\alpha\rangle).v$ is attribute abstraction. Note that type attributes are always values even if they are variables. Some of evaluation rules are given in Fig. 9, where we used attribute substitution of the form $[\langle v/A\rangle]v$. The effect of this is the value v' obtained by applying the substitution $[\langle v/A\rangle]$ to the result parts of environments appearing in v. To obtain a complete semantics we need to include appropriate values and evaluation rules for additional constructors introduced for primitive operators. We should also prove several standard properties such as type soundness, and develop practical implementation techniques for the semantics. We leave them for future research.

$$E \vdash c^b \Downarrow c^b$$

$$\frac{E(x) = \Lambda(A_1, \ldots, A_m).v \quad E \vdash a_i \Downarrow v_i}{E \vdash (x\ a_1 \cdots a_m) \Downarrow [v_1/A_1, \ldots, v_m/A_m]v}$$

$$E \vdash A \Downarrow E(A)$$

$$E \vdash \lambda x.e \Downarrow cls(\lambda x.e, E)$$

$$\frac{E \vdash e_1 \Downarrow cls(\lambda x.e_1', E') \quad E \vdash e_2 \Downarrow v_2 \quad E'\{x : v_2\} \vdash e_1' \Downarrow v}{E \vdash e_1\ e_2 \Downarrow v}$$

$$\frac{E\{A_1 : A_1, \ldots, A_m : A_m\} \vdash e_1 \Downarrow v_1 \quad E\{x : \Lambda(A_1, \ldots, A_m).v_1\} \vdash e_2 \Downarrow v}{E \vdash let\ x = \Lambda(A_1, \ldots, A_m).e_1\ in\ e_2 \Downarrow v}$$

Fig. 9. Some of Evaluation Rules of λ^{impl}

As seen from the rules for let and variables, the operational semantics evaluates the inside of specialization abstraction of the form $\Lambda(A_1, \ldots, A_m).e$, and attribute application performs substitution. This is needed to preserve the order of evaluation of Λ^{ml}. Since attributes are always static and reduction of primitive operations will be performed only after they are fully specialized, the above strategy would yield a sound evaluation strategy. Wright [32] proposed "value-only polymorphism," where polymorphism is restricted to syntactic values, to avoid anomalous interaction between polymorphism and imperative features. If we adopt this strategy, then a specialization abstraction can be implemented as an ordinary closure. While this strategy significantly simplifies implementation of λ^{impl}, we would like to avoid an unnecessary restriction as part of the language definition. As we shall comment below, we believe that the semantics described above can be implemented reasonably efficiently.

8 Conclusions and Further Investigations

We have presented a framework for type based specialization of polymorphism. We have first developed a method for coherent type reconstruction for an ML style implicitly typed polymorphic language. We have then given a framework for specializing polymorphic primitives into efficient low-level code depending on the type of arguments. This has been achieved by decomposing a polymorphic operation into a low-level generic operation and an attribute value determined by the type of the argument, and by introducing an abstraction mechanism over attributes. A polymorphic function is implemented by a function that takes an attribute value and performs efficient operation according to the attribute value. One distinguishing feature of our approach is that it treats attribute values as types. By exploiting this feature, we have achieved type based specialization of polymorphic functions.

There are a number of issues that would merit further investigation. We briefly mention some of them below.

One interesting issue would be to apply the framework presented here to *intentional polymorphism* of Harper and Morrisett [11]. So far we have tacitly assumed that the value denoted by an attribute type of the form $P_k(\tau)$ is atomic. This is the case for record polymorphism and unboxed calculus where an attribute type such as $index(l, \{l : int, m : bool\})$ and $size(real)$ denotes an integer (namely 0 and 2, respectively in these examples.) However, our formalism does not require such a restriction. The value denoted by attribute type $P_k(\tau)$ can be any value as far as it is statically computed. With this extension, our formalism can deal with less uniform polymorphic primitives, such as those handled by intentional polymorphism. For example, polymorphic equality can be treated as follows. Suppose the calculus contains product types and a constant kind EQ denoting the following set of closed types.

$$\delta ::= b \,|\, \delta \times \delta$$

The polymorphic equality $eq(e)$ has the following typing rule.

$$(\text{EQ})\ \frac{\mathcal{K}, \mathcal{T} \vdash e : \tau \times \tau \quad \mathcal{K} \vdash \tau :: EQ}{\mathcal{K}, \mathcal{T} \vdash eq(e) : bool}$$

Let $P_{EQ}(\tau)$ denotes the attribute type whose value is the equality function on τ. The attribute judgments can be given as follows.

$$(\text{B})\ \mathcal{L} \vdash eq^b : P_{EQ}(b)$$

$$(\text{FUN})\ \frac{\mathcal{L} \vdash A_1 : P_{EQ}(\tau_1) \quad \mathcal{L} \vdash A_2 : P_{EQ}(\tau_2)}{\mathcal{L} \vdash \lambda(x, y).(A_1\ x)\ and\ (A_2\ y) : P_{EQ}(\tau_1 \times \tau_2)}$$

Here, eq^b is the primitive equality function on base type b, and we have used "product patter" in lambda abstraction. Using this attribute judgment, we can translate the above equality typing as follows.

$$(\text{EQ})\ \frac{\mathcal{K}, \mathcal{L}, \mathcal{T} \vdash C : \tau \times \tau \quad \mathcal{K} \vdash \tau :: EQ \quad \mathcal{L} \vdash eq : P_{EQ}(\tau)}{\mathcal{K}, \mathcal{T} \vdash Eq(C, eq) : bool}$$

The evaluation of $Eq(C, eq)$ can be implemented by $(eq\ C)$. We believe that other non-parametric primitives can be treated similarly. This approach provides an alternative to a limited form of intentional polymorphism. A careful comparison with [11] would be beneficial in revealing the limitation and the strength of the two approaches.

Recently, Minamide [19] developed an explicit type passing calculus that combines some of the features of Λ^{impl} and a mechanism for efficient type passing. In our framework, we only consider those kinds that denote a subset of monotypes. One novel aspect of [19] is that it includes a mechanism to perform some computation on type parameters by introducing a form of second-order kinds such as those denoting products of types, and associated elimination operations. Integration of this mechanism with type based specialization would yield a more flexible calculus suitable for an intermediate language for implementing various advanced features of polymorphic languages.

There are also several recent papers for various type passing calculi and optimization methods based on type information such as [31, 9, 26, 8, 30, 6]. Compared with these methods, the feature that distinguishes our approach is a type-theoretical treatment of static computation of attributes by treating attribute values as types. This feature may be useful for refining those type passing calculi. One promising approach toward this direction would be to develop a method for optimizing those type passing calculi by evaluating attributes at compile-time based on our framework.

In a more practical perspective, we need to develop a systematic method to implement λ^{impl}. The crucial issue is to develop an efficient mechanism for attribute abstraction and attribute application. The semantics we described in the previous section requires that the body of attribute abstraction should be evaluated. As we suggested in [25], this can be done roughly as follow. When evaluating $\mathit{\Lambda}(A_1, \ldots, A_m).C$, the system first allocates dummy entries for $A_1, \ldots, A_m$ in the current evaluation environment and evaluates C. This should yield a closure whose environment contains those dummy entries. This closure is saved as a template. The attribute substitution can be implemented by making a copy of the template closure and updating the dummy entries in the environment to actual attribute values. We are currently formalizing this intuitive strategy as a robust operational semantics, and developing the necessary implementation techniques.

With these efforts of further refinements, we hope that the framework presented in this paper will serve as a type theoretical basis for efficient implementation of various advanced polymorphism.

Acknowledgments

The author would like to thank Takayasu Ito for his helpful comments.

References

1. Appel, A. W. and MacQueen, D. B. Standard ML of New Jersey. In *Proceedings of the 3rd International Symposium on Programming Languages and Logic Programming*. Lecture Notes in Computer Science, vol. 528, 1–13, 1991.
2. Breazu-Tannen, V., Coquand, T., Gunter, C., and Scedrov, A. Inheritance as explicit coercion. *Inf. Comput. 93*, 172–221, 1991.
3. Buneman, P. and Ohori, A. Polymorphism and type inference in database programming. *ACM Trans. Database Syst.*, 21(1), 30-74, 1996.
4. Damas, L. and Milner, R. Principal type-schemes for functional programs. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, 207–212, 1982.
5. de Bruijn, N. G. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indag. Math.*, Vol. 34, pp. 381–392, 1972.

6. Dubois, C and Weise, P. Generic polymorphism. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1995.
7. Girard, J.-Y. Une extension de l'interpretation de gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et théorie des types. In the *2nd Scandinavian Logic Symposium.* North-Holland, Amsterdam, 1971.
8. Hall, C. Using Hindley-Milner type inference to optimize list representation. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, 1994.
9. Hall, C., Hammond, K., Peyton Jones, S., and Wadler, P. Type class in Haskell. Tech. rep., Univ. of Glasgow, Glasgow, Scotland, 1994.
10. Harper, R. and Mitchell, J. C. On the type structure of Standard ML. *ACM Trans. Program. Lang. Syst. 15,* 2, 211–252, 1993.
11. Harper, R. and Morrisett, G. Compiling polymorphism using intensional type analysis. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, 130–141, 1995.
12. Hudak, P. *et. al.* Report on programming language Haskell a non-strict, purely functional language version 1.2. *SIGPLAN Notices*, Vol. 27, No. 5, 1992.
13. Jones, M. ML typing, explicit polymorphism and qualified types. In *Proceedings of Theoretical Aspects of Computer Software*, Lecture Notes in Computer Science, vol. 789, pages 56 –75, 1994.
14. Kahn, G. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, pages 22–39. Springer Verlag, Berlin, 1987.
15. Leroy, X. The ZINC Experiment: an economical implementation of the ML language. Technical report, No. 117, INRIA, France, 1992.
16. Leroy, X. Unboxed objects and polymorphic typing. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, 177–188, 1992.
17. Milner, R. A theory of type polymorphism in programming. *J. Comput. Syst. Sci. 17*, 348–375, 1978.
18. Milner, R., Tofte, M., and Harper, R. *The Definition of Standard ML.* MIT Press, Cambridge, Mass, 1990.
19. Minamide, Y. Compilation based on a calculus for explicit type passing. In *Proceedings of Fuji International Workshop on Functional and Logic Programming*, 301–320, 1996.
20. Mitchell, J. Type systems for programming languages. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. MIT Press, Cambridge, Mass., 365–458, 1990.
21. *The Objective Caml User's Manual.* INRIA Rocquencourt. B.P. 105,78153 Le Chesnay, France.
22. Ohori, A. A simple semantics for ML polymorphism. In *Proceedings of the ACM/IFIP Conference on Functional Programming Languages and Computer Architecture*, 281–292, 1989.
23. Ohori, A. A compilation method for ML-style polymorphic record calculi. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, 154–165, 1992.
24. Ohori, A. and Takamizawa, T. A polymorphic unboxed calculus as an abstract machine for polymorphic languages. To appear in *Journal of Lisp and Symbolic Computation*, 1997.
25. Ohori, A. A polymorphic record calculus and its compilation. *ACM Trans. Program. Lang. Syst.*, 17(6):844–895, 1995.
26. Peterson, J. and Jones, M. Implementing type classes. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 227–236, 1993.

27. Peyton Jones, S.L. and Launchbury, J. Unboxed values as first class citizens in a non-strict functional language. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 636–666. Lecture Notes in Computer Science, Vol 523. Springer-Verlag, Berlin, 1991.
28. Rémy, D. Efficient representation of extensible records. In *Proceedings of the ACM SIGPLAN Workshop on ML and Its Applications*, 12–16, 1994.
29. Reynolds, J. Towards a theory of type structure. In the *Paris Colloquium on Programming*. Springer-Verlag, Berlin, 408–425, 1974.
30. Shao, Z., Reppy, J., and Apple, A. W. Unrolling lists. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, 1994.
31. Tolmach, A. Tag-free garbage collection using explicit type parameters. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, 1–11, 1994.
32. Wright, A.K. Polymorphism for imperative languages without imperative types. Technical Report TR93-200, Rice University, 1993.