Type-Directed Specialization of Polymorphism*

Atsushi Ohori†
Research Institute for Mathematical Sciences
Kyoto University
Sakyo-ku, Kyoto 606-8502 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, these two features conflict each other 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 analyze the existing methods for compiling a record calculus and an unboxed calculus, extract their common structure, and develop a framework for type-directed specialization of polymorphism.

1 Introduction

The term "polymorphism" implies *generic behavior* 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 a field selection function on records

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

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

An important achievement in type theory of programming languages is the development of polymorphic type systems where generic behavior of a program such as above is cleanly represented by a polymorphic type. In Girard-Reynolds type system [10, 36], the function id is given the following type

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

*This is the author's version of the article published in *Information and Computation*, 155:64–107, 1999. A preliminary version of this article was published in Proceeding of TACS Conference, LNCS 1281, pages 107–137, September 1997, as an invited paper under the title: "A type system for specializing polymorphism."

†Partly supported by the Japanese Ministry of Education Grant-in-Aid for Scientific Research on Priority Area 275: "advanced databases", and by the Parallel and Distributed Processing Research Consortium, Japan.

representing the polymorphic behavior of *id*. This form of polymorphism is embodied in the type system of ML [23]. 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 [24], OCaml [22] and Haskell [15]. Recent studies of *record polymorphism* enable us to represent polymorphic functions operating on labeled record structures. Using the type system of [31], 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 $\{a : int, b : bool\}$. We call the typing mechanism refined with kinds *kinded typing*. This mechanism is similar to *bounded quantification* [6] where the range of type variables may be constraint to subtypes of a given type. Due to the refinement of kinded typing, ML-style parametric polymorphism can scale up to large and complicated practical software development such as database programming, where labeled data structures are essential (see, for example, [5] for discussion on polymorphism in database programming.)

It is highly desirable to develop a practical programming language that supports flexible polymorphic typing. However, polymorphism inherently conflicts with run-time efficiency of programs – another essential feature of programming languages. The source of polymorphism is generic 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 understand the mismatch consider the two polymorphic functions given above. The function $\lambda x.x$ is polymorphic because the variable binding mechanism in the lambda calculus is inherently generic. In an actual computer hardware, however, values have various different sizes, and therefore efficient variable binding requires size dependent operation. Similarly, the source of polymorphism of the function $\lambda x.\, x{\cdot}l$ is the generic field selection operation $\_{\cdot}l$ which accesses a field by a symbolic name. However, no currently available general purpose computer architecture efficiently supports named field access.

In a statically typed monomorphic language, the compiler generates optimized code specialized to the type of the actual data. For example, a labeled record is represented as a vector (sorted by field labels), and field selection is compiled to code that performs indexing into a vector. 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 less efficient data representation and costly run-time analysis. 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 [34, 20] and efficient data representation for associative access for labeled records [35]. These methods show some positive results, but polymorphic primitives remain 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 ideally 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 [31], which specializes polymorphic operations on labeled records and labeled variants to index operations. The other is an unboxed semantics for ML polymorphism [32], which specializes polymorphic variable binding to efficient size sensitive variable binding. These two methods 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 article is to analyze these two methods, to identify the crucial issues in program specialization, and to develop a framework for type-directed specialization of polymorphism. Specific contributions of this article include

- a refined presentation of ML and its type reconstruction method,
- a generic implementation language which can be used for an intermediate language for various polymorphic primitives, and
- a generic algorithm for type-directed specialization of polymorphism and its type preservation theorem.

In addition to giving a framework for type-directed specialization, we also provide a new operational semantics for the implementation language and discuss implementation strategies. We hope that the framework presented here will serve as a type-theoretical basis for compiling polymorphic functions into efficient code.

The rest of this paper is organized as follows. In Section 2, we describe the general structure of type-directed specialization of polymorphism. Section 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 behavior of a given program. There is one subtle problem in this process. That is the problem of "coherence" in type reconstruction. Section 4 analyzes this problem and provides a solution. In Section 5, we analyze the two specialization methods mentioned above and extract the general properties of type-directed specialization. Section 6 gives a framework for specialization. We first define an implementation language where specialization of generic primitives can be representable. We then develop a translation algorithm from the source language into the implementation language. For this implementation language, we give a call-by-value operational semantics and establish the type soundness. Finally, Section 7 concludes the paper with suggestions for further investigations.

2 General Structure of Specialization

The source language we are interested in is an ML-style programming language, whose polymorphism is based on *implicit typing*. The programmer writes a program without type specification, and the type system automatically infers its most general polymorphic type that represents the generic behavior of the program.

In conventional implementations of an ML-style language such as [2, 21], 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. However, the approach of type-directed specialization we are advocating makes crucial use of type information in compiling programs. To exploit type information, we use an explicitly typed intermediate language and organize the type-directed specialization into the following two phases.

1. Type Reconstruction.
   This phase performs type inference on a given raw term, and constructs an explicitly typed

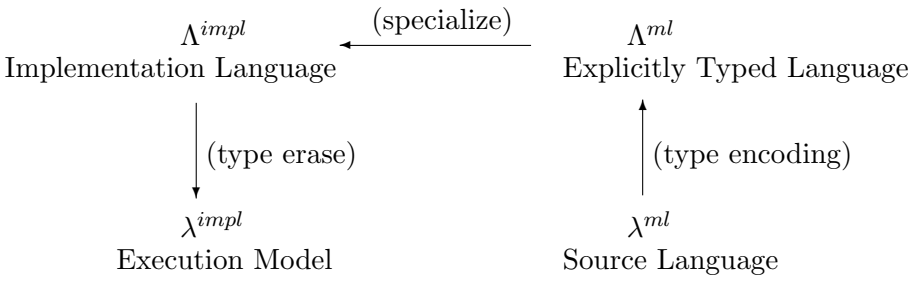

Figure 1: Relationship Among the Languages.

term that encodes all the inferred type information. This phase can be regarded as the combination of the inference of a typing derivation in an implicitly typed source language (denoted here by λ^{ml}) and the construction of a term of an explicitly typed language (denoted here by Λ^{ml}) from the typing derivation in λ^{ml} .

2. Specialization.
   This phase transforms a term of Λ^{ml} into a term of an implementation language, 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 computing type attribute at run-time. The implicitly typed language λ^{impl} obtained from Λ^{impl} by forgetting type information models efficient rum-time execution.

Recent works on type-directed compilation such as TIL compiler [37], intensional type analysis [13], and typed closure conversion [27] use typed intermediate languages for a similar purpose. Figure 1 shows the relationship among the languages considered in this paper.

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 construction of a Λ^{ml} term from a given λ^{ml} typing derivation is crucial in type-directed specialization. For this reason, we include a careful analysis on Λ^{ml} and construction of Λ^{ml} terms from λ^{ml} typing derivations, but 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.

3 The Source Language

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

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

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 term 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.

Terms are considered modulo 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

$$\text{(VAR)}\ \ \mathcal{T}\{x:\sigma\} \vdash x : \sigma \qquad \text{(CONST)}\ \ \mathcal{T} \vdash c^b : b$$

$$\text{(APP)}\ \ \frac{\mathcal{T} \vdash e_1 : \tau_1 \to \tau_2 \quad \mathcal{T} \vdash e_2 : \tau_1}{\mathcal{T} \vdash e_1\ e_2 : \tau_2} \qquad \text{(ABS)}\ \ \frac{\mathcal{T}\{x:\tau_1\} \vdash e_1 : \tau_2}{\mathcal{T} \vdash \lambda x.e_1 : \tau_1 \to \tau_2}$$

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

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

Figure 2: Damas-Milner Type System of Core ML

from any free variables, and we assume that this property is preserved by substitution. We shall implicitly make this assumption for all the languages containing bound variables we shall consider in this paper, including polymorphic types.

A type system for Core ML is given by Damas and Milner [8]. 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.

$$\begin{aligned} \tau &= b \mid t \mid \tau \to \tau \\ \sigma &= \tau \mid \forall t.\sigma \end{aligned}$$

A *type substitution*, or simply *substitution*, is a function from a finite set of type variables to monotypes. We write $[\tau_1/t_1, \cdots, \tau_n/t_n]$ for the substitution that maps each t_i to τ_i. A substitution S is extended to the set of all type variables by letting $S(t) = t$ for all $t \notin dom(S)$, and it in turn is extended uniquely to all the monotypes. The result of applying a substitution S to a polytype is the type obtained by applying S to its all *free type variables*. Under the bound type variable convention, we can simply take $S(\forall t_1 \ldots t_n.\tau) = \forall t_1 \ldots t_n.S(\tau)$. A substitution is also extended to other objects containing types. The set of free type variables of τ is denoted by $FTV(\tau)$. In [8], 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 function from a finite set of variables to polytypes. If f is a function then we write $f\{x:v\}$ for the extension of f that maps x to v, provided that $x \notin dom(f)$. The set of typing rules of Dama-Milner system is given in Figure 2.

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 the same properties. For example, a record kind $\{\!\{l:\tau\}\!\}$ denotes the set of all record types containing the field $l:\tau$. The purpose of introducing kind constraints is twofold.

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

This mechanism was first introduced in [30] for compiling record polymorphism. Jones [16] proposed a similar mechanism of "quantified types" for compiling type classes.

Required kinds depend on the form of polymorphism we would like to support. Instead of assuming some particular kinds, we consider the source language λ^{ml} as a family of languages

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 the kind 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 the conjunction of two atomic record kinds $\{\!\{l : t_1\}\!\}$ and $\{\!\{m : t_2\}\!\}$.

A concrete syntax of the set of kinds can be the following

$$k ::= B \mid C(\tau_1, \ldots, \tau_n) \mid k \wedge k$$

where B stands for a given set of constant kinds, C stands for a given set of kind constructors, and $k_1 \wedge k_2$ is the conjunction of consistent kinds k_1 and k_2. However, the following development will not depend on the concrete syntax of kinds.

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 (or a set) of elements of the form $a_1, \ldots, a_n$ when the exact sequence (set members) are not important. In particular, we sometimes write $\{\langle t{::}k \rangle\}$ for a kind assignment. The empty kind assignment is denoted by $\emptyset$. 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 $dom(\mathcal{K}) \cap dom(\mathcal{K}') = \emptyset$. These notations are compatible with our notation for function extension.

We say that a monotype 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 monotypes. Since we allow a kind to contain monotypes, 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.

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

Note that if $\vdash \mathcal{K}\{\langle t_1{::}k_1 \rangle\}\{\langle t_2{::}k_2 \rangle\}\mathcal{K}'$ and $\mathcal{K} \vdash \langle k_2 \rangle$ then $\vdash \mathcal{K}\{\langle t_2{::}k_2 \rangle\}\{\langle t_1{::}k_1 \rangle\}\mathcal{K}'$.

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

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

denoting the fact that monotype 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 $\vdash \mathcal{K}$, $\mathcal{K} \vdash \tau$ and $\mathcal{K} \vdash k$.
2. If $\mathcal{K} \vdash \tau$ then $\mathcal{K} \vdash \tau :: U$.
3. If $\mathcal{K}\{\langle t_1 :: k_1\rangle\}\{\langle t_2 :: k_2\rangle\}\mathcal{K}' \vdash \tau :: k$ and $\mathcal{K} \vdash \langle k_2\rangle$ then $\mathcal{K}\{\langle t_2 :: k_2\rangle\}\{\langle t_1 :: k_1\rangle\}\mathcal{K}' \vdash \tau :: k$
4. If $\mathcal{K}\{\langle t_0 :: k_0\rangle\}\mathcal{K}' \vdash \tau :: k$, $\vdash \mathcal{K}\mathcal{K}'$, $\mathcal{K}\mathcal{K}' \vdash \tau$, and $\mathcal{K}\mathcal{K}' \vdash k$ then $\mathcal{K}\mathcal{K}' \vdash \tau :: k$
5. If $\mathcal{K}\mathcal{K}' \vdash \tau :: k$, $\langle t_0\rangle \cap dom(\mathcal{K}\mathcal{K}') = \emptyset$ and $\vdash \mathcal{K}\{\langle t_0 :: k_0\rangle\}$ then $\mathcal{K}\{\langle t_0 :: k_0\rangle\}\mathcal{K}' \vdash \tau :: k$,
6. If $\mathcal{K}\{t_0 :: k_0\}\mathcal{K}' \vdash \tau :: k$ and $\mathcal{K} \vdash \tau_0 :: k_0$ then $\mathcal{K}([\tau_0/t_0]\mathcal{K}') \vdash [\tau_0/t_0]\tau :: [\tau_0/t_0]k$.

The first five properties are standard structural ones that should be satisfied by any kinding system. The last property requires that kinding is closed under kind-respecting type substitution.

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

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

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

If $n = 0$ then it is identified with τ. In this construct, type variable t_i is bound in $k_{i+1}, \ldots, k_n$ and τ. The set of free type variables of a polytype is defined as: $FTV(\forall(t_1 :: k_1 \ldots t_n :: k_n).\tau) = \bigcup_i (FTV(k_i) \setminus \{t_1, \ldots, t_{i-1}\}) \cup (FTV(\tau) \setminus \{t_1, \ldots, t_n\})$. A polytype σ is *well formed* under $\mathcal{K}$, written $\mathcal{K} \vdash \sigma$ if $FTV(\sigma) \subseteq dom(\mathcal{K})$. We extend this relation to type assignments, i.e., $\mathcal{K} \vdash \mathcal{T}$ if $\mathcal{K} \vdash \mathcal{T}(x)$ for all $x \in dom(\mathcal{T})$. Since kinds may contain type variables, when we apply a type substitution to a kinded polytype, we must also apply it to the set of kinds in the type. Under our bound type variable convention, we can simply define

$$S(\forall(t_1 :: k_1, \ldots, t_n :: k_n).\tau) = \forall(t_1 :: S(k_1), \ldots, t_n :: S(k_n)).S(\tau)$$

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

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

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 Figure 3. In rule (VAR), if $n = 0$ then it reduces to the usual variable axiom $\mathcal{K}, \mathcal{T}\{x : \tau\} \vdash x : \tau$. Note also that in rule (LET), $\mathcal{K}, \mathcal{T}\{x : \forall(\langle t :: k\rangle).\tau_1\} \vdash e_2 : \tau_2$ implies $\mathcal{K} \vdash \mathcal{T}\{x : \forall(\langle t :: k\rangle).\tau_1\}$, which guarantees the condition $\langle t\rangle \cap FTV(\mathcal{T}) = \emptyset$ required for type abstraction.

The following four lemmas can be shown by easy induction.

Lemma 3.1 *If* $\mathcal{K}\{\langle t_1 :: k_1\rangle\}\{\langle t_2 :: k_2\rangle\}\mathcal{K}', \mathcal{T} \vdash e : \tau$ *and* $\mathcal{K} \vdash \langle k_2\rangle$ *then* $\mathcal{K}\{\langle t_2 :: k_2\rangle\}\{\langle t_1 :: k_1\rangle\}\mathcal{K}', \mathcal{T} \vdash e : \tau$

Lemma 3.2 *If* $\mathcal{K}, \mathcal{T} \vdash e : \tau$, $dom(\mathcal{T}) \cap dom(\mathcal{T}') = \emptyset$, *and* $\mathcal{K} \vdash \mathcal{T}'$ *then* $\mathcal{K}, \mathcal{T}\mathcal{T}' \vdash e : \tau$.

Lemma 3.3 *If* $\mathcal{K}, \mathcal{T}\mathcal{T}' \vdash e : \tau$, *and* $dom(\mathcal{T}') \cap FV(e) = \emptyset$ *then* $\mathcal{K}, \mathcal{T} \vdash e : \tau$.

Lemma 3.4 *If* $\mathcal{K}\mathcal{K}', \mathcal{T} \vdash e : \tau$, $\langle t_0\rangle \cap dom(\mathcal{K}\mathcal{K}') = \emptyset$, *and* $\vdash \mathcal{K}\{\langle t_0 :: k_0\rangle\}$ *then* $\mathcal{K}\{\langle t_0 :: k_0\rangle\}\mathcal{K}', \mathcal{T} \vdash e : \tau$

The above four lemmas hold for all the language we shall define in this article.

The next lemma shows that typings are closed under kind-respecting type substitution.

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

$$(\text{VAR})\quad \frac{\mathcal{K} \vdash \mathcal{T}\{x : \forall(t_1{::}k_1, \ldots, t_n{::}k_n).\tau\} \qquad \mathcal{K} \vdash \tau_i :: [\tau_1/t_1, \ldots, \tau_{i-1}/t_{i-1}]k_i \ (1 \le i \le n)}{\mathcal{K}, \mathcal{T}\{x : \forall(t_1{::}k_1, \ldots, t_n{::}k_n).\tau\} \vdash x \,:\, [\tau_1/t_1, \ldots, \tau_n/t_n]\tau}$$

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

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

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

Figure 3: The Type System of λ^{ml}

Lemma 3.5 *If* $\mathcal{K}\{t_0{::}k_0\}\mathcal{K}', \mathcal{T} \vdash e \,:\, \tau$ *and* $\mathcal{K} \vdash \tau_0 :: k_0$ *then* $\mathcal{K}([\tau_0/t_0]\mathcal{K}'), [\tau_0/t_0]\mathcal{T} \vdash e \,:\, [\tau_0/t_0]\tau$.

Proof By induction on e. We only show the cases for variables and let expressions; other cases follow directly from the corresponding induction hypotheses.

Case x. Suppose $\mathcal{K}\{t_0{::}k_0\}\mathcal{K}', \mathcal{T}\{x : \forall(t_1{::}k_1, \ldots, t_n{::}k_n).\tau'\} \vdash x \,:\, \tau$. Then by the definition of the type system, $\mathcal{K}\{t_0{::}k_0\}\mathcal{K}' \vdash \mathcal{T}\{x : \forall(t_1{::}k_1, \ldots, t_n{::}k_n).\tau'\}$, and there must be some $\tau_1, \ldots, \tau_n$ such that $\mathcal{K}\{t_0{::}k_0\}\mathcal{K}' \vdash \tau_i :: [\tau_1/t_1, \ldots, \tau_{i-1}/t_{i-1}]k_i$ $(1 \le i \le n)$ and $\tau = [\tau_1/t_1, \ldots, \tau_n/t_n]\tau'$. By the assumptions on kinding, the well formedness condition of typing assignments, and bound type variable convention, we have

$$\mathcal{K}([\tau_0/t_0]\mathcal{K}') \vdash [\tau_0/t_0](\mathcal{T})\{x : \forall(t_1{::}[\tau_0/t_0]k_1, \ldots, t_n{::}[\tau_0/t_0]k_n).[\tau_0/t_0]\tau'\},$$

and

$$\mathcal{K}([\tau_0/t_0]\mathcal{K}') \vdash [\tau_0/t_0]\tau_i :: [[\tau_0/t_0]\tau_1/t_1, \ldots, [\tau_0/t_0]\tau_{i-1}/t_{i-1}]([\tau_0/t_0]k_i) \quad (1 \le i \le n).$$

Then by the rule (VAR),

$$\begin{aligned}&\mathcal{K}([\tau_0/t_0]\mathcal{K}'), [\tau_0/t_0](\mathcal{T})\{x : \forall(t_1{::}[\tau_0/t_0]k_1, \ldots, t_n{::}[\tau_0/t_0]k_n).[\tau_0/t_0]\tau'\}\\ &\quad \vdash \quad x \,:\, [[\tau_0/t_0]\tau_1/t_1, \ldots, [\tau_0/t_0]\tau_n/t_n]([\tau_0/t_0]\tau')\end{aligned}$$

By the bound type variable convention, we can assume that $t_1, \ldots, t_n$ does not appear in τ_0. Thus

$$\mathcal{K}([\tau_0/t_0]\mathcal{K}'), [\tau_0/t_0](\mathcal{T}\{x : \forall(t_1{::}k_1, \ldots, t_n{::}k_n).\tau'\}) \vdash x \,:\, [\tau_0/t_0]([\tau_1/t_1, \ldots, \tau_n/t_n]\tau').$$

as desired.

Case $\mathit{let}\ x = e_1\ \mathit{in}\ e_2$. Suppose $\mathcal{K}\{t_0{::}k_0\}\mathcal{K}', \mathcal{T} \vdash \mathit{let}\ x = e_1\ \mathit{in}\ e_2 \,:\, \tau$. Then we must have $\mathcal{K}\{t_0{::}k_0\}\mathcal{K}'\{\langle t{::}k\rangle\}, \mathcal{T} \vdash e_1 \,:\, \tau_1$ and $\mathcal{K}\{t_0{::}k_0\}\mathcal{K}', \mathcal{T}\{x : \forall(\langle t{::}k\rangle).\tau_1\} \vdash e_2 \,:\, \tau$. By the induction hypothesis and bound type variable convention, we have

$$\mathcal{K}([\tau_0/t_0]\mathcal{K}')\{\langle t{::}[\tau_0/t_0]k\rangle\}, [\tau_0/t_0]\mathcal{T} \vdash e_1 \,:\, [\tau_0/t_0]\tau_1$$

and

$$\mathcal{K}([\tau_0/t_0]\mathcal{K}'), [\tau_0/t_0](\mathcal{T})\{x : \forall(\langle t{::}[\tau_0/t_0]k\rangle).[\tau_0/t_0]\tau_1)\} \vdash e_2 \,:\, [\tau_0/t_0]\tau$$

By the rule (LET), we have

$$\mathcal{K}([\tau_0/t_0]\mathcal{K}'), [\tau_0/t_0](\mathcal{T}) \vdash \mathit{let}\ x = e_1\ \mathit{in}\ e_2 \,:\, [\tau_0/t_0]\tau$$

as desired. ∎

We say that a type variable is *free in a typing derivation* if it appears in some $\mathcal{T}$ or it appears as τ in some typing of the form $\mathcal{K}, \mathcal{T} \vdash e : \tau$ or some kinding of the form $\mathcal{K} \vdash \tau :: k$, and if it is not discharged by the rule (LET). Its inductive definition can easily be given. The following property holds for free type variables in a derivation.

Proposition 3.6 *The set of all free type variables of any typing derivation of $\mathcal{K}, \mathcal{T} \vdash e : \tau$ is contained in $dom(\mathcal{K})$.*

Proof By the assumption on kinding, if $\mathcal{K} \vdash \tau :: k$ then any type variable in τ or k is contained in $dom(\mathcal{K})$. It is therefore sufficient to verify that each typing rule preserves the following property: if $\mathcal{K}, \mathcal{T} \vdash e : \tau$ then $\mathcal{K} \vdash \mathcal{T}$ and $\mathcal{K} \vdash \tau$. ∎

It should be noted that in Damas-Milner system, the set of free type variables in a derivation cannot be determined from a typing. To see this, consider the following typing.

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

Although no type variable appears in this typing, a typing derivation may contain free type variables in types of x and y. Free type variables in a typing derivation are those that may affect type-directed specialization, and the above proposition guarantees that $\mathcal{K}$ records all those type variables.

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.

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

A declaration can be regarded as a nested let binding as shown in the following, which can be proved by simple induction on n.

Proposition 3.7 *If $\vdash \{x_1 = e_1, \ldots, x_n = e_n\} : \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ and $\mathcal{K}, \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\} \vdash e : \tau$ then $\mathcal{K}, \emptyset \vdash \mathit{let}\ x_1 = e_1\ \mathit{in}\ \cdots\ \mathit{let}\ x_n = e_n\ \mathit{in}\ e : \tau$.*

By taking e in the above proposition to be a trivial value () of a trivial type *unit*, then declaration $\vdash \{x_1 = e_1, \ldots, x_n = e_n\} : \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ can be regarded as the following closed typing

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

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

4 Reconstructing Explicitly Typed Terms

As observed by Harper and Mitchell [12], a typing derivation of Core ML corresponds to a typing of an explicitly typed term. For Damas-Milner type system, the corresponding set of explicitly typed terms, called XML terms, is given by the following grammar

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

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 shown in [12].

Theorem 4.1 *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 shorthand" 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 [29] that some typing corresponds to provably unequal XML terms. The following example is taken from [29]. Suppose we have two distinct base types b_1, b_2 and consider the following Damas-Milner typing:

$$\{x : \forall t.\, t \to b_1, y : \forall t.\, t \to t\} \vdash (x\ y)\ :\ b_1.$$

The following two XML terms both correspond to derivations of the above typing:

$$\{x : \forall t.\, t \to b_1, y : \forall t.\, t \to t\} \vdash ((x\ b_2 \to b_2)\ (y\ b_2))\ :\ b_1$$

$$\{x : \forall t.\, t \to b_1, y : \forall t.\, t \to t\} \vdash ((x\ b_1 \to b_1)\ (y\ b_1))\ :\ b_1$$

But since these terms are in normal form and therefore they are not convertible. Using the terminology of [4], we can say that reconstruction from Damas-Milner typings to XML terms cannot be *coherent*. Since type-directed specialization we are advocating relies on reconstruction of an explicitly typed term from an untyped term, the failure of coherence implies that the choice of a type reconstruction algorithm may affect the meaning of the compiled term. Such a situation is undesirable. Another more serious problem is that a term that causes the failure of coherence often contains free type variables that does not appear in the typing judgment. For example, consider the possible explicitly typed term for $(f\ (\lambda x.x{\cdot}l))$ under the assumption $\{f : \forall t.t \to b\}$. A natural type reconstruction strategy would produce the term $(f\ t_1 \to t_2)\ (\lambda x : t_1.x{\cdot}l)$ with the kind constraint $t_1 :: \{\!\{l : t_2\}\!\}$. Since t_1, t_2 do not appear in the type environment nor in the result type, they will not be abstracted nor instantiated. As a result, the type system cannot determine specialization information for the terms containing those type variables, and the specializer cannot produce a code for the function $(\lambda x : t_1.x{\cdot}l)$ occurring in the above context.

An example and an observation describing the same phenomenon of failure of coherence as above were re-stated in [17], and this problem was discussed. The solution suggested in [17] is simply to restrict terms to be a subset of legal ML terms to avoid above failure of coherence. However, in [29] we already showed that there is a coherent type reconstruction for all the ML terms based on a non-conventional presentation of ML polymorphism using only monotypes. In [12], it was also observed that translation of closed terms is coherent. Those results suggest that we can achieve coherent type reconstruction for a Damas-Milner style calculus. In this section, we provide a solution to this problem by defining an explicitly typed calculus Λ^{ml} as a refinement of XML, and giving a coherent transformation from λ^{ml} to Λ^{ml}.

4.1 The Explicitly Typed Calculus: Λ^{ml}

This section defines Λ^{ml} as an intermediate language for the type-directed specialization method. In order to discuss the problem of coherent type reconstruction, we need a semantic framework for

Λ^{ml}. We base our development on an equational theory analogous to the β equality in the lambda calculus. Since any reduction relation such as one corresponding to a call-by-value operational semantics should be sound with respect to this equational theory, the coherence condition obtained in this section can be regarded as the necessary condition that should be satisfied by any operational semantics.

We define Λ^{ml} as a language having 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.

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

$(x\ \langle\tau\rangle)$ is a variable with nested type application and $\Lambda(\langle t::k\rangle).M$ in let expression is nested type abstraction. If $\langle\tau\rangle$ in $(x\ \langle\tau\rangle)$ and $\langle t::k\rangle$ in $\Lambda(\langle t::k\rangle).M$ are empty then we regard them as x and M, respectively. The set of free variables of M is denoted by $FV(M)$, and the set of free type variables of M is denoted by $FTV(M)$. The definition of $FTV(M)$ is obtained by extending the following clauses according to the structure of M.

$$\begin{aligned} FTV((x\ \langle\tau\rangle)) &= FTV(\langle\tau\rangle) \\ FTV(\lambda x : \tau.M) &= FTV(\tau) \cup FTV(M) \\ FTV(let\ x : \sigma = \Lambda(t_1::k_1 \ldots t_n::k_n).M_1\ in\ M_2) & \\ = FTV(\sigma) \cup \bigcup_i (FTV(k_i) \setminus \{t_1, \ldots, t_{i-1}\}) \cup (FTV(M_1) \setminus \{t_1, \ldots, t_n\}) & \\ \cup FTV(M_2) & \end{aligned}$$

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 clauses according to the structure of the term.

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

For the case of $n = 0$, we simply write $[M/x]N$ instead of $[().M/x]N$. The reduction axioms for Λ^{ml} are given below.

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

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

We write $M \longrightarrow M'$ if M' is obtained from M by applying one of the reduction axioms to some subterm of M, and write $M \xrightarrow{*} M'$ for the reflexive transitive closure of the one-step reduction relation $M \longrightarrow M'$. The convertibility relation is written $M \xleftrightarrow{*} M'$.

The kinding relation is the same as that of λ^{ml}. The set of typing rules for Λ^{ml} is given in Figure 4. Similarly to λ^{ml}, in rule (LET), $\mathcal{K}, \mathcal{T}\{x : \forall(\langle t::k\rangle).\tau_1\} \vdash M_2 : \tau_2$ implies $\mathcal{K} \vdash \mathcal{T}\{x : \forall(\langle t::k\rangle).\tau_1\}$, which guarantees the condition $\langle t\rangle \cap FTV(\mathcal{T}) = \emptyset$ required for type abstraction.

The following property can be proved similarly to Lemma 3.5.

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

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

$$(\text{APP})\quad \frac{\mathcal{K}, \mathcal{T} \vdash M_1 \,:\, \tau_1 \to \tau_2 \quad \mathcal{K}, \mathcal{T} \vdash M_2 \,:\, \tau_1}{\mathcal{K}, \mathcal{T} \vdash M_1\ M_2 \,:\, \tau_2}$$

$$(\text{ABS})\quad \frac{\mathcal{K}, \mathcal{T}\{x : \tau_1\} \vdash M_1 \,:\, \tau_2}{\mathcal{K}, \mathcal{T} \vdash \lambda x : \tau_1.M_1 \,:\, \tau_1 \to \tau_2}$$

$$(\text{LET})\quad \frac{\mathcal{K}\{\langle t{::}k\rangle\}, \mathcal{T} \vdash M_1 \,:\, \tau_1 \quad \mathcal{K}, \mathcal{T}\{x : \forall(\langle t{::}k\rangle).\tau_1\} \vdash M_2 \,:\, \tau_2}{\mathcal{K}, \mathcal{T} \vdash \mathit{let}\ x : \forall(\langle t{::}k\rangle).\tau_1 = \Lambda(\langle t{::}k\rangle).M_1\ \mathit{in}\ M_2 \,:\, \tau_2}$$

Figure 4: The Type System of Λ^{ml}

Lemma 4.2 *If* $\mathcal{K}\{t_0{::}k_0\}\mathcal{K}', \mathcal{T} \vdash M \,:\, \tau$ *and* $\mathcal{K} \vdash \tau_0 :: k_0$ *then* $\mathcal{K}([\tau_0/t_0]\mathcal{K}'), [\tau_0/t_0]\mathcal{T} \vdash [\tau_0/t_0]M \,:\, [\tau_0/t_0]\tau$.

The following term substitution lemma also hold.

Lemma 4.3 *If* $\mathcal{K}, \mathcal{T}\{x \,:\, \forall(\langle t_0{::}k_0\rangle).\tau_0\} \vdash M_1 \,:\, \tau$ *and* $\mathcal{K}\{\langle t_0{::}k_0\rangle\}, \mathcal{T} \vdash M_2 \,:\, \tau_0$ *then* $\mathcal{K}, \mathcal{T} \vdash [(\langle t_0\rangle).M_2/x]M_1 \,:\, \tau$.

Proof By induction on M_1. We only show the cases for variables and let expressions. Other cases can be shown using the corresponding induction hypotheses.

Case $(y\ \langle\tau'\rangle)$. If $x \neq y$, then the result follows by the free variable lemma (Lemma 3.3 for Λ^{ml}). Let $x = y$ and suppose $\mathcal{K}, \mathcal{T}\{x \,:\, \forall(t_1{::}k_1, \ldots, t_n{::}k_n).\tau'\} \vdash (x\ \tau_1 \cdots \tau_n) \,:\, \tau$. Let $S_i = [\tau_1/t_1, \ldots, \tau_i/t_i]$. By the type system, we must have $\mathcal{K} \vdash \tau_i :: S_{i-1}(k_i)$ for $(1 \leq i \leq n)$. Using this property, Lemma 4.2 and bound type variable convention, we can show the following by induction on i.

$$\mathcal{K}\{t_{i+1}{::}S_i(k_{i+1}), \ldots, t_n{::}S_i(k_n)\}, S_i(\mathcal{T}) \vdash S_i(M_2) \,:\, S_i(\tau_0) \quad (1 \leq i \leq n)$$

Therefore we have

$$\mathcal{K}, S_n(\mathcal{T}) \vdash S_n(M_2) \,:\, S_n(\tau_0)$$

But since $S_n(\mathcal{T}) = \mathcal{T}$, $S_n(\tau_0) = \tau_0$, and $S_n(M_2) = [(t_1, \ldots, t_n).M_2/x](x\ \tau_1 \cdots \tau_n)$, this proves the case for variables.

Case $\mathit{let}\ y : \forall(\langle t_1{::}k_1\rangle).\tau_1 = M_1^1\ \mathit{in}\ M_1^2$. Suppose

$$\mathcal{K}, \mathcal{T}\{x : \forall(\langle t_0{::}k_0\rangle).\tau_0\} \vdash \mathit{let}\ y : \forall(\langle t{::}k\rangle).\tau_1 = M_1^1\ \mathit{in}\ M_1^2 \,:\, \tau$$

We must have

$$\mathcal{K}\{\langle t{::}k\rangle\}, \mathcal{T}\{x : \forall(\langle t_0{::}k_0\rangle).\tau_0\} \vdash M_1^1 \,:\, \tau_1$$

and

$$\mathcal{K}, \mathcal{T}\{x : \forall(\langle t_0{::}k_0\rangle).\tau_0\}\{y : \forall(\langle t{::}k\rangle).\tau_1\} \vdash M_1^2 \,:\, \tau$$

By Lemma 3.4, $\mathcal{K}\{\langle t{::}k\rangle\}\{\langle t_0{::}k_0\rangle\}, \mathcal{T} \vdash M_2 \,:\, \tau_0$. Then by the induction hypothesis,

$$\mathcal{K}\{\langle t{::}k\rangle\}, \mathcal{T} \vdash [(\langle t_0\rangle).M_2/x]M_1^1 \,:\, \tau_1$$

By bound variable convention, we can assume that $x \neq y$. By Lemma 3.2, $\mathcal{K}\{\langle t_0 :: k_0\rangle\}, \mathcal{T}\{y : \forall(\langle t::k\rangle).\tau_1\} \vdash M_2 : \tau_0$. Applying the induction hypothesis to M_1^2, we have

$$\mathcal{K}, \mathcal{T}\{y : \forall(\langle t::k\rangle).\tau_1\} \vdash [(\langle t_0\rangle).M_2/x]M_1^2 : \tau$$

Then by the rule (LET) we have

$$\mathcal{K}, \mathcal{T} \vdash let\ y : \forall(\langle t::k\rangle).\tau_1 = [(\langle t_0\rangle).M_2/x]M_1^1\ in\ [(\langle t_0\rangle).M_2/x]M_1^2 : \tau$$

as desired. ∎

Using these properties, we can prove the following subject reduction theorem.

Theorem 4.4 *If $\mathcal{K}, \mathcal{T} \vdash M : \tau$ and $M \xrightarrow{*} N$ then $\mathcal{K}, \mathcal{T} \vdash N : \tau$.*

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

$$\{x_1 : \sigma_1 = \Lambda(\langle t_1 :: k_1\rangle).M_1, \ldots, x_n : \sigma_n = \Lambda(\langle t_n :: k_n\rangle).M_n\}$$

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.

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

4.2 Coherent Type Reconstruction

In this subsection, we present a coherent reconstruction of a Λ^{ml} declaration from a λ^{ml} declaration.

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 4.5 *Let $\mathcal{T}$ be a type assignment that does not contain any polytypes, 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 [29]. ∎

The pure calculus Λ^{ml} is strongly normalizing. This is seen from the following observation. For each well-typed term in the pure calculus Λ^{ml} we can construct a well-typed term in System F by replacing each kind with the kind U and un-nesting type abstraction and type applications. Moreover, the constructed System F term can simulates the reduction of the original term. Then Theorem 4.4 and Lemma 4.5 imply the following.

Corollary 4.6 *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 $M_1 \stackrel{*}{\longleftrightarrow} M_2$.*

In particular, all the Λ^{ml} typings corresponding to a given λ^{ml} typing of the form $\mathcal{K}, \emptyset \vdash e : \tau$ 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. Proposition 3.7 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 achieve coherent type reconstruction. Corollary 4.6 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.

The above 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} declaration differ only in their type annotations, so this amounts to choosing canonical type annotations. The desired property for canonical type annotation is that they do not contain type variables that are not "involved" in any type in the given typing judgment. As we noted earlier, those type variables will not be instantiated and therefore the type system cannot determine specialization information for the terms containing those type variables.

With kind constraints, a type may indirectly involve some type variables other than its own 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 $FTV(k) \subseteq EFTV(\mathcal{K}, \tau)$.

For example, $t_2 \in EFTV(\{t_1::\{\!\{l : t_2\}\!\}\}, t_1)$. We also define $EFTV(\mathcal{K}, \mathcal{T}) = \bigcup\{EFTV(\mathcal{K}, \tau) | x : \tau \in \mathcal{T}\}$. 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 $EFTV(\mathcal{K}, \mathcal{T}) \cup EFTV(\mathcal{K}, \tau)$.

To eliminate vacuous type variables and to define a desirable canonical type annotations, we make the following additional assumption on a given kind structure:

> for any kind k there is a type τ_k such that if $\mathcal{K} \vdash k$ then $\mathcal{K} \vdash \tau_k :: k$.

We believe that this condition is satisfied by most of kind structures. For example, if k is the record $\{\!\{l : \tau\}\!\}$, we can take τ_k to be the record type $\{l : \tau\}$. We then have the following.

Proposition 4.7 *If $\mathcal{K}, \mathcal{T} \vdash e : \tau$ then there is some $\mathcal{K}'$ such that $dom(\mathcal{K}') = EFTV(\mathcal{K}, \mathcal{T}) \cup EFTV(\mathcal{K}, \tau)$, and $\mathcal{K}', \mathcal{T} \vdash e : \tau$.*

This is an immediate consequence of the above assumption on kinding and Lemma 3.5. This guarantees that for any typing there is a typing derivation that does not contain vacuous type variables. Among possible Λ^{ml} terms corresponding to a given λ^{ml} declaration, we choose one that does not contain vacuous type variables.

Using this property, we define an algorithm to transform a derivation of a λ^{ml} declaration to a Λ^{ml} declaration as follows.

1. The translation of $\emptyset$ is $\emptyset$.

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

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

In the above algorithm, if a given typing derivation for $\mathcal{K}, \mathcal{T} \vdash e : \tau$ does not contain vacuous type variable, then the elimination of vacuous type variables in M is of course unnecessary. However, this is very unlikely, since all the type inference algorithms for ML the author aware of produce a typing derivation containing vacuous type variables. Proposition 3.7 and Corollary 4.6 guarantee that the application of substitution $[\tau_{k'_1}/t_1, \ldots . \tau_{k'_n}/t_n]$ does not change the meaning of the term. 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 [31] and for size sensitive lambda binding [32]. 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 languages.

5.1 Compilation of Polymorphic Record Operations

Here we only consider the following minimal set of terms

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

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

$$\begin{array}{rcl} \tau & ::= & b \mid t \mid \tau \to \tau \mid \{l : \tau, \ldots, l : \tau\} \\ k & ::= & U \mid \{\!\{l : \tau, \ldots, l : \tau\}\!\} \end{array}$$

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 typing rule for polymorphic field selection is given as follows.

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

The following is an example of typing.

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

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

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

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

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

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

Figure 5: Some of Typing Rules of the Polymorphic Record Calculus

The goal of specializing a polymorphic record calculus is to establish a method to compile labeled field selection into indexing operation. To formally define this process, we define the implementation calculus with vectors and index operation.

The set of indexes is given by the following syntax

$$\mathcal{I} ::= n \mid I$$

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

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

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.

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

$index(l, \tau)$ is an *index type* denoting the singleton set of 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 computed statically, 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* $\mathcal{A}$ is a function from a finite set of index variables to index types. We write $\mathcal{A} \vdash \mathcal{I} \,:\, index(l, \tau)$ if $\mathcal{I}$ is the index value of l in type τ. Under the assumption that the fields in a record type are sorted by labels, this relation is given by the following rules.

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

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

The typing rules (other than the standard rules of Λ^{ml}) are given in Figure 5.

The following theorem is proved in [31].

Theorem 5.1 *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.

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

When this function is let-bound and used for some instance type, the necessary index value is computed statically and inserted. This is done by using the information encoded in the singleton type $index(l, t_2)$. 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

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

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

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

As mentioned in Introduction, a crucial information for unboxed semantics is the size of values. To implement lambda abstraction (and other size sensitive operations such as second projection) efficiently, we 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 that require size information when they are compiled into size sensitive efficient code. This property is represented by the following typing rules.

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

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

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 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.

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

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

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

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

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

Figure 6: Some of Typing Rules of the Unboxed Calculus

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 singleton set of 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.

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

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

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

Some of typing rules are given in Figure 6.

The following theorem is proved in [32].

Theorem 5.2 *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.

$$\{t :: S\}, \emptyset, \emptyset \vdash \delta I.\lambda^I x.x \,:\, size(t) \Rightarrow t \to t$$

When this function is let-bound and used for some instance type τ for t, the necessary size value is computed statically from the singleton type $size(\tau)$ and a size application is inserted. For example, from

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

the translator will produce the following code

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

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

From these analyses, we observe the following common structure in type-directed specialization of polymorphism.

- A polymorphic primitive is decomposed into a low-level generic operation and a type attribute that is required for executing the operation on the type.
- Kinding information determines what sort of type attributes are required for specialization. For example, if the source program has a typing of the form

  $$\{t::k_\pi\}, \emptyset \vdash e \ : \ \tau$$

  for some kind k_π associated with a polymorphic operation π, then the type system determines that e performs operation π on values having type t_1, and translates this typing to the following typing by introducing an auxiliary variable I as follows

  $$\{t::k_\pi\}, \{I : P_{k_\pi}(t)\}, \emptyset \vdash C_e \ : \ \tau$$

  where $P_{k_\pi}(t)$ is the type attribute needed to specialize the operation π for the type t.
- Polymorphism is recovered by introducing an abstraction mechanism over attributes that must be specialized. For example, when the above term is let-bound, it is used as the following polymorphic function.

  $$\emptyset, \emptyset, \emptyset \vdash \delta I.C_e \ : \ \forall t::k_\pi.P_{k_\pi}(t) \Rightarrow \tau$$
- The type system computes necessary attributes values statically by treating a static computation of an attribute value itself as a type denoting the singleton set of the result of static computation. In the above example, if t is instantiated to τ_0, then the type of the function becomes $P_{k_\pi}(\tau_0) \Rightarrow [\tau_0/t]\tau$, where $P_{k_\pi}(\tau_0)$ is a singleton type denoting the attribute of type τ_0, and the type system can statically compute the value needed to specialize π from the type $P_{k_\pi}(\tau_0)$.

By developing a typed implementation language 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-directed specialization exploited in the above two examples to a wide range of polymorphic primitives.

Type-directed specialization analyzed above is related to *intensional polymorphism* by Harper and Morrisett [13], 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. 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 mechanism alone does not directly produce efficient code, and some optimization must also be incorporated for producing efficient code. Of course, one can expect that most of the static computation could somehow be carried out at compile time or perhaps at link time. However, incorporating those optimization in an ad-hoc manner is far from trivial. As summarized above, one distinguishing feature of our type-directed specialization is to provide a type-theoretical basis for statically computing those attributes of a type that are relevant for efficient execution, and for passing only those relevant values at run-time.

The methods for compiling overloading such as [33] also use a mechanism similar to one developed for record polymorphism [30, 31]. In these systems, the compiler passes appropriate monomorphic version of functions determined by the static type information. However, again, they do not fully address the issue of specialization of polymorphism into efficient code.

In revising this article, the author noticed that Crary, Weirich, and Morriset [7] later proposed a type system for type-passing semantics, which contains the mechanisms similar to those developed in the present article and in [30, 31].

6 A Framework for Type-Directed Specialization

Based on the observations in the previous section, we develop a framework for type-directed specialization.

6.1 The Source Language

The source language is the explicitly typed kinded language Λ^{ml} extended with data structures and polymorphic operations. We assume that there is a given set of data constructors (ranged over by f) with the associated type constructors (ranged over by F.) We also extend Λ^{ml} with polymorphic term constructors (ranged over by π). To define a typing rule for π, we introduce the following notation. Let $T[X_1, \ldots, X_n]$ ($K[X_1, \ldots, X_n]$) be a closed monotype (a closed atomic kind) possibly containing special unknown symbols $X_1, \ldots, X_n$, and write $T[\tau_1, \ldots, \tau_n]$ ($K[\tau_1, \ldots, \tau_n]$) for the type (kind) obtained from $T[X_1, \ldots, X_n]$ ($K[X_1, \ldots, X_n]$) by substituting each τ_i for X_i. We assume that for each polymorphic term constructor there are associated type expressions $T^1_\pi[X_1, \ldots, X_n]$, $T^2_\pi[X_1, \ldots, X_n]$, $T^3_\pi[X_1, \ldots, X_n]$, and a kind expression $K_\pi[X_1, \ldots, X_n]$. Using these notations, the source language is extended with the following typing rules for each data constructor f and polymorphic construct π.

$$(\textsc{data})\ \frac{\mathcal{K}, \mathcal{T} \vdash M_i : \tau_i\ (1 \leq i \leq n)}{\mathcal{K}, \mathcal{T} \vdash f(M_1, \ldots, M_n) : F(\tau_1, \ldots, \tau_n)}$$

$$(\pi)\ \frac{\mathcal{K}, \mathcal{T} \vdash M : T^1_\pi[\tau_1, \ldots, \tau_n] \qquad \mathcal{K} \vdash T^2_\pi[\tau_1, \ldots, \tau_n] :: K_\pi[\tau_1, \ldots, \tau_n]}{\mathcal{K}, \mathcal{T} \vdash \pi(M, T^2_\pi[\tau_1, \ldots, \tau_n], K_\pi[\tau_1, \ldots, \tau_n]) : T^3_\pi[\tau_1, \ldots, \tau_n]}$$

The rule (DATA) is for a constructor that does not require any kind constraints. A data constructor that requires kind constraint and subsequent specialization such as polymorphic variant is modeled by the rule (π). In the rule (π), the term in the conclusion is annotated so that it encodes the derivation in the term. This is only needed to defining a specialization algorithm as an algorithm to transform an Λ^{ml} term to an Λ^{impl} term, so we omit the type annotation in π constructor until we describes a specialization algorithm.

We believe that this form of rule scheme is general enough to represent most of polymorphic operations. As an example, the typing rule for polymorphic field selection is obtained by letting $T^1_\pi[\tau_1, \tau_2] = \tau_1$, $T^2_\pi[\tau_1, \tau_2] = \tau_1$, $T^3_\pi[\tau_1, \tau_2] = \tau_2$, $K_\pi[\tau_1, \tau_2] = \{\!\{l : \tau_2\}\!\}$, and $\pi(M) = M{\cdot}l$. The typing rule for unboxed abstraction can also be regarded as a special case of this rule by considering it to be a combination of generating an intermediate abstraction $(x : \tau).M$ of some special type $abs(\tau, \tau')$ which can only be used as an argument to the polymorphic operator with the typing rule specified as $T^1_\pi[\tau_1, \tau_2] = abs(\tau_1, \tau_2)$, $T^2_\pi[\tau_1, \tau_2] = \tau_1$, $T^3_\pi[\tau_1, \tau_2] = \tau_1 \to \tau_2$, $K_\pi[\tau_1, \tau_2] = S$, and $\pi((x : \tau_1).M) = \lambda x : \tau_1.M$.

6.2 The Implementation Language : Λ^{impl}

As explained earlier, a polymorphic primitive $\pi(M)$ is compiled to 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. Furthermore, this attribute value is computed statically from the type of the argument. To represent such operation, we assume that for each atomic kind $K_\pi[\tau_1, \ldots, \tau_n]$ introduced for a primitive operation π, there is an associated type attribute $P_{K_\pi[\tau_1,\ldots,\tau_n]}$ such that for any type τ of kind $k_\pi[\tau_1, \ldots, \tau_n]$, $P_{K_\pi[\tau_1,\ldots,\tau_n]}(\tau)$ denotes the unique attribute value of type τ. We further assume that if τ is not a type variable then $P_{K_\pi[\tau_1,\ldots,\tau_n]}(\tau)$ denotes a unique constant value determined by the topmost type constructor of τ. The rationale of this restriction is to limit computation of type attribute at compile time so that polymorphic specialization can be realized by passing constant values at run-time. Without this restriction, some mechanism to create and pass partially computed attributes will be needed. We write $|P_{K_\pi[\tau_1,\ldots,\tau_n]}(\tau)|$ for the unique value denoted by $P_{K_\pi[\tau_1,\ldots,\tau_n]}(\tau)$. If M has type τ and $c = |P_{K_\pi[\tau_1,\ldots,\tau_n]}(\tau)|$, then the polymorphic construct $\pi(M)$ is compiled to $C_\pi(C_M, c)$, where C_M is the compiled term of M. For example, for the filed selection, we can take $P_{\{\!\{l:\tau_1\}\!\}}(\tau) = index(l, \tau)$ and $C_\pi(C_M, \mathcal{I}) = C_M[\mathcal{I}]$ where $\mathcal{I}$ is a value of the type attribute $index(l, \tau)$. For the unboxed abstraction, we can take $P_S(\tau) = size(\tau)$ and $C_\pi(C_M, \mathcal{I}) = \lambda^{\mathcal{I}} C_M$ where $\mathcal{I}$ is a value of the type attribute $size(\tau)$. We shall formalize this representation as a typing rule of Λ^{impl} below. A key feature of the implementation language is to treat a type attribute of the form $P_{K_\pi[\tau_1,\ldots,\tau_n]}(\tau)$ as a type denoting the singleton set of the attribute value itself, and to introduce an abstraction mechanism over attributes. We call those types *attribute types* and use α for a meta variable ranging over attribute types.

We introduce a set of *attribute variables* (ranged over by I). The set of *attribute terms* (ranged over by a) is given by the following syntax.

$$a ::= c \mid I$$

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

$$C ::= c^b \mid (x\ \langle\tau\rangle\ \langle a\rangle) \mid \lambda x : \tau.C \mid C\ C \mid f(C, \ldots, C) \mid C_\pi(C, a) \mid let\ x : \sigma = \Lambda(\langle t::k\rangle).(\langle I : \alpha\rangle).C\ in\ C$$

$(x\ \langle\tau\rangle\ \langle a\rangle)$ is a variable with type instantiation and attribute application. In $let\ x : \sigma = \Lambda(\langle t::k\rangle).(\langle I : \alpha\rangle).C_1\ in\ C_2$, type variables $\langle t\rangle$ and attribute variables $\langle I\rangle$ in C_1 are abstracted. We write $[a_1/I_1, \ldots, a_m/I_m]C$ for the term obtained from C by substituting a_i for I_i. Term substitution is refined to the operator that integrates type instantiation and attribute substitution. We write $[(t_1, \ldots, t_n).(I_1, \ldots, I_m).C/x]C'$ for the term obtained from C' by substituting $[a_1/I_1, \ldots, a_m/I_m]([\tau_1/t_1, \ldots, \tau_n/t_n]C)$ for each occurrence of the form $(x\ \tau_1 \cdots \tau_n\ a_1 \cdots a_m)$. Its formal definition is obtained by extending the following clauses inductively to other term constructors.

$$\begin{aligned}
[(t_1, \ldots, t_n).(I_1, \ldots, I_m).C/x](x\ \tau_1 \cdots \tau_n\ a_1 \cdots a_m) &= [a_1/I_1, \ldots, a_m/I_m]([\tau_1/t_1, \ldots, \tau_n/t_n]C) \\
[(t_1, \ldots, t_n).(I_1, \ldots, I_m).C/x](y\ \tau_1 \cdots \tau_m\ a_1 \cdots a_l) &= (y\ \tau_1 \cdots \tau_m\ a_1 \cdots a_l)
\end{aligned}$$

If $n = 0$ and $m = 0$ then we simply write $[C/x]C'$ instead of writing $[().().C/x]C'$.

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

(β) $(\lambda x : \tau.C)\ C' \Longrightarrow [C'/x]C$

(LET) $let\ x : \sigma = \Lambda(\langle t::k\rangle).(\langle I : \alpha\rangle).C\ in\ C' \Longrightarrow [(\langle t\rangle).(\langle I\rangle).C/x]C'$

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

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

$$\mathcal{A} \vdash a \,:\, \alpha$$

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

(CONST) $\mathcal{A} \vdash c \,:\, P_{K_\pi[\tau_1,\ldots,\tau_n]}(\tau)$ if τ is not a type variable and $|P_{K_\pi[\tau_1,\ldots,\tau_n]}(\tau)| = c$

(AVAR) $\mathcal{A}\{I : \alpha\} \vdash I \,:\, \alpha$

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

$$\begin{array}{rcl} \tau & ::= & t \mid b \mid \tau \to \tau \mid F(\tau, \ldots, \tau) \\ \sigma & ::= & \forall(\langle t::k\rangle).(\langle\alpha\rangle) \Rightarrow \tau \end{array}$$

Note that polymorphic type abstraction and attribute abstraction are integrated in the definition of polytypes. If both $\langle t::k\rangle$ and $\langle\alpha\rangle$ in $\forall(\langle t::k\rangle).(\langle\alpha\rangle) \Rightarrow \tau$ are empty then the polytype is identified with τ.

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

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

The set of typing rules is given in Figure 7.

For this language, the following three substitution lemmas hold.

Lemma 6.1 *If* $\mathcal{K}\{t_0::k_0\}\mathcal{K}', \mathcal{A}, \mathcal{T} \vdash C \,:\, \tau$ *and* $\mathcal{K} \vdash \tau_0 :: k_0$ *then*
$\mathcal{K}([\tau_0/t_0]\mathcal{K}'), [\tau_0/t_0]\mathcal{A}, [\tau_0/t_0]\mathcal{T} \vdash [\tau_0/t_0]C \,:\, [\tau_0/t_0]\tau$.

Proof By induction on C. Here we only show the cases for variables, polymorphic operations and let expressions.

Case $(x\ \tau_1 \cdots \tau_n\ a_1 \cdots a_m)$. Suppose

$$\mathcal{K}\{t_0::k_0\}\mathcal{K}', \mathcal{A}, \mathcal{T}\{x : \forall(t_1::k_1, \ldots, t_n::k_n).(\alpha_1, \ldots, \alpha_m) \Rightarrow \tau'\} \vdash (x\ \tau_1 \cdots \tau_n\ a_1 \cdots a_m) \,:\, \tau$$

Then we must have

$$\begin{array}{l} \mathcal{K}\{t_0::k_0\}\mathcal{K}' \vdash \tau_i :: [\tau_1/t_1, \ldots, \tau_{i-1}/t_{i-1}]k_i, \\ \mathcal{K}\{t_0::k_0\}\mathcal{K}' \vdash \mathcal{T}\{x : \forall(t_1::k_1, \ldots, t_n::k_n).(\alpha_1, \ldots, \alpha_m) \Rightarrow \tau'\}, \\ \mathcal{K}\{t_0::k_0\}\mathcal{K}' \vdash \mathcal{A}, \\ \mathcal{A} \vdash a_i \,:\, [\tau_1/t_1, \ldots, \tau_n/t_n](\alpha_i), \text{ and} \\ \tau = [\tau_1/t_1, \ldots, \tau_n/t_n]\tau' \end{array}$$

By the assumption of kinding and the bound type variable convention, we have

$$\mathcal{K}([\tau_0/t_0]\mathcal{K}') \vdash [\tau_0/t_0]\tau_i :: [[\tau_0/t_0]\tau_1/t_1, \ldots, [\tau_0/t_0]\tau_{i-1}/t_{i-1}]([\tau_0/t_0]k_i)$$

By the definition of well-formedness of types and the bound type variable convention, we have

$$\begin{array}{l} \mathcal{K}([\tau_0/t_0]\mathcal{K}') \\ \quad \vdash \ ([\tau_0/t_0]\mathcal{T})\{x : \forall(t_1::[\tau_0/t_0]k_1, \ldots, t_n::[\tau_0/t_0]k_n).([\tau_0/t_0]\alpha_1, \ldots, [\tau_0/t_0]\alpha_m) \Rightarrow ([\tau_0/t_0]\tau_0)\} \end{array}$$

$$\text{(CONST)}\quad \mathcal{K}, \mathcal{A}, \mathcal{T} \vdash c^b \,:\, b \quad \text{if } \mathcal{K} \vdash \mathcal{T} \text{ and } \mathcal{K} \vdash \mathcal{A}$$

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

$$\text{(APP)}\quad \frac{\mathcal{K}, \mathcal{A}, \mathcal{T} \vdash C_1 \,:\, \tau_1 \to \tau_2 \quad \mathcal{K}, \mathcal{A}, \mathcal{T} \vdash C_2 \,:\, \tau_1}{\mathcal{K}, \mathcal{A}, \mathcal{T} \vdash C_1\ C_2 \,:\, \tau_2}$$

$$\text{(ABS)}\quad \frac{\mathcal{K}, \mathcal{A}, \mathcal{T}\{x : \tau_1\} \vdash C_1 \,:\, \tau_2}{\mathcal{K}, \mathcal{A}, \mathcal{T} \vdash \lambda x : \tau_1.C_1 \,:\, \tau_1 \to \tau_2}$$

$$\text{(DATA)}\quad \frac{\mathcal{K}, \mathcal{A}, \mathcal{T} \vdash C_i \,:\, \tau_i\ (1 \le i \le n)}{\mathcal{K}, \mathcal{A}, \mathcal{T} \vdash f(C_1, \ldots, C_n) \,:\, F(\tau_1, \ldots, \tau_n)}$$

$$(C_\pi)\quad \frac{\begin{array}{l}\mathcal{K}, \mathcal{A}, \mathcal{T} \vdash C \,:\, T^1_\pi[\tau_1, \ldots, \tau_n] \\ \mathcal{K} \vdash T^2_\pi[\tau_1, \ldots, \tau_n] :: K_\pi[\tau_1, \ldots, \tau_n] \\ \mathcal{A} \vdash a \,:\, P_{K_\pi[\tau_1, \ldots, \tau_n]}(T^2_\pi[\tau_1, \ldots, \tau_n])\end{array}}{\mathcal{K}, \mathcal{A}, \mathcal{T} \vdash C_\pi(C, a) \,:\, T^3_\pi[\tau_1, \ldots, \tau_n]}$$

$$\text{(LET)}\quad \frac{\begin{array}{l}\mathcal{K}\{\langle t{::}k\rangle\}, \mathcal{A}\{\langle I : \alpha\rangle\}, \mathcal{T} \vdash C_1 \,:\, \tau_1 \\ \mathcal{K}, \mathcal{A}, \mathcal{T}\{x : \forall(\langle t{::}k\rangle).(\langle\alpha\rangle) \Rightarrow \tau_1\} \vdash C_2 \,:\, \tau_2\end{array}}{\mathcal{K}, \mathcal{A}, \mathcal{T} \vdash let\ x : \forall(\langle t{::}k\rangle).(\langle\alpha\rangle) \Rightarrow \tau_1 = \Lambda(\langle t{::}k\rangle).(\langle I : \alpha\rangle).C_1\ in\ C_2 \,:\, \tau_2}$$

Figure 7: The Type System of the Implementation Language Λ^{impl}

By the definition of well-formedness of types, the bound type variable convention, and the definition of attribute judgments, we have

$$\begin{array}{l}\mathcal{K}([\tau_0/t_0]\mathcal{K}') \vdash [\tau_0/t_0]\mathcal{A} \\ [\tau_0/t_0]\mathcal{A} \vdash a_i \,:\, [[\tau_0/t_0]\tau_1/t_1, \ldots, [\tau_0/t_0]\tau_n/t_n]([\tau_0/t_0]\alpha_i)\end{array}$$

Then by the rule (VAR) we have

$$\begin{array}{l}\mathcal{K}([\tau_0/t_0]\mathcal{K}'), [\tau_0/t_0]\mathcal{A}, ([\tau_0/t_0]\mathcal{T}\{x : \forall(t_1{::}k_1, \ldots, t_n{::}k_n).(\alpha_1, \ldots, \alpha_m) \Rightarrow \tau' \\ \quad \vdash \quad (x\ [\tau_0/t_0]\tau_1 \cdots [\tau_0/t_0]\tau_n\ a_1 \cdots a_m) \,:\, [[\tau_0/t_0]\tau_1/t_1, \ldots, [\tau_0/t_0]\tau_n/t_n]([\tau_0/t_0]\tau')\end{array}$$

as desired.

Case $C_\pi(C, a)$. Suppose $\mathcal{K}\{t_0{::}k_0\}\mathcal{K}', \mathcal{A}, \mathcal{T} \vdash C_\pi(C, a) \,:\, T^3_\pi[\tau_1, \ldots, \tau_n]$. Then we must have

$$\begin{array}{l}\mathcal{K}\{t_0{::}k_0\}\mathcal{K}', \mathcal{A}, \mathcal{T} \vdash C \,:\, T^1_\pi[\tau_1, \ldots, \tau_n], \\ \mathcal{K}\{t_0{::}k_0\}\mathcal{K}' \vdash T^2_\pi[\tau_1, \ldots, \tau_n] :: K_\pi[\tau_1, \ldots, \tau_n], \text{ and} \\ \mathcal{A} \vdash a \,:\, P_{K_\pi[\tau_1, \ldots, \tau_n]}(T^2_\pi[\tau_1, \ldots, \tau_n]).\end{array}$$

By the induction hypothesis,

$$\mathcal{K}([\tau_0/t_0]\mathcal{K}'), [\tau_0/t_0]\mathcal{A}, [\tau_0/t_0]\mathcal{T} \vdash [\tau_0/t_0]C \,:\, T^1_\pi[[\tau_0/t_0]\tau_1, \ldots, [\tau_0/t_0]\tau_n]$$

Since T_2, K_π does not contain type variables, by the assumption on kindings,

$$\mathcal{K}([\tau_0/t_0]\mathcal{K}') \vdash T^2_\pi[[\tau_0/t_0]\tau_1, \ldots, [\tau_0/t_0]\tau_n] :: K_\pi[[\tau_0/t_0]\tau_1, \ldots, [\tau_0/t_0]\tau_n].$$

Since an attribute value is determined by the top-most type constructor, by the definition of attribute typings,

$$[\tau_0/t_0]\mathcal{A} \vdash a \ : \ P_{K_\pi[[\tau_0/t_0]\tau_1,\ldots,[\tau_0/t_0]\tau_n]}(T^2_\pi[[\tau_0/t_0]\tau_1, \ldots, [\tau_0/t_0]\tau_n]).$$

Since the rule (C_π) is a rule schema, all the rule instances obtained by substituting τ_0 for t_0 are also valid inference rules. We therefore have the following.

$$\mathcal{K}([\tau_0/t_0]\mathcal{K}'), [\tau_0/t_0]\mathcal{A}, [\tau_0/t_0]\mathcal{T} \vdash C_\pi([\tau_0/t_0]C, a) \ : \ T^3_\pi[[\tau_0/t_0]\tau_1, \ldots, [\tau_0/t_0]\tau_n]$$

Case for let expressions. Suppose

$$\mathcal{K}\{t_0{::}k_0\}\mathcal{K}', \mathcal{A}, \mathcal{T} \vdash \mathit{let}\ x : \forall(\langle t{::}k\rangle).(\langle\alpha\rangle) \Rightarrow \tau_1 = \Lambda(\langle t{::}k\rangle).(\langle I : \alpha\rangle).C_1\ \mathit{in}\ C_2 \ : \ \tau_2$$

We must have the following

$$\begin{array}{l}\mathcal{K}\{t_0{::}k_0\}\mathcal{K}'\{\langle t{::}k\rangle\}, \mathcal{A}\{\langle I : \alpha\rangle\}, \mathcal{T} \vdash C_1 \ : \ \tau_1 \\ \mathcal{K}\{t_0{::}k_0\}\mathcal{K}', \mathcal{A}, \mathcal{T}\{x : \forall(\langle t{::}k\rangle).(\langle\alpha\rangle) \Rightarrow \tau_1\} \vdash C_2 \ : \ \tau_2\end{array}$$

By the induction hypotheses,

$$\mathcal{K}([\tau_0/t_0]\mathcal{K}')\{\langle t{::}[\tau_0/t_0]k\rangle\}, [\tau_0/t_0]\mathcal{A}\{\langle I : [\tau_0/t_0]\alpha\rangle\}, [\tau_0/t_0]\mathcal{T} \vdash [\tau_0/t_0]C_1 \ : \ [\tau_0/t_0]\tau_1$$

$$\mathcal{K}([\tau_0/t_0]\mathcal{K}'), [\tau_0/t_0]\mathcal{A}, [\tau_0/t_0]\mathcal{T}\{x : \forall(\langle t{::}[\tau_0/t_0]k\rangle).(\langle[\tau_0/t_0]\alpha\rangle) \Rightarrow [\tau_0/t_0]\tau_1\} \vdash [\tau_0/t_0]C_2 \ : \ [\tau_0/t_0]\tau_2$$

By the rule (LET), we have

$$\begin{array}{lll}\mathcal{K}([\tau_0/t_0]\mathcal{K}'), [\tau_0/t_0]\mathcal{A}, [\tau_0/t_0]\mathcal{T} & \vdash & \mathit{let}\ x : \forall(\langle t{::}[\tau_0/t_0]k\rangle).(\langle[\tau_0/t_0]\alpha\rangle) \Rightarrow [\tau_0/t_0]\tau_1 \\ & & \quad = \Lambda(\langle t{::}[\tau_0/t_0]k\rangle).(\langle I : [\tau_0/t_0]\alpha\rangle).[\tau_0/t_0]C_1 \\ & & \mathit{in}\ [\tau_0/t_0]C_2 \\ & : & [\tau_0/t_0]\tau_2\end{array}$$

■

Lemma 6.2 *If* $\mathcal{K}, \mathcal{A}\{\langle I_0 : \alpha_0\rangle\}, \mathcal{T} \vdash C \ : \ \tau$ *and* $\mathcal{A} \vdash \langle a_0\rangle \ : \ \langle\alpha_0\rangle$ *then* $\mathcal{K}, \mathcal{A}, \mathcal{T} \vdash [\langle a_0/I_0\rangle]C \ : \ \tau$.

Proof By induction on C. The only interesting cases are C_π and let expression. Other cases follow directly from the induction hypotheses.

Case $C_\pi(C, a)$. Suppose $\mathcal{K}, \mathcal{A}\{\langle I_0 : \alpha_0\rangle\}, \mathcal{T} \vdash C_\pi(C, a) \ : \ T^3_\pi[\tau_1, \ldots, \tau_n]$. Then we must have

$$\begin{array}{l}\mathcal{K}, \mathcal{A}\{\langle I_0 : \alpha_0\rangle\}, \mathcal{T} \vdash C \ : \ T^1_\pi[\tau_1, \ldots, \tau_n], \\ \mathcal{K} \vdash T^2_\pi[\tau_1, \ldots, \tau_n] :: K_\pi[\tau_1, \ldots, \tau_n], \text{ and} \\ \mathcal{A}\{\langle I_0 : \alpha_0\rangle\} \vdash a \ : \ P_{K_\pi[\tau_1,\ldots,\tau_n]}(T^2_\pi[\tau_1, \ldots, \tau_n]).\end{array}$$

By the induction hypothesis, $\mathcal{K}, \mathcal{A}, \mathcal{T} \vdash [\langle a_0/I_0\rangle]C \ : \ T^1_\pi[\tau_1, \ldots, \tau_n]$. If a is not among $\langle I_0\rangle$, then by definition of attribute typing, $\mathcal{A} \vdash a \ : \ P_{K_\pi[\tau_1,\ldots,\tau_n]}(T^2_\pi[\tau_1, \ldots, \tau_n])$. If a is $I_i \in \langle I_0\rangle$, then

$\alpha_i = P_{K_\pi[\tau_1,\ldots,\tau_n]}(T^2_\pi[\tau_1,\ldots,\tau_n])$ and therefore $\mathcal{A} \vdash [\langle a_0/I_0\rangle]a : P_{K_\pi[\tau_1,\ldots,\tau_n]}(T^2_\pi[\tau_1,\ldots,\tau_n])$. In either case, by the rule (C_π) we have $\mathcal{K},\mathcal{A},\mathcal{T} \vdash [\langle a_0/I_0\rangle](C_\pi(C,a)) : T^3_\pi[\tau_1,\ldots,\tau_n]$.

Case for let expression. Suppose

$$\mathcal{K},\mathcal{A}\{\langle I_0 : \alpha_0\rangle\},\mathcal{T} \vdash let\ x : \forall(\langle t::k\rangle).(\langle\alpha\rangle) \Rightarrow \tau_1 = \Lambda(\langle t::k\rangle).(\langle I : \alpha\rangle).C_1\ in\ C_2 : \tau_2$$

By the bound variable convention for bound attribute variables, we can assume that $\langle I\rangle \cap \langle I_0\rangle = \emptyset$. By the typing rule, we must have $\mathcal{K}\{\langle t::k\rangle\},\mathcal{A}\{\langle I_0 : \alpha_0\rangle\}\{\langle I : \alpha\rangle\},\mathcal{T} \vdash C_1 : \tau_1$ and $\mathcal{K},\mathcal{A}\{\langle I_0 : \alpha_0\rangle\},\mathcal{T}\{x : \forall(\langle t::k\rangle).(\langle\alpha\rangle) \Rightarrow \tau_1\} \vdash C_2 : \tau_2$. By the induction hypothesis, $\mathcal{K}\{\langle t::k\rangle\},\mathcal{A}\{\langle I : \alpha\rangle\},\mathcal{T} \vdash [\langle a_0/I_0\rangle]C_1 : \tau_1$ and $\mathcal{K},\mathcal{A},\mathcal{T}\{x : \forall(\langle t::k\rangle).(\langle\alpha\rangle) \Rightarrow \tau_1\} \vdash [\langle a_0/I_0\rangle]C_2 : \tau_2$. By the rule (LET), we have

$$\mathcal{K},\mathcal{A},\mathcal{T} \vdash let\ x : \forall(\langle t::k\rangle).(\langle\alpha\rangle) \Rightarrow \tau_1 = \Lambda(\langle t::k\rangle).(\langle I : \alpha\rangle).[\langle a_0/I_0\rangle]C_1\ in\ [\langle a_0/I_0\rangle]C_2 : \tau_2$$

■

Lemma 6.3 *If* $\mathcal{K}\{\langle t_0::k_0\rangle\},\mathcal{A}\{\langle I_0 : \alpha_0\rangle\},\mathcal{T} \vdash C_2 : \tau_0$ *and* $\mathcal{K},\mathcal{A},\mathcal{T}\{x : \forall(\langle t_0::k_0\rangle).(\langle\alpha_0\rangle) \Rightarrow \tau_0\} \vdash C_1 : \tau$, *then* $\mathcal{K},\mathcal{A},\mathcal{T} \vdash [(\langle t_0\rangle).(\langle I_0\rangle).C_2/x]C_1 : \tau$.

Proof By induction on C_1. We only show the cases for variables and let expressions. Other cases can easily be show using the corresponding induction hypothesis.

Case $(y\ \tau_1\cdots\tau_n\ a_1\cdots a_m)$. If $x \neq y$, then the result follows by the free variable lemma (Lemma 3.2) for this language. Let $x = y$ and suppose

$$\mathcal{K},\mathcal{A},\mathcal{T}\{x : \forall(t_0^1::k_0^1,\ldots,t_0^n::k_0^n).(\alpha_0^1,\ldots,\alpha_0^m) \Rightarrow \tau_0\} \vdash (x\ \tau_1\cdots\tau_n\ a_1\cdots a_m) : [\tau_1/t_0^1,\ldots,\tau_n/t_0^n]\tau_0$$

Let $S_i = [\tau_1/t_0^1,\ldots,\tau_i/t_0^i]$. By the type system, we must have $\mathcal{K} \vdash \tau_i :: S_{i-1}(k_0^i)$ for $(1 \leq i \leq n)$. By the bound type variable convention, we can assume that no t_0^i is free in $\tau_j (1 \leq i,j \leq n)$. By the type system, no t_0^i is free in $\mathcal{T}$ and $\mathcal{A}$. Using these property and Lemma 6.1, we can show the following by induction on i

$$\mathcal{K}\{t_0^i::S_{i-1}(k_0^i),\ldots,t_0^n::S_{i-1}(k_0^n)\},\mathcal{A}\{\langle I_0 : S_{i-1}(\alpha_0)\rangle\},\mathcal{T} \vdash S_{i-1}(C_2) : S_{i-1}(\tau_0)$$

In particular, we have

$$\mathcal{K},\mathcal{A}\{\langle I_0 : S_n(\alpha_0)\rangle\},\mathcal{T} \vdash S_n(C_2) : S_n(\tau_0)$$

By the type system, we must have $\mathcal{A} \vdash a_i : S_n(\alpha_0^i)$ for each $1 \leq i \leq m$. Then by Lemma 6.2, we have

$$\mathcal{K},\mathcal{A},\mathcal{T} \vdash [a_1/I_0^1,\ldots,a_m/I_0^m](S_n(C_2)) : S_n(\tau_0)$$

But since

$$[a_1/I_0^1,\ldots,a_m/I_0^m](S_n(C_2)) = [(t_0^1,\ldots,t_0^n)).(I_0^1,\ldots,I_0^m).C_2/x]C_1$$

this proves the case for variables.

Case for let expressions. Suppose

$$\begin{array}{l}\mathcal{K},\mathcal{A},\mathcal{T}\{x : \forall(\langle t_0::k_0\rangle).(\langle\alpha_0\rangle) \Rightarrow \tau_0\}\\ \quad\vdash\ let\ y : \forall(\langle t_1::k_1\rangle).(\langle\alpha_1\rangle) \Rightarrow \tau_1 = \Lambda(\langle t_1::k_1\rangle).(\langle I_1 : \alpha_1\rangle).C_1^1\ in\ C_1^2\ :\ \tau\end{array}$$

We must have

$$\mathcal{K}\{\langle t_1::k_1\rangle\},\mathcal{A}\{\langle I_1 : \alpha_1\rangle\},\mathcal{T}\{x : \forall(\langle t_0::k_0\rangle).(\langle\alpha_0\rangle) \Rightarrow \tau_0\} \vdash C_1^1 : \tau_1$$

and

$$\mathcal{K}, \mathcal{A}, \mathcal{T}\{x : \forall(\langle t_0 :: k_0\rangle).(\langle\alpha_0\rangle) \Rightarrow \tau_0\}\{y : \forall(\langle t_1 :: k_1\rangle).(\langle\alpha_1\rangle) \Rightarrow \tau_1\} \vdash C_1^2 \ : \ \tau$$

It is easily checked that the following weakening lemma for attribute assignment holds: if $\mathcal{K}, \mathcal{A}, \mathcal{T} \vdash C \ : \ \tau$, $dom(\mathcal{A}) \cap dom(\mathcal{A}') = \emptyset$, and $\mathcal{K} \vdash \mathcal{A}'$ then $\mathcal{K}, \mathcal{A}\mathcal{A}', \mathcal{T} \vdash C \ : \ \tau$. By the weakening lemma for kind environment (Lemma 3.4 for this language),

$$\mathcal{K}\{\langle t_1 :: k_1\rangle\}\{\langle t_0 :: k_0\rangle\}, \mathcal{A}\{\langle I_1 : \alpha_1\rangle\}\{\langle I_0 : \alpha_0\rangle\}, \mathcal{T} \vdash C_2 \ : \ \tau_0$$

By the induction hypothesis,

$$\mathcal{K}\{\langle t_1 :: k_1\rangle\}, \mathcal{A}\{\langle I_1 : \alpha_1\rangle\}, \mathcal{T} \vdash [(\langle t_0\rangle).(\langle I_0\rangle)C_2/x]C_1^1 \ : \ \tau_1$$

By the bound variable convention, we can assume that $x \neq y$. By the weakening lemma for type assignment (Lemma 3.2 for this language),

$$\mathcal{K}\{\langle t_0 :: k_0\rangle\}, \mathcal{A}\{\langle I_0 : \alpha_0\rangle\}, \mathcal{T}\{y : \forall(\langle t_1 :: k_1\rangle).(\langle\alpha_1\rangle) \Rightarrow \tau_1\} \vdash C_2 \ : \ \tau_0$$

Then by the induction hypothesis, we have

$$\mathcal{K}, \mathcal{A}, \mathcal{T}\{y : \forall(\langle t_1 :: k_1\rangle).(\langle\alpha_1\rangle) \Rightarrow \tau_1\} \vdash [(\langle t_0\rangle).(\langle I_0\rangle).C_2/x]C_1^2 \ : \ \tau$$

Then by the rule (LET) we have

$$\mathcal{K}, \mathcal{A}, \mathcal{T} \vdash let \ y : \forall(\langle t_1 :: k_1\rangle).(\langle\alpha_1\rangle) \Rightarrow \tau_1 = [(\langle t_0\rangle).(\langle I_0\rangle).C_2/x]C_1^1 \ in \ [(\langle t_0\rangle).(\langle I_0\rangle)C_2/x]C_1^2 \ : \ \tau$$

as desired. ∎

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

6.3 Type-Directed Specialization Algorithm

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

$$(\pi) \quad \frac{\mathcal{K}, \mathcal{T} \vdash M \ : \ T_\pi^1[\tau_1, \ldots, \tau_n] \qquad \mathcal{K} \vdash T_\pi^2[\tau_1, \ldots, \tau_n] :: K_\pi[\tau_1, \ldots, \tau_n]}{\mathcal{K}, \mathcal{T} \vdash \pi(M, T_\pi^2[\tau_1, \ldots, \tau_n], K_\pi[\tau_1, \ldots, \tau_n]) \ : \ T_\pi^3[\tau_1, \ldots, \tau_n]}$$

to the Λ^{impl} typing

$$(C_\pi) \quad \frac{\begin{array}{l}\mathcal{K}, \mathcal{A}, \mathcal{T} \vdash C \ : \ T_\pi^1[\tau_1, \ldots, \tau_n] \\ \mathcal{K} \vdash T_\pi^2[\tau_1, \ldots, \tau_n] :: K_\pi[\tau_1, \ldots, \tau_n] \\ \mathcal{A} \vdash a \ : \ P_{K_\pi[\tau_1, \ldots, \tau_n]}(T_\pi^2[\tau_1, \ldots, \tau_n])\end{array}}{\mathcal{K}, \mathcal{A}, \mathcal{T} \vdash C_\pi(C, a) \ : \ T_\pi^3[\tau_1, \ldots, \tau_n]}$$

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

We first define the type translation. 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. For example, if $\mathcal{K} = \{\ldots, t :: \{\!\{l_1 : t_1, l_2 : t_2\}\!\}, \ldots\}$ then the atomic kinding $t :: \{\!\{l_1 : t_1\}\!\}$ is in $\mathcal{K}$. Let $\{t_1' :: k_1', \ldots, t_m' :: k_m'\}$ be the set of

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

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

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

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

$$\mathcal{C}(\mathcal{A}_\mathcal{K}, (\mathcal{T})^*, f(M_1, \ldots, M_n)) = f(\mathcal{C}(\mathcal{A}_\mathcal{K}, (\mathcal{T})^*, M_1), \ldots, \mathcal{C}(\mathcal{A}_\mathcal{K}, (\mathcal{T})^*, M_n))$$

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

$$\begin{aligned}
&\mathcal{C}(\mathcal{A}_\mathcal{K}, (\mathcal{T})^*, let\ x : \forall(\langle t{::}k\rangle).\tau_1 = \Lambda(\langle t{::}k\rangle).M_1\ in\ M_2) \\
&\quad = \text{let } \forall(\langle t{::}k\rangle).(\alpha_1, \ldots, \alpha_m) \Rightarrow \tau_1 = (\forall(\langle t{::}k\rangle).\tau_1)^* \\
&\qquad\quad I_1, \ldots, I_m \text{ fresh} \\
&\qquad\quad C_1 = \mathcal{C}(\mathcal{A}_\mathcal{K}\{I_1 : \alpha_1, \ldots I_m : \alpha_m\}, (\mathcal{T})^*, M_1) \\
&\qquad\quad C_2 = \mathcal{C}(\mathcal{A}_\mathcal{K}, (\mathcal{T})^*(\{x : \forall(\langle t{::}k\rangle).\tau_1\})^*, M_2) \\
&\quad \text{in } let\ x : \forall(\langle t{::}k\rangle).(\alpha_1, \ldots, \alpha_m) \Rightarrow \tau_1 = \Lambda(\langle t{::}k\rangle).(I_1 : \alpha_1, \ldots, I_m : \alpha_m).C_1\ in\ C_2
\end{aligned}$$

Figure 8: The Compilation Algoritm from Λ^{ml} to Λ^{impl}

all atomic kindings in $\{t_1{::}k_1, \ldots, t_n{::}k_n\}$. The type $(\sigma)^*$ of Λ^{impl} corresponding to σ is defined as below.

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

We extend this relation to type assignments. 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 $\mathcal{A}_\mathcal{K}$ induced by $\mathcal{K}$ as

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

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

Using these notations, the compilation algorithm is given in Figure 8 as an algorithm $\mathcal{C}$ that takes a triple $(\mathcal{A}_\mathcal{K}, (\mathcal{T})^*, M)$ and computes a term of the implementation language. Since $\mathcal{A}_\mathcal{K}$ has the property that it has an element $I : P_k(t)$ 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.

This algorithm preserves typings in the following sense.

Theorem 6.4 *If $\mathcal{K}, \mathcal{T} \vdash M : \tau$ is a typing in Λ^{ml} and $\mathcal{C}(\mathcal{A}_\mathcal{K}, (\mathcal{T})^*, M) = C$ then $\mathcal{K}, \mathcal{A}_\mathcal{K}, (\mathcal{T})^* \vdash C : \tau$ is a typing in Λ^{impl}.*

Proof By induction on the structure of M using the following property: if $\mathcal{K} \vdash \mathcal{T}$ then $\mathcal{K} \vdash (\mathcal{T})^*$ and $\mathcal{K} \vdash \mathcal{A}_\mathcal{K}$. The case for atomic constants is trivial. Cases for abstraction and application follows from the induction hypothesis.

Case $(x\ \tau_1 \cdots \tau_n)$. Suppose $\mathcal{K}, \mathcal{T}\{x : \forall(t_1::k_1, \ldots, t_n::k_n).\tau_0\} \vdash (x\ \tau_1 \cdots \tau_n) : \tau$. Then $\mathcal{K} \vdash \tau_i :: [\tau_1/t_1, \ldots, \tau_{i-1}/t_{i-1}]k_i$ $(1 \leq i \leq n)$ and $\tau = [\tau_1/t_1, \ldots, \tau_n/t_n]\tau_0$. Let a_i, α_i be the type attributes and the attribute types mentioned in the algorithm, respectively. Since $\mathcal{K} \vdash \tau_i$, $\mathcal{K} \vdash \alpha'_i$ for each $1 \leq i \leq n$. If α_i is of the form $P_k(t)$ for some type variable, then $t \in dom(\mathcal{K})$. By the definition of $\mathcal{A}_\mathcal{K}$, there must be some I such that $(I : P_k(t)) \in \mathcal{A}_\mathcal{K}$, and $a_i = I$. If α_i is of the form $P_k(\tau)$ for some type that is not a type variable, then by our assumption on type attributes, there is a unique c such that $|P_k(\tau)| = c$, and $a_i = c$. In either cases, we have $\mathcal{A}_\mathcal{K} \vdash a_i : \alpha_i$. Then by the rule (VAR), we have

$$\mathcal{K}, \mathcal{A}_\mathcal{K}, (\mathcal{T}\{x : \forall(t_1::k_1, \ldots, t_n::k_n).\tau_0\})^* \vdash (x\ \tau_1 \cdots \tau_n\ a_1 \cdots a_m) : \tau$$

Case $\pi(M, \tau, k_\pi)$. Suppose $\mathcal{K}, \mathcal{T} \vdash \pi(M, \tau, k_\pi) : \tau'$. Then there must be some τ_0 such that $\mathcal{K}, \mathcal{T} \vdash M : \tau_0$ and $\mathcal{K} \vdash \tau :: k_\pi$. Let C_1 be the term mentioned in the algorithm. By the induction hypothesis, $\mathcal{K}, \mathcal{A}_\mathcal{K}, (\mathcal{T})^* \vdash C_1 : \tau_0$. Let a be the attribute mentioned in the algorithm. By the relationship between the rule (π) in Λ^{ml} and the rule (C_π) in Λ^{impl}, it is sufficient to show that $\mathcal{A}_\mathcal{K} \vdash a : P_{k_\pi}(\tau)$. By definition of $\mathcal{A}_\mathcal{K}$, if τ is a type variable t, then $\mathcal{A}_\mathcal{K}$ contains an entry $I : P_{k_\pi}(\tau)$. If τ is not a type variable, then there is some constant c such that $|P_{k_\pi}(\tau)| = c$. In either cases, $\mathcal{A}_\mathcal{K} \vdash a : P_{k_\pi}(\tau)$.

Case $let\ x : \forall(\langle t::k \rangle).\tau_1 = \Lambda(\langle t::k \rangle).M_1\ in\ M_2$. Suppose

$$\mathcal{K}, \mathcal{A}, \mathcal{T} \vdash let\ x : \forall(\langle t::k \rangle).\tau_1 = \Lambda(\langle t::k \rangle).M_1\ in\ M_2 : \tau$$

Then we must have $\mathcal{K}\{\langle t::k \rangle\}, \mathcal{A}, \mathcal{T} \vdash M_1 : \tau_1$; $\mathcal{K}, \mathcal{A}, \mathcal{T}\{x : \forall(\langle t::k \rangle).\tau_1\} \vdash M_2 : \tau$; and $\langle t \rangle \cap FTV(\mathcal{T}) = \emptyset$. By the definition of $(\forall(\langle t::k \rangle).\tau_1)^*$, the set of attribute types $\alpha_1, \ldots, \alpha_m$ mentioned in the algorithm satisfies the following

$$\{\alpha_1, \ldots, \alpha_m\} = \{P_{k'}(t') | t'::k' \text{ is an atomic kinding in } \langle t::k \rangle\}$$

Therefore by the definition of $\mathcal{A}_\mathcal{K}$, $\mathcal{A}_{\mathcal{K}\{\langle t::k \rangle\}} = \mathcal{A}_\mathcal{K}\{I_1 : \alpha_1, \ldots, I_m : \alpha_m\}$. Then by the induction hypothesis, we have

$$\mathcal{K}\{\langle t::k \rangle\}, \mathcal{A}_\mathcal{K}\{I_1 : \alpha_1, \ldots, I_m : \alpha_m\}, (\mathcal{T})^* \vdash C_1 : \tau_1$$

Since $(\mathcal{T}\{x : \forall(\langle t::k \rangle).\tau_1\})^* = (\mathcal{T})^*\{x : \forall(\langle t::k \rangle).(\alpha_1, \ldots, \alpha_m) \Rightarrow \tau_1\}$, by the induction hypothesis we have

$$\mathcal{K}, \mathcal{A}_\mathcal{K}, (\mathcal{T})^*\{x : \forall(\langle t::k \rangle).(\alpha_1, \ldots, \alpha_m) \Rightarrow \tau_1\} \vdash C_2 : \tau$$

By the rule (LET) we have

$$\mathcal{K}, \mathcal{A}_\mathcal{K}, (\mathcal{T})^* \vdash let\ x : \forall(\langle t::k \rangle).(\alpha_1, \ldots, \alpha_m) \Rightarrow \tau_1 = \Lambda(\langle t::k \rangle).(I_1 : \alpha_1, \ldots, I_m : \alpha_m).C_1\ in\ C_2 : \tau$$

as desired. ∎

6.4 Operational Semantics and Implementation Strategies

Λ^{impl} is a model for a language that can be implemented efficiently without computing attributes of types at run-time. The evaluation model of Λ^{impl} is therefore the language obtained by erasing

$$\text{(CONST)}\quad \mathcal{K},\mathcal{A},\mathcal{T} \vdash c^b \,:\, b \quad \text{if } \mathcal{K} \vdash \mathcal{T} \text{ and } \mathcal{K} \vdash \mathcal{A}$$

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

$$\text{(APP)}\quad \frac{\mathcal{K},\mathcal{A},\mathcal{T} \vdash e_1 \,:\, \tau_1 \to \tau_2 \quad \mathcal{K},\mathcal{A},\mathcal{T} \vdash e_2 \,:\, \tau_1}{\mathcal{K},\mathcal{A},\mathcal{T} \vdash e_1\ e_2 \,:\, \tau_2}$$

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

$$\text{(DATA)}\quad \frac{\mathcal{K},\mathcal{A},\mathcal{T} \vdash e_i \,:\, \tau_i\ (1 \le i \le n)}{\mathcal{K},\mathcal{A},\mathcal{T} \vdash f(e_1,\ldots,e_n) \,:\, F(\tau_1,\ldots,\tau_n)}$$

$$(c_\pi)\quad \frac{\begin{array}{l}\mathcal{K},\mathcal{A},\mathcal{T} \vdash e \,:\, T^1_\pi[\tau_1,\ldots,\tau_n] \\ \mathcal{K} \vdash T^2_\pi[\tau_1,\ldots,\tau_n] :: K_\pi[\tau_1,\ldots,\tau_n] \\ \mathcal{A} \vdash a \,:\, P_{K_\pi[\tau_1,\ldots,\tau_n]}(T^2_\pi[\tau_1,\ldots,\tau_n])\end{array}}{\mathcal{K},\mathcal{A},\mathcal{T} \vdash \pi(e,a) \,:\, T^3_\pi[\tau_1,\ldots,\tau_n]}$$

$$\text{(LET)}\quad \frac{\mathcal{K}\{\langle t{::}k\rangle\},\mathcal{A}\{\langle I : \alpha\rangle\},\mathcal{T} \vdash e_1 \,:\, \tau_1 \quad \mathcal{K},\mathcal{A},\mathcal{T}\{x : \forall(\langle t{::}k\rangle).(\langle\alpha\rangle) \Rightarrow \tau_1\} \vdash e_2 \,:\, \tau_2}{\mathcal{K},\mathcal{A},\mathcal{T} \vdash let\ x = \Lambda(\langle I\rangle).e_1\ in\ e_2 \,:\, \tau_2}$$

Figure 9: The Type System of λ^{impl}

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

$$e ::= c^b \mid (x\ a_1 \cdots a_m) \mid \lambda x.e \mid e\ e \mid f(e,\ldots,e) \mid \pi(e,a) \mid let\ x = \Lambda(I_1,\ldots,I_m).e\ in\ e$$

where we erase type information but not specialization information. The type system for λ^{impl} is given in Figure 9, which is essentially the same as that of Λ^{impl}.

We need to give an operational semantics that serves as an abstract description of an actual implementation. The challenge here is to develop a proper mechanism for evaluating specialization abstraction of the form $\Lambda(I_1,\ldots,I_m).e$. Since specialization abstraction is inserted by the type system to achieve efficient evaluation of a polymorphic function, this construct should be transparent and should not block the evaluation of e. Wright [40] proposed "value-only polymorphism," where polymorphism is restricted to syntactic values. 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 a part of the language definition. In this subsection, we develop an operational semantics for specialization abstraction that achieves the desired behavior mentioned above without placing the value-only restriction on polymorphism.

The operational semantics that is the closest to programming language implementation is perhaps natural semantics [18], where evaluation is defined by a set of evaluation relation of the form

$$E \vdash e \Downarrow v$$

denoting the fact that term e evaluates to value v under run-time environment E, which is a mapping from variables to values. In this setting, a value is one of a constant, a data structure or a closure. Since a closure delays the computation inside of the abstraction, in addition to closures, we need a new run-time data structure denoted by a term of the form $\Lambda(\langle I \rangle).e_1$. A naive approach is to introduce values of the form $\Lambda(\langle I \rangle).v$, whose operational meaning is given by substituting attribute values for attribute variables I, i.e. if x denotes $\Lambda(\langle I \rangle).v$, then attribute application $(x\ \langle a \rangle)$ evaluates to a value obtained by substituting $\langle a \rangle$ for attribute variables $\langle I \rangle$ in v. This requires a substitution to be applied to expressions in a closure. In an actual implementation this corresponds to changing code at run-time, and is unrealistic in most of run-time systems. To develop a practical evaluation model, we need to introduce a run-time data structure representing attribute variable binding.

We assume that there is a countable set of *dynamic attribute variables* ranged over by p. A *dynamic attribute value*, ranged over by γ, is either a dynamic attribute variable p or an attribute constant c. Let A be a *run-time attribute environment* which is a mapping from runtime attribute variables to runtime attribute values, and let E be a runtime value environment which is a mapping from variables to values. The set of values of λ^{impl} is given by the following grammar.

$$v ::= c^b \mid cls(A, E, \lambda x.e) \mid \Lambda(\langle p \rangle).v \mid f(v, \ldots, v) \mid [\pi(v, \gamma)] \mid \text{WRONG}$$

$cls(A, E, \lambda x.e)$ is a function closure. $\Lambda(\langle p \rangle).v$ is attribute abstraction. $\langle p \rangle$ in $\Lambda(\langle p \rangle).v$ are bound dynamic attribute variables, on which we assume the usual bound variable convention. If $\langle p \rangle$ is empty then $\Lambda(\langle p \rangle).v$ is identified with v. $f(v, \ldots, v)$ is a value for data structure. $[\pi(v, \gamma)]$ denotes the result of the polymorphic operation π on (v, γ), which may be WRONG *except when* it is well typed by the value typing defined below; in such case it produces a value of the same type. WRONG denotes run-time type error.

An *attribute substitution* is a mapping from a finite subset of dynamic attribute variables to dynamic attribute values. We write $[\gamma_1/p_1, \ldots, \gamma_n/p_n]$ for the attribute substitution that maps p_i to γ_i. The effect of applying an attribute substitution S to values is defined inductively as follows.

$$\begin{aligned}
S(p) &= \begin{cases} p & \text{if } p \notin dom(S) \\ S(p) & \text{if } p \in dom(S) \end{cases} \\
S(c^b) &= c^b \\
S(cls(A, E, \lambda x.e)) &= cls(\{I : S(\gamma) | (I : \gamma) \in A\}, \{x : S(v) | (x : v) \in E\}, \lambda x.e) \\
S(\Lambda(\langle p \rangle).v) &= \Lambda(\langle p \rangle).S(v) \\
S(f(v_1, \ldots, v_n)) &= f(S(v_1), \ldots, S(v_n)) \\
S([\pi(v, \gamma)]) &= [\pi(S(v), S(\gamma))]
\end{aligned}$$

The case of $[\pi(v, \gamma)]$ deserves some explanation. If the language does not contain polymorphic data constructor that requires specialization, then it is always the case that γ in $[\pi(v, \gamma)]$ is a constant attribute and $[\pi(v, \gamma)]$ is a value of one of the other forms possibly involving some part of v, and therefore this case is vacuous. However, if the language contains a polymorphic constructor such as polymorphic variants, then the runtime system needs to build a data structure containing dynamic attribute variables, which should be substituted at the time of specialization. The above case models this situation.

Using this mechanism, we define the operational semantics by the following two form of evaluation relations:

$A, E \vdash e \Downarrow v$ term e is evaluated to v under A and E, and

$A \vdash a \Downarrow \gamma$ static attribute a is evaluated to runtime attribute γ under A.

$$A \vdash c \Downarrow c$$

$$A \vdash I \Downarrow A(p) \quad \text{if } p \in dom(A)$$

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

$$\frac{x \in dom(E) \quad E(x) = \Lambda(p_1, \ldots, p_m).v \quad A \vdash a_i \Downarrow \gamma_i}{E \vdash (x\ a_1 \cdots a_m) \Downarrow [\gamma_1/p_1, \ldots, \gamma_m/p_m]v}$$

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

$$\frac{A, E \vdash e_1 \Downarrow cls(A_0, E_0, \lambda x.e_0) \quad A, E \vdash e_2 \Downarrow v_0 \quad A, E_0\{x : v_0\} \vdash e_0 \Downarrow v}{A, E \vdash e_1\ e_2 \Downarrow v}$$

$$\frac{A, E \vdash e_i \Downarrow v_i\ (1 \leq i \leq n)}{A, E \vdash f(e_1, \ldots, e_n) \Downarrow f(v_1, \ldots, v_n)}$$

$$\frac{A, E \vdash e \Downarrow v_1 \quad A \vdash a \Downarrow \gamma}{A, E \vdash \pi(e, a) \Downarrow [\pi(v_1, \gamma)]}$$

$$\frac{A\{I_1 : p_1, \ldots, I_m : p_m\}, E \vdash e_1 \Downarrow v_1 \quad A, E\{x : \Lambda(p_1, \ldots, p_m).v_1\} \vdash e_2 \Downarrow v}{A, E \vdash let\ x = \Lambda(I_1, \ldots, I_m).e_1\ in\ e_2 \Downarrow v} \quad p_1, \ldots, p_m \text{ fresh}$$

Figure 10: Evaluation Rules of λ^{impl}

The sets of rules that determine these two relations are given in Figure 10. These sets of rules should be taken with the following implicit rules yielding WRONG: if the evaluation of any of its component specified in the rule yields WRONG or does not satisfy the condition of the rule then the entire term yields WRONG.

To establish the soundness of typing, we define typing of dynamic values. Since our operational semantics involves manipulation of dynamic attribute variables, which act as a form of pointers (without dynamic creation or mutation), we need to set up value typing analogous to those involves pointers and stores. Here we adopt Leroy's approach [19]. The technique of defining typing of a closure using static typing judgment given below is due to Tofte[38].

Let $\mathcal{P}$ be a *dynamic attribute type assignment*, which maps dynamic attribute variables to attribute types. Value typing is defined by the following form of judgments:

$$\begin{array}{rl} \mathcal{K}, \mathcal{P} \models v : \sigma & \text{value } v \text{ has type } \sigma \text{ under } \mathcal{K} \text{ and } \mathcal{P}, \text{ and} \\ \mathcal{P} \models \gamma : \alpha & \text{dynamic attribute } \gamma \text{ has attribute type } \alpha \text{ under } \mathcal{P}. \end{array}$$

The set of rules to determine typing of run-time values is given below.

- $\mathcal{P} \models p : \mathcal{P}(p)$
- $\mathcal{P} \models c : \alpha$ if $c = |\alpha|$
- $\mathcal{K}, \mathcal{P} \models c^b : b$
- $\mathcal{K}, \mathcal{P} \models cls(A, E, \lambda x.e) : \tau_1 \to \tau_2$
  if there are some $\mathcal{A}, \mathcal{T}$ such that $\mathcal{P} \models A : \mathcal{A}$, $\mathcal{K}, \mathcal{P} \models E : \mathcal{T}$, and $\mathcal{K}, \mathcal{A}, \mathcal{T} \vdash \lambda x.e : \tau_1 \to \tau_2$
- $\mathcal{K}, \mathcal{P} \models [\pi(v, \gamma)] : T^3_\pi[\tau_1, \ldots, \tau_n]$
  if $\mathcal{K}, \mathcal{P} \models v : T^1_\pi[\tau_1, \ldots, \tau_n]$ and $\mathcal{P} \models \gamma : P_{K_\pi[\tau_1, \ldots, \tau_n]}(T^2_\pi[\tau_1, \ldots, \tau_n])$.

- $\mathcal{K}, \mathcal{P} \models \Lambda(p_1, \ldots, p_n).v : (\alpha_1, \ldots, \alpha_n) \Rightarrow \tau$ if $\mathcal{K}, \mathcal{P}\{p_1 : \alpha_1, \ldots, p_n : \alpha_n\} \models v : \tau$.
- $\mathcal{K}, \mathcal{P} \models v : \forall(\langle t::k \rangle).\tau$ if $\mathcal{K}\{\langle t::k \rangle\}, \mathcal{P} \models v : \tau$

The typing relation for A and E are given below.

- $\mathcal{P} \models A : \mathcal{A}$ if $dom(A) = dom(\mathcal{A})$ and for each $I \in dom(A)$, $\mathcal{P} \models A(I) : \mathcal{A}(I)$.
- $\mathcal{K}, \mathcal{P} \models E : \mathcal{T}$ if $dom(E) = dom(\mathcal{T})$ and for each $x \in dom(A)$, $\mathcal{K}, \mathcal{P} \models E(x) : \mathcal{T}(x)$.

Note that these mutually recursive definitions are well founded ones inductively defined on the structure of v.

To show the soundness of type system for λ^{impl}, we need substitution lemmas for attribute variables. The following holds by definition of attribute typings.

Lemma 6.5 *If* $\mathcal{P}\{p_1 : \alpha_1, \ldots, p_n : \alpha_n\} \models \gamma_0 : \alpha_0$, *and* $\mathcal{P} \models \gamma_i : \alpha_i (1 \leq i \leq n)$, *then* $\mathcal{P} \models [\gamma_1/p_1, \ldots, \gamma_n/p_n]\gamma_0 : \alpha_0$.

Lemma 6.6 *If* $\mathcal{K}, \mathcal{P}\{\langle p_0 : \alpha_0 \rangle\} \models v : \sigma$, *and* $\mathcal{P} \models \gamma : \alpha$ *for each* $(\gamma : \alpha) \in \langle \gamma_0 : \alpha_0 \rangle$, *then* $\mathcal{K}, \mathcal{P} \models [\langle \gamma_0/p_0 \rangle]v : \sigma$.

Proof By the definition of value typing for polytypes, it is sufficient to show the lemma for all the monotypes. Suppose $\mathcal{K}, \mathcal{P}\{\langle p_0 : \alpha_0 \rangle\} \models v : \tau$. The proof is by induction on the structure of v. The case for constants is trivial. The case for data constructors follows from the induction hypotheses.

Case $cls(A, E, \lambda x.e)$. Suppose $\mathcal{K}, \mathcal{P}\{\langle p_0 : \alpha_0 \rangle\} \models cls(A, E, \lambda x.e) : \tau_1 \to \tau_2$. Then there are some $\mathcal{A}, \mathcal{T}$ such that $\mathcal{P}\{\langle p_0 : \alpha_0 \rangle\} \models A : \mathcal{A}$; $\mathcal{K}, \mathcal{P}\{\langle p_0 : \alpha_0 \rangle\} \models E : \mathcal{T}$; and $\mathcal{K}, \mathcal{A}, \mathcal{T} \vdash \lambda x.e : \tau_1 \to \tau_2$. By Lemma 6.5, $\mathcal{P} \models [\langle \gamma_0/p_0 \rangle]A : \mathcal{A}$. By the induction hypothesis, for each $x \in dom(E)$, $\mathcal{K}, \mathcal{P} \models [\langle \gamma_0/p_0 \rangle]E(x) : \mathcal{T}(x)$, and therefore $\mathcal{K}, \mathcal{P} \models [\langle \gamma_0/p_0 \rangle]E : \mathcal{T}$. Thus we have $\mathcal{K}, \mathcal{P} \models [\langle \gamma_0/p_0 \rangle](cls(A, E, \lambda x.e)) : \tau_1 \to \tau_2$.

Case $\Lambda(\langle p_1 \rangle).v$. Suppose $\mathcal{K}, \mathcal{P}\{\langle p_0 : \alpha_0 \rangle\} \models \Lambda(\langle p_1 \rangle).v : (\langle \alpha_1 \rangle) \Rightarrow \tau_1$. Then $\mathcal{K}, \mathcal{P}\{\langle p_0 : \alpha_0 \rangle\}\{\langle p_1 : \alpha_1 \rangle\} \models v : \tau_1$. By the bound attribute variable assumption, we can assume that $\langle p_0 \rangle \cap \langle p_1 \rangle = \emptyset$. By the induction hypothesis, $\mathcal{K}\{\langle t::k \rangle\}, \mathcal{P}\{\langle p_1 : \alpha_1 \rangle\} \models [\langle \gamma_0/p_0 \rangle]v : \tau_1$. Then by the typing rules, we have $\mathcal{K}, \mathcal{P} \models [\langle \gamma_0/p_0 \rangle]v : (\langle \alpha_1 \rangle) \Rightarrow \tau$.

Case $[\pi(v_1, \gamma_1)]$. Suppose $\mathcal{K}, \mathcal{P}\{\langle p_0 : \alpha_0 \rangle\} \models [\pi(v_1, \gamma_1)] : \tau$. Then we must have: $\tau = T^3_\pi[\tau_1, \ldots, \tau_n]$; $\mathcal{K}, \mathcal{P}\{\langle p_0 : \alpha_0 \rangle\} \models v_1 : T^1_\pi[\tau_1, \ldots, \tau_n]$; and $\mathcal{P} \models \gamma_1 : P_{K_\pi[\tau_1, \ldots, \tau_n]}(T^2_\pi[\tau_1, \ldots, \tau_n])$. By the induction hypothesis, $\mathcal{K}, \mathcal{P} \models [\langle \gamma_0/p_0 \rangle]v_1 : T^1_\pi[\tau_1, \ldots, \tau_n]$. By Lemma 6.5, $\mathcal{P} \models [\langle \gamma_0/p_0 \rangle]\gamma_1 : P_{K_\pi[\tau_1, \ldots, \tau_n]}(T^2_\pi[\tau_1, \ldots, \tau_n])$. Then by value typing we have $\mathcal{K}, \mathcal{P} \models [\pi([\langle \gamma_0/p_0 \rangle]v_1, [\langle \gamma_0/p_0 \rangle]\gamma_1)] : T^3_\pi[\tau_1, \ldots, \tau_n]$. ∎

The following lemma can be proved by induction on v using the type substitution lemma for λ^{impl} whose proof is essentially the same as that of Lemma 6.1.

Lemma 6.7 *If* $\mathcal{K}\{t::k\}\mathcal{K}', \mathcal{P} \models v : \sigma$ *and* $\mathcal{K} \vdash \tau :: k$ *then* $\mathcal{K}([\tau/t]\mathcal{K}'), [\tau/t]\mathcal{P} \models v : [\tau/t]\sigma$

Using these properties, we show the type soundness theorem.

Theorem 6.8 *If* $\mathcal{K}, \mathcal{A}, \mathcal{T} \vdash e : \tau$, $\mathcal{P} \models A : \mathcal{A}$, $\mathcal{K}, \mathcal{P} \models E : \mathcal{T}$, *and* $A, E \vdash e \Downarrow v$ *then* $\mathcal{K}, \mathcal{P} \models v : \tau$.

Proof By induction on the length of computation $A, E \vdash e \Downarrow v$. The proof proceeds by cases in term of the structure of e. The case of constants is trivial. Cases for abstraction and application can be shown similarly to the corresponding proof of [19].

Case $(x\ a_1 \cdots a_m)$. Suppose:

$$\mathcal{K}, \mathcal{A}, \mathcal{T}\{x : \forall(t_1{::}k_1, \ldots, t_n{::}k_n).(\alpha_1, \ldots, \alpha_m) \Rightarrow \tau_0\} \vdash (x\ a_1 \cdots a_m) : [\tau_1/t_1, \ldots, \tau_n/t_n]\tau_0.$$

By the assumption on E, we have $\mathcal{K}, \mathcal{P} \models E(x) : \forall(t_1{::}k_1, \ldots, t_n{::}k_n).(\alpha_1, \ldots, \alpha_m) \Rightarrow \tau_0$. By the typing rule for λ^{impl}, $\mathcal{K} \vdash \tau_i :: [\tau_1/t_1, \ldots, \tau_{i-1}/t_{i-1}]k_i$. Since $t_1, \ldots, t_n$ do not appear in $\mathcal{P}$, by Lemma 6.7 we have the following.

$$\mathcal{K}, \mathcal{P} \models E(x) : ([\tau_1/t_1, \ldots, \tau_n/t_n]\alpha_1, \ldots, [\tau_1/t_1, \ldots, \tau_n/t_n]\alpha_m) \Rightarrow [\tau_1/t_1, \ldots, \tau_n/t_n]\tau_0$$

By the typing rules for values, $E(x)$ must be of the form $\Lambda(p_1, \ldots, p_n).v_0$. By the typing rule for λ^{impl}, we must have $\mathcal{A} \vdash a_i : [\tau_1/t_1, \ldots, \tau_n/t_n](\alpha_i)$. By the assumption on A, we have $A \vdash a_i \Downarrow \gamma_i$ such that $\mathcal{K}, \mathcal{P} \models \gamma_i : [\tau_1/t_1, \ldots, \tau_n/t_n](\alpha_i)$. Then $A, E \vdash (x\ a_1 \cdots a_m) \Downarrow [\gamma_1/p_1, \ldots, \gamma_n/p_n]v_0$. But by the typing rule for values,

$$\mathcal{K}, \mathcal{P}\{p_1 : [\tau_1/t_1, \ldots, \tau_n/t_n](\alpha_1), \ldots, p_m : [\tau_1/t_1, \ldots, \tau_n/t_n](\alpha_m)\} \models v_0 : [\tau_1/t_1, \ldots, \tau_n/t_n]\tau_0$$

Then by Lemma 6.6, $\mathcal{K}, \mathcal{P} \models [\gamma_1/p_1, \ldots, \gamma_n/p_n]v_0 : [\tau_1/t_1, \ldots, \tau_n/t_n]\tau_0$, as desired.

Case $f(e_1, \ldots, e_n)$. Suppose $\mathcal{K}, \mathcal{A}, \mathcal{T} \vdash f(e_1, \ldots, e_n) : T(\tau_1, \ldots, \tau_n)$. By simple induction on i using the induction hypothesis, we have $\mathcal{K}, \mathcal{P} \models e_1 \Downarrow v_i$ and $\mathcal{K}, \mathcal{P} \models v_i : \tau_i$. Therefore $A, E \vdash f(e_1, \ldots, e_n) \Downarrow f(v_1, \ldots, v_n)$ and $\mathcal{K}, \mathcal{P} \models f(v_1, \ldots, v_n) : T(\tau_1, \ldots, \tau_n)$.

Case $\pi(e, a)$. Suppose $\mathcal{K}, \mathcal{A}, \mathcal{T} \vdash \pi(e, a) : \tau$. Then we must have $\tau = T^3_\pi[\tau_1, \ldots, \tau_n]$; $\mathcal{K}, \mathcal{A}, \mathcal{T} \vdash e : T^1_\pi[\tau_1, \ldots, \tau_n]$; $\mathcal{K} \vdash T^2_\pi[\tau_1, \ldots, \tau_n] :: K_\pi[\tau_1, \ldots, \tau_n]$; and $\mathcal{A} \vdash a : P_{K_\pi[\tau_1, \ldots, \tau_n]}(T^2_\pi[\tau_1, \ldots, \tau_n])$. Suppose $A, E \vdash \pi(e, a) \Downarrow v$. Then $A, E \vdash e \Downarrow v_1$ for some v_1. By the induction hypothesis, $\mathcal{K}, \mathcal{P} \models v_1 : T^1_\pi[\tau_1, \ldots, \tau_n]$. Since $\mathcal{P} \models A : \mathcal{A}$, we have $A \vdash a \Downarrow \gamma$ such that $\mathcal{P} \models \gamma : P_{K_\pi[\tau_1, \ldots, \tau_n]}(T^2_\pi[\tau_1, \ldots, \tau_n])$. Then $\mathcal{K}, \mathcal{P} \models [\pi(v_1, a)] : T^3_\pi[\tau_1, \ldots, \tau_n]$. By our assumption that well-typed run-time primitive produces a value of the same type, $[\pi(v_1, a)] \Downarrow v \neq \text{WRONG}$ such that $\mathcal{K}, \mathcal{P} \models v : T^3_\pi[\tau_1, \ldots, \tau_n]$.

Case $let\ x = \Lambda(\langle I \rangle).e_1\ in\ e_2$. Suppose $\mathcal{K}, \mathcal{A}, \mathcal{T} \vdash let\ x = \Lambda(\langle I \rangle).e_1\ in\ e_2 : \tau$. Then we must have $\mathcal{K}\{\langle t{::}k \rangle\}, \mathcal{A}\{\langle I : \alpha \rangle\}, \mathcal{T} \vdash e_1 : \tau_1$ and $\mathcal{K}, \mathcal{A}, \mathcal{T}\{x : \forall(\langle t{::}k \rangle).(\langle \alpha \rangle) \Rightarrow \tau_1\} \vdash e_2 : \tau$. Suppose $A, E \vdash let\ x = \Lambda(\langle I \rangle).e_1\ in\ e_2 \Downarrow v$. Then $A\{\langle I : p \rangle\}, E \vdash e_1 \Downarrow v_1$ for some v_1 where $\langle p \rangle$ are fresh. Since $\mathcal{K}\{\langle t{::}k \rangle\}, \mathcal{P}\{\langle p : \alpha \rangle\} \models A\{\langle I : p \rangle\} : \mathcal{A}\{\langle I : \alpha \rangle\}$ and $\mathcal{K}\{\langle t{::}k \rangle\}, \mathcal{P}\{\langle p : \alpha \rangle\} \models E : \mathcal{T}$, $\mathcal{K}\{\langle t{::}k \rangle\}, \mathcal{P}\{\langle p : \alpha \rangle\} \models v_1 : \tau_1$. By the definition of evaluation, $A, E\{x : \Lambda(\langle p \rangle).v_1\} \vdash e_2 \Downarrow v$. By the definition of value typing, $\mathcal{K}, \mathcal{P} \models \Lambda(\langle p \rangle).v_1 : \forall(\{\langle t{::}k \rangle\}).(\langle \alpha \rangle) \Rightarrow \tau_1$. Thus $\mathcal{K}, \mathcal{P} \models E\{x : \Lambda(\langle p \rangle).v_1\} : \mathcal{T}\{x : \forall(\langle t{::}k \rangle).(\langle \alpha \rangle) \Rightarrow \tau_1\}$. Therefore by the induction hypothesis, $\mathcal{K}, \mathcal{P} \models v : \tau$. ∎

This result together with the type preservation theorem of the specialization transformation (Theorem 6.4) and the type soundness theorem of the implementation language Λ^{impl} (Theorem 6.8) ensure that the type system for the source language is sound with respect to the operational semantics obtained by the operational semantics of the specialized language.

7 Conclusions and Further Investigations

We have presented a framework for type-directed 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 types of arguments. This has been achieved by decomposing a polymorphic operation into a low-level generic operation and an attribute value of 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 an operation according to the attribute value. One distinguishing feature of our approach is that it use *singleton types* and encodes attribute values as types in the type system. By exploiting this feature, we have achieved type-directed specialization of polymorphic functions.

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

Efficient implementation strategy. To implement the framework presented here we need to develop an efficient implementation strategy for the operational of λ^{impl}. One unusual feature of the operational semantics is that attribute application (rule for $(x\ \langle I\rangle)$) involves substitution of a run-time value. As seen from the definition of substitution, the effect of substitution of the form $[\gamma_1/p_1, \ldots, \gamma_m/p_m]v$ is to update value environments E, attribute environments A and data structures in v but not code part of closures (i.e. $\lambda x.e$ in some $cls(A, E, \lambda x.e)$) in v. One strategy for implementing attribute application is the following.

1. An attribute abstraction $\Lambda(\langle p\rangle).v$ is implemented by a value together with sets $\langle\overline{p}\rangle$ of pointers indicating the places in v containing p.

2. Attribute application is implemented by first copying v and then updating the places pointed to by $\langle\overline{p}\rangle$ with the corresponding values.

If we implement an attribute abstraction $\Lambda(\langle I\rangle).e$ as a closure, then for each time at attribute application, a runtime data structure needs to be created by evaluating e under a new attribute environment. The above implementation evaluates e only once and should therefore yield faster implementation for attribute application. The value v in $\Lambda(\langle p\rangle).v$ can be regarded as a "template" value for each instance. We therefore believe that even if we adopted value-only polymorphism, this operational semantics is worthwhile adopting. A possible further optimization is to eliminate run-time traversal and copy of a value whenever possible. One promising approach is Minamide [26] where unnecessary run-time copy is eliminated by creating a data with a hole and destructively filling the hole whenever possible. We believe that this method can be adopted for implementing $\Lambda(\langle p\rangle).v$ and its application to attribute values.

Preservation of the semantics. The type preservation theorem (Theorem 6.4) and the type soundness theorem (Theorem 6.8) guarantees that the type system of the source language is sound with respect to the semantics achieved by a compiler embodying the type-directed specialization of polymorphism. In addition to this, we would like to show that type-directed specialization of polymorphism preserves the semantics of the source program. One promising strategy is to use the technique of logical relation and to set up a type index family of relations between the domain D_1^{σ} of Λ^{ml} and the corresponding domain $D_2^{(\sigma)^*}$ of Λ^{impl} in such a way that the type translation of σ is $(\sigma)^*$ and that the relation is strong enough to implies that the related elements have the same behavior. (See [28] for a survey on logical relations and their applications.) The semantic preservation can then be shown that the meaning of the translated term is related to the meaning of the source term. In [31, 25], such proofs are carried out. We believe that for each concrete instance of our framework, those technique can be adopted. However, a significant more work would be needed to develop a general result for our framework in such a way that it can be instantiated with various polymorphic primitives and type constructors. In order to establish such a method, we need to abstract the

necessary properties that should hold for the relationship between polymorphic primitives in λ^{ml} and the corresponding ones in Λ^{impl} and to properly strengthen the logical predicate. Recent results on generalizing logical predicates such as [1, 3, 14] may shed some light on developing such a general technique.

Intensional polymorphism. One interesting issue would be to apply the framework presented here to *intensional polymorphism* of Harper and Morrisett [13]. 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 intensional polymorphism. Elsman [9] has recently developed a compilation method for tag-free polymorphic equality based on a mechanism similar to that of [31]. His system can be regarded as an instance of the framework presented here with this extension.

Other Type Passing Calculi. Recently, Minamide [25] have 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 [25] 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-directed 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 [39, 11, 33]. Compared with these methods, the feature that distinguishes our approach is a type-theoretical treatment of static computation of type 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.

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 the three reviewers for their careful and through reading and their detailed comments, which have been very useful in improving this article. The author also thanks Takayasu Ito for his helpful comments on the conference version of this article.

References

[1] M. Alimohamed. A characterization of lambda definability in categorical models of implicit polymorphism. *Theoretical Computer Science*, 146:5–23, 1995.

[2] A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In *Proceedings of Third International Symposium on Programming Languages and Logic Programming*, pages 1–13, 1991.

[3] L. Birkedal and R. Harper. Relational interpretations of recursive types in an operational setting. In *Proc. Theoretical Aspects of Computer Software*, 1997.

[4] V. Breazu-Tannen, T. Coquand, C.A. Gunter, and A. Scedrov. Inheritance as explicit coercion. *Information and Computation*, 93:172–221, 1991.

[5] P. Buneman and A. Ohori. Polymorphism and type inference in database programming. *ACM Transactions on Database Systems*, 21(1):30–74, 1996.

[6] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surverys*, 17(4):471–522, December 1985.

[7] K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type-erasure semantics. In *Proc. International Conference on Functional Programming*, 1998.

[8] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[9] M. Elsman. Polymorphic equality - no tags required. In *Proceedings of the 2nd International Workshop on Types in Compilation*, Kyoto, March 1998.

[10] J.-Y. Girard. 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 *Second Scandinavian Logic Symposium*. North-Holland, 1971.

[11] C. Hall, K. Hammond, S. Peyton Jones, and P. Wadler. Type classes in Haskell. Technical report, University of Glasgow, 1994.

[12] R. Harper and J. C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, 1993.

[13] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Proc. ACM Symposium on Principles of Programming Languages*, 1995.

[14] M. Hasegawa. Logical predicates for intuitionistic linear type theories. In *To appear in Proc. TLCA Conference*, 1999.

[15] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. Guzman, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Perterson. Report on programming language Haskel a non-strict, purely functional language version 1.2. *SIGPLAN Notices, Haskell special issue*, 27(5), 1992.

[16] M. Jones. A theory of qualified types. In *Proc. ESOP Symposium*, 1992.

[17] M. Jones. ML typing, explicit polymorphism and qualified types. In *Proc. Theoretical Aspects of Computer Software, SLNCS 789*, 1994.

[18] G. Kahn. Natural semantics. In *Proc. Symposium on Theoretical Aspects of Computer Science*, pages 22–39. Springer Verlag, 1987.

[19] X. Leroy. *Polymorphic typing of an algorithmic language*. PhD thesis, University of Paris VII, 1992.

[20] X. Leroy. Unboxed objects and polymorphic typing. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 177–188, 1992.

[21] X. Leroy. The ZINC experiment: an economical implementation of the ML language. Technical Report 117, INRIA, 1992.

[22] X. Leroy. *The Objective Caml User's Manual.* INRIA Rocquencourt, B.P. 105 78153 Le Chesnay France, 1997.

[23] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[24] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, 1990.

[25] Y. Minamide. Compilation based on a calculus for explicit type passing. In *Proceedings of Fuji International Workshop on Functional and Logic Programming*, pages 301–320, 1996.

[26] Y. Minamide. A functional representation of data structures with a hole. In *Proc. ACM Symposium on Principles of Programming Languages*, 1998.

[27] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 271–283. ACM Press, 1996.

[28] J.C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 8, pages 365–458. MIT Press/Elsevier, 1990.

[29] A. Ohori. A simple semantics for ML polymorphism. In *Proceedings of ACM/IFIP Conference on Functional Programming Languages and Computer Architecture*, pages 281–292, London, England, September 1989.

[30] A Ohori. A compilation method for ML-style polymorphic record calculi. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 154–165, 1992.

[31] A. Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, 1995. A preliminary summary appeared at ACM POPL, 1992 under the title "A compilation method for ML-style polymorphic record calculi".

[32] A. Ohori and T. Takamizawa. A polymorphic unboxed calculus as an abstract machine for polymorphic languages. *Journal of Lisp and Symbolic Computation*, 10(1):61–91, 1997.

[33] J. Peterson and M. Jones. Implementing type classes. In *Proc. ACM Conference on Programming Language Design and Implementation*, 1993.

[34] S.L. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 636–666. Springer Lecture Notes in Computer Science, Vol 523, 1991.

[35] D. Rémy. Efficient representation of extensible records. In *Proc. ACM SIGPLAN Workshop on ML and its applications*, pages 12–16, 1994.

[36] J.C. Reynolds. Towards a theory of type structure. In *Paris Colloq. on Programming*, pages 408–425. Springer-Verlag, 1974.

[37] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL : A type-directed optimizing compiler for ML. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implemantation*, pages 181–192, New York, 1996. ACM Press.

[38] M. Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, Department of Computer Science, University of Edinburgh, 1988.

[39] A Tolmach. Tag-free garbage collection using explicit type parameters. In *Proc. ACM Conference on Lisp and Functional Programming*, 1994.

[40] A.K. Wright. Polymorphism for imperative languages without imperative types. Technical report TR93-200, Rice University, 1993.