An Unboxed Operational Semantics for ML Polymorphism*

ATSUSHI OHORI**   ohori@kurims.kyoto-u.ac.jp

TOMONOBU TAKAMIZAWA†

Research Institute for Mathematical Sciences, Kyoto University, Sakyo-ku, Kyoto 606 Japan

Keywords: Operational Semantics, Polymorphism, Type Inference, Unboxed Objects, ML

Abstract. We present an *unboxed* operational semantics for an ML-style polymorphic language. Different from the conventional formalisms, the proposed semantics accounts for actual representations of run-time objects of various types, and supports a refined notion of polymorphism that allows polymorphic functions to be applied directly to values of various different representations. In particular, polymorphic functions can receive multi-word constants such as floating-point numbers without requiring them to be "boxed" (i.e. heap allocated.) This semantics will serve as an alternative basis for implementing polymorphic languages. The development of the semantics is based on the technique of the type-inference-based compilation for polymorphic record operations [20]. We first develop a lower-level calculus, called a *polymorphic unboxed calculus*, that accounts for direct manipulation of unboxed values in a polymorphic language. This unboxed calculus supports efficient value binding through integer representation of variables. Different from de Bruijn indexes, our integer representation of a variable corresponds to the actual offset to the value in a run-time environment consisting of objects of various sizes. Polymorphism is supported through an abstraction mechanism over argument sizes. We then develop an algorithm that translates ML into the polymorphic unboxed calculus by using type information obtained through type inference. At the time of polymorphic *let* binding, the necessary size abstractions are inserted so that a polymorphic function is translated into a function that is polymorphic not only in the type of the argument but also in its size. The ML type system is shown to be sound with respect to the operational semantics realized by the translation.

1. Introduction

A commonly accepted view of *parametric polymorphism* underlying polymorphic programming languages is that a polymorphic function has the same behavior for

all possible instance types, and can therefore be implemented by the same code. This form of polymorphism is based on the generic nature of the value binding mechanism modeled by lambda abstraction. In this model, a variable can be bound to values of various types and those values can be freely substituted for the variable. A computational model of this binding mechanism is de Bruijn indexes [4], which has been the basis for various abstract machines for implementing ML-style polymorphic languages. In the perspective of implementation, de Bruijn indexes correspond to indexes into an array representing a run-time environment. An important implicit assumption underlying this binding mechanism is that all run-time data elements have the *same size* and therefore a run-time environment can be represented as a directly indexable array. In an actual computer system, however, run-time objects have various different sizes depending on their types, and do not conform to this model. To overcome this mismatch, in most implementations of polymorphic languages [2, 1, 11], a run-time object is represented as a *boxed object*, i.e. as a pointer to a value allocated in a heap. By this correspondence, in this paper, we call an operational semantics based on this uniform binding mechanism a *boxed semantics*.

Heap allocation is of course necessary for certain objects such as function closures and dynamically allocated lists. However, indiscriminately boxing all run-time objects significantly impairs the efficiency of the compiled code. To compile a program into efficient code for ordinary architecture, it is essential to assign a natural and efficient representation to each type and to use the operations specialized to the representation. This is a standard practice that is routinely carried out by any serious optimizing compiler of a conventional statically typed monomorphic language. It would be unfortunate if we are forced to abandon this fairly obvious and apparently effective representation optimization when we switch to a more advanced language with a polymorphic type system.

This problem has recently attracted attention of several researchers, and several methods have been proposed for run-time efficiency of polymorphic languages. Morrison et. al. [17] have described several optimization techniques for run-time specialization of functions using type information at run-time. Peyton Jones and Launchbury [22] have proposed a calculus where boxed and unboxed objects are explicitly manipulated and polymorphism is restricted to boxed objects. They have proposed several optimization strategies by source-to-source program translation. Leroy [14] proposed a systematic method to transform ML into a calculus similar to that of Peyton Jones and Launchbury. His strategy is to keep objects unboxed as long as its type is statically determined. To combine this strategy with polymor-

phic functions, his algorithm inserts representation conversion functions before and after each invocation of a polymorphic function. Leroy's method is further refined in [10, 24].

These methods of "mixed representations" try to minimize boxed representation overhead by localizing them to polymorphic functions. This approach should be effective for those applications whose cost is largely determined by monomorphic manipulation of multi-word data such as arithmetic computation loops. One apparent limitation of this approach is that it does not eliminate the mismatch between the boxed representation required by polymorphic functions and various unboxed data; polymorphic functions still require their arguments to be boxed. Due to this limitation, this approach would not speedup those applications that use polymorphic functions in manipulating multi-word data, or in some cases it would even decrease the efficiency of programs due to boxing/unboxing coercion overhead generated by the compiler. As Leroy pointed out, another weakness of this approach is that it does not work well for recursively defined data types such as lists or trees, since these data structures require boxing/unboxing coercions to be lifted to the structures by recursively applying the necessary coercions.

A potentially promising alternative to the mixed representation optimization is to develop a compilation method that allows polymorphic functions to manipulate unboxed objects of various sizes. Such a method would produces efficient code not only for monomorphic parts of a program but also for those that make heavy use of polymorphic functions with multi-word data and recursively defined data structures. To develop such a compilation method, we must first develop an operational semantics for a higher-order polymorphic functional language that accounts for direct manipulation of run-time objects of various sizes. Existing formalisms such as natural semantics [12] and various abstract machines including [2, 15] are all based on a boxed semantics mentioned above. To our knowledge, there is no formalism that allows polymorphic functions to manipulate a run-time environment consisting of unboxed values. The goal of this paper is to develop one such formalism of an *unboxed* operational semantics. Just as boxed semantics has been used as a high level description of conventional boxed implementation of a polymorphic language, we believe that our unboxed semantics will serve as a high level description of unboxed implementation of a polymorphic language.

In this paper, we focus on multi-word constants such as floating point numbers and do not attempt to unbox structured data constructors such as records. The rationale of this restriction is to support recursively defined data. Structured data constructors are often used in recursively defined data (for example, through

`datatype` construct in Standard ML) for representing dynamic data structures such as lists or trees. These dynamic data structure require the data constructors to be boxed. Our approach allows unboxed multi-world constants to be combined with recursively defined data without any additional machinery. For example, in our formalism, a list of floating point numbers is represented as a list of unboxed values and can be processed by various polymorphic combinators such as map and fold. We shall comment on the issue of unboxed structured data in Subsection 5.3.

We believe that the formalism of unboxed operational semantics we shall present in this paper can be used to implement a practical compiler for ML-style polymorphic programming languages. In actual implementation, it is of course necessary to consider various low-level structures of the target computer hardware. For example, most of current computer architectures provide "registers" for fast manipulation of several run-time objects including floating-point data, and proper usage of them is crucial in producing efficient code. Development of various optimization techniques for those low-level features is beyond the scope of the present article. However, since techniques for better usage of those features such as floating point registers naturally require run-time objects to be unboxed, we believe that our unboxed semantics also contributes to the development of better optimization techniques in compiling polymorphic languages.

In the rest of this section, we shall explain the problems and outline the solutions presented in this article. Our strategy of developing an unboxed operational semantics of ML is to follow the compilation method for record polymorphism presented in [20, 19]. We define a lower-level calculus, called a *polymorphic unboxed calculus*, that can manipulate unboxed values directly, and develop a type-inference-based translation algorithm of ML into this calculus. The combination of the translation algorithm and the operational semantics of the unboxed calculus achieves the desired unboxed semantics of ML.

There are two major technical challenges in developing the polymorphic unboxed calculus. The first is the development of a mechanism for representing variables that achieves fast access in an environment consisting of unboxed objects. Since sizes of unboxed objects differ, we cannot use de Bruijn indexes, but instead, we must develop a representation of a variable that indicates the actual position of the value in a run-time environment. This is a subtle problem especially for a polymorphic language. To see some of the difficulties, suppose $\lambda x.M$ is a polymorphic function of type $\forall t.t \rightarrow \sigma$. Since the offsets of variables defined in M depend on the size of x which varies according to the type of the actual argument, it is not possible to determine their offsets statically. We solve the problem by introducing two forms of

integer representations of variables: *direct indexes* of the form $dacc(n)$ and *indirect indexes* of the form $iacc(n)$. Suppose a variable x is bound by the kth nested lambda abstraction (counting from outer to inner.) If all the sizes of the values of the surrounding lambda variables are statically determined, then x is translated to $dacc(n)$ where n is the total sizes of the values of the surrounding lambda variables. Otherwise, x is translated to $iacc(k)$, where k is used as the index to the *offset table* whose ith element is the actual offset of the ith lambda variable. In most cases including all monomorphic parts of the program, variables are translated into direct indexes. As a simple example, the function $\lambda f.\lambda x.f\ (f\ x)$ is represented as $\lambda.\lambda.dacc(0)\ (dacc(0)\ dacc(1))$ under this scheme, assuming that a function closure (denoted by f) is a pointer of size 1.

The second challenge is the development of a mechanism for evaluating size dependent polymorphic operations. Since unboxed objects have different sizes, lambda abstraction and other primitive operations should be specialized to the sizes of the arguments. For this purpose, we refine de Bruijn style (name free) lambda abstraction $\lambda.M$ to $\lambda^k.M$ for each size k of the argument. To support polymorphic functions, we adopt the mechanism developed in [20] for compiling a polymorphic record calculus, and transform a size dependent function into a higher-order function that takes a size as a parameter. The general idea of passing size as a parameter and specializing a polymorphic function was described by Morrison et. al. [17]. The technical contribution of the present paper is to provide a formal basis for the approaches based on this general idea. In addition to $\lambda^k.M$, we allow abstraction of the form $\lambda^\alpha.M$ where α is (an integer representation of) a variable. This function first fetches the size, say k, of the argument through the variable α and then behaves like $\lambda^k.M$. By abstracting the variable α, we can define a function that is polymorphic not only in the type of its argument but also in its size. As an example, the polymorphic identity function $\lambda x.x$ is represented in the unboxed calculus as $\lambda^1\lambda^{dacc(0)}.dacc(1)$, where the first λ^1 receives a one-word datum indicating the size of x and the second $\lambda^{dacc(0)}$ uses this datum through the variable $dacc(0)$ bound by the first λ^1 and receives the value of x. Primitive operations such as projections are also refined in this way.

We define a polymorphic unboxed calculus incorporating these two ideas, give its operational semantics and prove the soundness of the type system. The type soundness guarantees both type consistency and size consistency of primitive operations used in any type-checked programs. The operational semantics is given in the style of natural semantics [12]. Different from the usual natural semantics, our

natural semantics properly accounts for direct manipulation of unbox objects, and can serve as a basis for unboxed implementation.

We then define an algorithm to translate ML into this calculus. The algorithm first performs type inference for a given raw ML term and produces an explicitly typed term. This process is analogous to the translation of Core ML into an explicitly typed second-order calculus, Core XML [7]. However, since translation of ML typings to Core XML terms is not coherent as shown in [18], Core XML is not entirely appropriate as an intermediate language for compilation. Our translation properly deals with this problem. In addition, the inference process determines for each type variable whether its size information is needed or not. The algorithm then converts the explicitly typed term into a term of the unboxed calculus by inserting appropriate size abstraction when a variable is bound to a polymorphic function through *let*, and size application when a *let* bound variable is instantiated.

Let us illustrate the flavor of the translation by simple examples. First, consider the monomorphic function

$$\lambda f.\lambda x.f(x + 3.14) : (real \rightarrow real) \rightarrow real \rightarrow real$$

where $+$ is the floating point addition. Under the assumption that the size of a function closure is 1 and the size of a floating point number (of type $real$) is 2, this function is translated into the following term in the polymorphic unboxed calculus

$$\lambda^1.\lambda^2.dacc(0)(dacc(1) + 3.14) : (real \rightarrow real) \rightarrow real \rightarrow real$$

where $dacc(0)$ and $dacc(1)$ indicate that the values bound to f and x are found at the offsets 0 and 1 in the run-time environment, respectively. Next, the polymorphic function

$$\lambda f.\lambda x.f\ x\ :\ \forall t.(t \rightarrow t) \rightarrow t \rightarrow t$$

is translated into the following term.

$$\lambda^1 \lambda^1 \lambda^{dacc(0)}(dacc(1)\ dacc(2)) : \forall s.size(s) \rightarrow (s \rightarrow s) \rightarrow s \rightarrow s$$

Since the size of f is statically known, both f and x are translated into direct indexes. However, if we reverse the order of abstractions as in $\lambda x.\lambda f.f\ x$, then x is translated into a direct index but f needs to be translated into an indirect index.

The rest of the paper is organized as follows. Section 2 defines Core ML. Section 3 defines the polymorphic unboxed calculus, gives its unboxed operational semantics, and proves the type soundness of the calculus, which establishes that a type-checked

program does not cause type and size error. Section 4 gives a translation algorithm from Core ML into the unboxed calculus, and shows that it preserves typings. Section 5 describes several further refinements. Section 6 compares this work with related works.

2. The Core ML

The source language we consider is the polymorphic core of ML. Following the analysis of Damas and Milner [3], we present the language as a functional calculus with a predicative polymorphic type system. The set of types is divided into monotypes (ranged over by τ) and polytypes (ranged over by σ) given below:

$$\tau ::= b^k \mid t \mid \tau \to \tau \mid \tau * \tau$$
$$\sigma ::= \tau \mid \forall t.\, \sigma$$

where b^k stands for base types for atomic constants of size k, t for type variables and $\tau * \tau$ for product types.

In the standard analysis of programming languages based on lambda calculus, the unit of evaluation or a "program" is identified with a closed term. In the semantic framework we shall develop in this paper, however, the behavior of a term depends crucially on its typing as well as its untyped term structure. We therefore need to consider a program as a closed term having a *closed* typing, i.e. the one that does not contain free type variables. Only those terms have meaning independent of the context and can therefore be compiled separately. To emphasize this property and to make our presentation more readable, we introduce the syntactic classes of *declarations* in addition to the usual definition of terms. Declaration correspond to compilation units in an actual ML implementation such as [1]. This restriction, however, conflicts with the Standard ML module system [16], which allows signatures to contain free type variables. To combine the approach presented here with Standard ML modules, we need to refine signatures so that they also contain size information of type variables. A detailed analysis of this issue is outside the scope of this paper.

The set of terms (ranged over by e) and the set of declarations (ranged over by d) of Core ML are given by the syntax:

$$e ::= x \mid c^{b^k} \mid \lambda x.\, e \mid e\, e \mid Pair(e,\, e) \mid \Pi_1\, e \mid \Pi_2\, e \mid let\ x\ =\ e\ in\ e$$
$$d ::= \emptyset \mid d; val\ x = e$$

x is a variable of the calculus, c^{b^k} is a constant of base type b^k, Π_1 is the first projection, Π_2 is the second projection, and $let\ x = e_1\ in\ e_2$ is ML's polymorphic let expression where a polymorphic function e_1 is bound and used in e_2. $\emptyset$ is the empty declaration, and $d; val\ x = e$ is the extension of the declaration d with $x = e$. We identify the terms that differ only in the names of bound variables, and further adopt the usual "bound variable convention," i.e. we assume that the set of all bound variables are distinct and are different from any free variables, and that this property is preserved by substitution. We write $[e_1/x]e_2$ for the term obtained from e_2 by substituting e_1 for all free occurrences of x in e_2.

In order to define a type system of the calculus, we introduce several notions on types and type variables. The set of free type variables of σ, denoted by $FTV(\sigma)$, is defined as usual. A type σ is *closed* if $FTV(\sigma) = \emptyset$. We extend these definitions to other syntactic structures containing types, including type assignments defined below. We identifies types that differ only in the names of bound type variables, and assume the "bound type variable convention" similar to the bound variable convention stated above. A *substitution* is a function from a subset of type variables to monotypes. We write $[\tau_1/t_1, \ldots, \tau_n/t_n]$ for the substitution S such that the domain of S, written $dom(S)$, is $\{t_1, \ldots, t_n\}$ and $S(t_i) = \tau_i$ $(1 \le i \le n)$. 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 extends uniquely to monotypes. The result of applying a substitution S to a polytype $\forall t.\sigma$ is the type obtained by applying S to its all *free type variables*. Under the bounded type variable convention, we can simply take $S(\forall t.\sigma) = \forall t.S(\sigma)$. In what follows, we identify a substitution with its extension to types, but we maintain that the domain of a substitution S always means the domain of the original function S. If S_1 and S_2 are substitutions, we write $S_1 S_2$ for the substitution S such that $dom(S) = dom(S_1) \cup dom(S_2)$; $S(t) = S_1(S_2(t))$ if $t \in dom(S_2)$; and $S(t) = S_1(t)$ if $t \in dom(S_1) \setminus dom(S_2)$. We further assume that this operation associates to the right so that $S_1 S_2 S_3$ means $S_1(S_2 S_3)$.

We say that τ' is an *instance* of $\sigma = \forall t_1 \ldots t_n.\, \tau$, written $\sigma > \tau'$, if there exists a substitution S such that $dom(S) = \{t_1, \ldots, t_n\}$ and $S(\tau) = \tau'$. This relation can be extended to polytypes as follows. Let $\sigma_1 = \forall t_1 \ldots t_n.\, \tau_1$, and $\sigma_2 = \forall t'_1 \ldots t'_m.\, \tau_2$. $\sigma_1 > \sigma_2$ if $\sigma_1 > \tau_2$ and no t'_j is free in σ_1. One can prove that $\sigma_1 > \sigma_2$ if and only if for all monotypes τ, if $\sigma_2 > \tau$ then $\sigma_1 > \tau$.

A *type assignment* Γ is a *list* (vector) of pairs of a variable and a type. The rationale behind this restricted treatment, compared to the usual approach of defining a type assignment to be a finite function from a set of variables to types, is that a type assignment in our calculus not only specifies the type of variables but it

also specifies the order of function abstraction, and therefore the order of values in the run-time environment. We use this correspondence to translate variables to indexes. We write $\Gamma; \{x : \sigma\}$ for the assignment obtained from Γ by adding a pair $\{x : \sigma\}$ at the end. We assume that a type assignment does not bind the same variable twice. Under the bound variable convention, there is no loss of generality in making this assumption. We write $\Gamma(x)$ for the type σ such that $\{x : \sigma\} \in \Gamma$ if one exists. We also use the following notations for denoting lists: $\emptyset$ is the empty list, $[a_1; \cdots; a_n]$ is the list consisting of the elements $a_1, \ldots, a_n$, and if L is a list and a is an element then $L; a$ is the list obtained by attaching a at the end of L.

For a type assignment Γ and a type τ, the *closure of σ with respect to* Γ, written $Clos(\Gamma, \tau)$, is the polytype $\forall t_1 \ldots t_n. \tau$ such that $t_1, \ldots, t_n$ are the set of type variables $FTV(\tau) \setminus FTV(\Gamma)$ in an arbitrary, but fixed order. It is always the case that if $Clos(\Gamma, \tau) = \sigma$ then τ is an instance of σ.

The type system is given as a set of rules to derive the following two forms of *typing judgments*:

$$\Gamma \vdash e : \tau \quad \text{term } e \text{ has type } \tau \text{ under type assignment } \Gamma, \text{ and}$$

$$\vdash d : \Gamma \quad \text{declaration } d \text{ yields type assignment } \Gamma.$$

The Core ML type system is given in Figure 1. Note that the rule for lambda abstraction only allows the last variable in a type assignment to be abstracted. As a consequence, the order of lambda abstractions is completely determined by the order of the occurrences of variables in a type assignment.

For typings, the following properties hold, whose proof can be found for example in [25].

LEMMA 1 *Let S be any substitution. If $\sigma > \tau$ then $S(\sigma) > S(\tau)$. If $\Gamma \vdash e : \tau$ then $S(\Gamma) \vdash e : S(\tau)$.*

For declarations, the following property is easily shown by induction on the length of the declaration.

LEMMA 2 *If $\vdash d : \Gamma$ then Γ is closed.*

A type τ is *principal* for a term e in Γ (or we simply say $\Gamma \vdash e : \tau$ is a *principal typing*) if $\Gamma \vdash e : \tau$ and, whenever $\Gamma \vdash e : \tau'$ then $Clos(\Gamma, \tau) > \tau'$. The *principal type assignment Γ for a declaration d* is defined inductively as follows:

- $\emptyset$ is the principal type assignment for $\emptyset$,

Typing rules for declarations:

$$\vdash \emptyset : \emptyset \qquad\qquad \frac{\vdash d : \Gamma \qquad\qquad \Gamma \vdash e : \tau}{\vdash (d; val\ x = e) : (\Gamma; \{x : Clos(\Gamma, \tau)\})}$$

Typing rules for terms:

$$\Gamma \vdash c^{b^k} : b^k \qquad\qquad\qquad\qquad \Gamma \vdash x : \tau \quad \text{if } x : \sigma \in \Gamma \text{ s.t. } \sigma > \tau$$

$$\frac{\Gamma; \{x : \tau_1\} \vdash e : \tau_2}{\Gamma \vdash \lambda x.\, e : \tau_1 \to \tau_2} \qquad\qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash Pair(e_1, e_2) : \tau_1 * \tau_2} \qquad\qquad \frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash \Pi_i e : \tau_i}\ i = 1, 2$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma; \{x : Clos(\Gamma, \tau_1)\} \vdash e_2 : \tau_2}{\Gamma \vdash let\ x\ =\ e_1\ in\ e_2 : \tau_2}$$

Figure 1. The Type System of Core ML

- $\Gamma; \{x : \forall t_1 \ldots t_n.\tau\}$ is the principal type assignment for $d; val\ x = e$ if Γ is the principal type assignment for d, $\{t_1, \ldots, t_n\} = FTV(\tau)$ and τ is principal for e in Γ.

In the last definition, since Γ is closed, $Clos(\Gamma, \tau) = \forall t_1 \ldots t_n.\tau$. It should be noted that for open terms, our definition of principal typing is weaker than the corresponding one in [3], but it is equivalent for closed typings (i.e. when $\Gamma = \emptyset$) and declarations. Since we are usually interested in types of closed terms and declarations, this definition is sufficient.

3. The Polymorphic Unboxed Calculus

This section defines the polymorphic unboxed calculus. This calculus serves as an abstract machine for unboxed operational semantics, and is the target calculus of our translation.

We let k range over the set of natural numbers used as *size constants* in the calculus, and let α range over the set of size constants and the two kinds of variables given below:

$$\alpha\ ::=\ k \mid dacc(n) \mid iacc(n)$$

where n ranges over natural numbers; $dacc(n), iacc(n)$ are *direct indexes* and *indirect indexes* respectively, as explained in the introduction.

The set of *raw terms* of the calculus (ranged over by M) and the set of *declarations* (ranged over by D) are given by the following syntax:

$$M ::= c^{b^k} \mid \alpha \mid \lambda^\alpha. M \mid M\ M \mid Pair^{\alpha,\alpha}(M, M) \mid \Pi_\alpha M \mid let^\alpha\ M\ in\ M$$
$$D ::= \emptyset \mid D; M^k$$

where $\lambda^\alpha.M$ stands for lambda abstraction whose argument size is denoted by α; $Pair^{\alpha_1,\alpha_2}(M_1, M_2)$ for a pair of terms M_1 and M_2 whose sizes are denoted by α_1 and α_2, respectively; $\Pi_\alpha M$ accesses the position denoted by α in a multi-word value denoted by M, and corresponds to projection functions. $let^\alpha\ M_1\ in\ M_2$ binds a variable to M_1 of size denoted by α in M_2, and is operationally equivalent to $(\lambda^\alpha.M_2)\ M_1$.

3.1. The type system

To establish the type soundness of the operational semantics of ML we shall develop later, we define a polymorphic type system of the unboxed calculus. The set of monotypes and polytypes of this calculus are given by the following syntax:

$$\tau ::= b^k \mid t \mid size(\tau) \mid \tau \to \tau \mid \tau * \tau$$
$$\sigma ::= \tau \mid \forall t. \sigma$$

In the above definition, we have introduced a special type constructor $size(\tau)$. This type denotes the singleton set of integer representing the *size* of values of type τ. If τ is a type variable t, then $size(\tau)$ denotes (the singleton set of) an unknown integer determined by instantiation of t. For example, if the size of an integer is 1 then $size(int)$ is the type whose only element is 1. In other words, $size(\tau)$ is an integer value which is itself a type. This technique of treating values as types, originally developed in [20], enables us to perform necessary specialization in the framework of type inference system.

The *size* of a type σ, denoted by $|\sigma|$ is defined as follows.

- $|b^k| = k$

- $|\tau \to \tau| = 1$

- $|size(\tau)| = 1$

- $|\tau_1 * \tau_2| = 1$

- $|t|$ undefined.

- $|\forall t_1.\ldots.t_n.\tau| = |\tau|$

Note that the size of a type variable is undefined and is treated specially in the following development. The following is easily shown from the definition.

LEMMA 3 *For any substitution S, if $|\sigma|$ is defined then $|S(\sigma)|$ is defined and equal to $|\sigma|$.*

In the above definition we have assumed that functions and products are boxed values of size 1. As mentioned in Introduction, the rationale behind the choice of boxed products is that they are necessary for defining dynamic data structures such as lists or trees. This choice also significantly simplifies the translation we shall develop later due to the property that the size of a type is always determined by the topmost type constructor. Note, however, that the components of a pair are not required to be boxed. For example, $Pair^{real,real}(3.14, 2.18)$ is a boxed pair of unboxed real numbers, and in an actual implementation it will be represented as a pointer to a vector consisting of two unboxed floating point numbers. The following development correctly models this representation.

A *type assignment* Δ in this calculus is a list of types. We write $\Delta.j$ to denote the jth element in the list Δ (counting from left, starting with 1). We also define the following notation to specify an element in a list Δ:

$$\Delta[n] = \begin{cases} \Delta.1 & \text{if } n = 0 \\ \Delta.j & \text{if } |\Delta.l| \text{ is defined for all } 1 \leq l \leq j-1 \\ & \text{and } n = |\Delta.1| + \cdots + |\Delta.(j-1)| \\ \text{undefined} & \text{otherwise} \end{cases}$$

The intention of this definition is that if $\Delta[n] = \Delta.j$ then n is the offset of the jth element in the run-time environment.

As in Core ML, the type system is defined as a system to derive the following two forms of judgments:

$\Delta \vdash M : \tau$ term M has type τ under type assignment Δ, and

$\vdash D : \Delta$ declaration D yields type assignment Δ.

The type system of the polymorphic unboxed calculus is given in Figure 2.

An example of a typing derivation in this calculus is given in Figure 3, which shows a typing derivation for the term $\lambda^1.\lambda^2.dacc(0)(dacc(1) + 3.14)$ corresponding to the term $\lambda f.\lambda x.f(x + 3.14)$ given in the introduction.

Typing rules for declarations:

$$\vdash \emptyset : \emptyset \qquad \frac{\vdash D : \Delta \qquad \Delta \vdash M : \tau \qquad \Delta \vdash k : size(\tau)}{\vdash (D; M^k) : (\Delta; Clos(\Delta, \tau))}$$

Typing rules for terms:

$$\Delta \vdash c^{b^k} : b^k$$

$$\Delta \vdash k : size(\sigma) \quad \text{if } |\sigma| = k$$

$$\Delta \vdash dacc(n) : \tau \quad \text{if } \Delta[n] = \sigma \text{ and } \sigma > \tau$$

$$\Delta \vdash iacc(n) : \tau \quad \text{if } \Delta.n = \sigma \text{ and } \sigma > \tau$$

$$\frac{\Delta; \tau_1 \vdash M : \tau_2 \qquad \Delta \vdash \alpha : size(\tau_1)}{\Delta \vdash \lambda^\alpha.\, M : \tau_1 \to \tau_2}$$

$$\frac{\Delta \vdash M_1 : \tau_1 \to \tau_2 \qquad \Delta \vdash M_2 : \tau_1}{\Delta \vdash M_1\ M_2 : \tau_2}$$

$$\frac{\Delta \vdash M_1 : \tau_1 \quad \Delta \vdash M_2 : \tau_2 \quad \Delta \vdash \alpha_1 : size(\tau_1) \quad \Delta \vdash \alpha_2 : size(\tau_2)}{\Delta \vdash Pair^{\alpha_1, \alpha_2}(M_1, M_2) : \tau_1 * \tau_2}$$

$$\frac{\Delta \vdash M : \tau_1 * \tau_2}{\Delta \vdash \Pi_0\ M : \tau_1}$$

$$\frac{\Delta \vdash M : \tau_1 * \tau_2 \quad \Delta \vdash \alpha : size(\tau_1)}{\Delta \vdash \Pi_\alpha\ M : \tau_2}$$

$$\frac{\Delta \vdash M_1 : \tau_1 \quad \Delta; Clos(\Delta, \tau_1) \vdash M_2 : \tau_2 \quad \Delta \vdash \alpha : size(\tau_1)}{\Delta \vdash let^\alpha\ M_1\ in\ M_2 : \tau_2}$$

Figure 2. The Type System of the Polymorphic Unboxed Calculus

$$\frac{[real \rightarrow real; real] \vdash dacc(1) : real \quad [real \rightarrow real; real] \vdash 3.14 : real}{[real \rightarrow real; real] \vdash dacc(1) + 3.14 : real} \quad \equiv \quad \Omega_1$$

$$\frac{[real \rightarrow real; real] \vdash dacc(0) : real \rightarrow real \quad \Omega_1}{[real \rightarrow real; real] \vdash dacc(0)(dacc(1) + 3.14) : real} \quad \equiv \quad \Omega_2$$

$$\frac{\Omega_2 \quad [real \rightarrow real; real] \vdash 2 : size(real)}{[real \rightarrow real] \vdash \lambda^2.dacc(0)(dacc(1) + 3.14) : real \rightarrow real} \quad \equiv \quad \Omega_3$$

$$\frac{\Omega_3 \quad [real \rightarrow real] \vdash 1 : size(real \rightarrow real)}{\emptyset \vdash \lambda^1.\lambda^2.dacc(0)(dacc(1) + 3.14) : (real \rightarrow real) \rightarrow real \rightarrow real}$$

Figure 3. Examples of typing derivations in the unboxed calculus

3.2. Operational semantics

We define an unboxed operational semantics of this calculus in the style of natural semantics [12] by giving a set of rules to derive the following evaluation relation:

$$E, A, i \vdash M \Downarrow v$$

indicating that term M is evaluated to *canonical values* v in the run-time environment determined by E, A and i.

The set of *canonical values* (ranged over by v) is defined as follows:

$$v \ ::= \ c^{b^k} \mid k \mid cls(k, E, A, i, M) \mid pair(v_1, v_2) \mid wrong$$

where $cls(k, E, A, i, M)$ is a function closure and *wrong* represents run-time error. The *size* $|v|$ *of a canonical value* v is defined as follows:

- $|c^{b^k}| = k$

- $|k| = 1$

- $|cls(k, E, A, i, M)| = 1$

- $|pair(v_1, v_2)| = 1$

E, A, and i determine the state of a run-time evaluation, each of which is explained below.

- E is a *value environment*, which is a vector of canonical values. We write $E.j$ to denote the jth element in E (counting from left to right, starting with 1).

Note, however, that this is a meta notation for elements in E we use in the following discussions and proofs. To model the operation for run-time access of elements in E, we define the operation $E[n]$ similarly to $\Delta[n]$ as follows:

$$E[n] = \begin{cases} E.1 & \text{if } n = 0 \\ E.j & \text{if } n = |E.1| + \cdots + |E.(j-1)| \\ wrong & \text{otherwise} \end{cases}$$

The operation $E[n]$ reflects the property that E is a vector containing canonical values of various sizes, and each element in E is accessed by its offset.

- A is an *offset table*, which is a vector of natural numbers maintaining the offsets of each element of E. We define the operation $A.j$ such that if j is smaller or equal to the number of elements in A then it returns the jth element in A, otherwise $A.j$ returns *wrong*. Since all the elements of A have the same size, $A.j$ is directly implementable.

- i is a natural number indicating the offset of the next available space in the environment.

As we shall show later, the operational semantics is defined so that the following properties are invariant: A and E have the same number of elements, $A.j$ is the offset to the jth value in E, and i is the sum of the sizes of all the elements in E.

Figure 4 gives the operational semantics for the polymorphic unboxed calculus, which specifies the usual call-by-value, left-to-right evaluation. This set 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, or any of the operations used in the rule yields *wrong*, then the entire term yields *wrong*. For example, for the function application, the following rules are implicitly assumed:

$$\frac{E_1, A_1, i_1 \vdash M_1 \Downarrow v \text{ and } v \text{ is not of the form } cls(k, E_2, A_2, i_2, M_3)}{E_1, A_1, i_1 \vdash M_1 \ M_2 \Downarrow wrong}$$

$$\frac{E_1, A_1, i_1 \vdash M_1 \Downarrow cls(k, E_2, A_2, i_2, M_3) \quad E_1, A_1, i_1 \vdash M_2 \Downarrow wrong}{E_1, A_1, i_1 \vdash M_1 \ M_2 \Downarrow wrong}$$

$$\frac{E_1, A_1, i_1 \vdash M_1 \Downarrow cls(k, E_2, A_2, i_2, M_3) \quad E_1, A_1, i_1 \vdash M_2 \Downarrow v \quad |v| \neq k}{E_1, A_1, i_1 \vdash M_1 \ M_2 \Downarrow wrong}$$

The rules for other term constructors are similarly understood. Note however that

$$E, A, i \vdash c^{b^k} \Downarrow c^{b^k}$$

$$E, A, i \vdash k \Downarrow k$$

$$\frac{E, A, i \vdash \alpha \Downarrow k}{E, A, i \vdash \lambda^{\alpha}. \, M \Downarrow cls(k, E, A, i, M)}$$

$$E, A, i \vdash dacc(n) \Downarrow E[n]$$

$$E, A, i \vdash iacc(n) \Downarrow E[A.n]$$

$$\frac{\begin{array}{c} E_1, A_1, i_1 \vdash M_1 \Downarrow cls(k, E_2, A_2, i_2, M_3) \\ E_1, A_1, i_1 \vdash M_2 \Downarrow v_1 \\ E_2; v_1, A_2; i_2, i_2 + k \vdash M_3 \Downarrow v_2 \end{array} \quad \text{if } |v_1| = k}{E_1, A_1, i_1 \vdash M_1 \, M_2 \Downarrow v_2}$$

$$\frac{E, A, i \vdash M_1 \Downarrow v_1 \quad E, A, i \vdash M_2 \Downarrow v_2 \quad E, A, i \vdash \alpha_i \Downarrow k_i}{E, A, i \vdash Pair^{\alpha_1 \alpha_2}(M_1, M_2) \Downarrow pair(v_1, v_2)} \quad \text{if } |v_i| = k_i \ (i = 1, 2)$$

$$\frac{E, A, i \vdash M \Downarrow pair(v_1, v_2)}{E, A, i \vdash \Pi_0 \, M \Downarrow v_1}$$

$$\frac{E, A, i \vdash M \Downarrow pair(v_1, v_2) \quad E, A, i \vdash \alpha \Downarrow k}{E, A, i \vdash \Pi_{\alpha} \, M \Downarrow v_2} \quad \text{if } |v_1| = k$$

$$\frac{E, A, i \vdash M_1 \Downarrow v_1 \quad E, A, i \vdash \alpha \Downarrow k \quad E; v_1, A; i, i + k \vdash M_2 \Downarrow v_2}{E, A, i \vdash let^{\alpha} \, M_1 \, in \, M_2 \Downarrow v_2} \quad \text{if } |v_1| = k$$

(The rules yielding *wrong* are omitted.)

Figure 4. The Operational Semantics of the Polymorphic Unboxed Calculus

in an actual implementation, the evaluation simply proceeds by assuming that the argument value satisfies the conditions, and does not perform any run-time check of size correctness or type correctness. When the argument does not conform to the constraints required by the operation, something just goes wrong. The introduction of *wrong* value in our formalism is to model this situation. Later, we shall show that well typed programs never go wrong by showing the type soundness theorem with respect to this operational semantics.

$$\dfrac{\begin{array}{c} \emptyset, \emptyset, 0 \vdash \lambda^2.\lambda^2.dacc(0) + dacc(2) \Downarrow cls(2, \emptyset, \emptyset, 0, \lambda^2.dacc(0) + dacc(2)) \\ \emptyset, \emptyset, 0 \vdash 2.72 \Downarrow 2.72 \\ [2.72], [0], 2 \vdash \lambda^2.dacc(0) + dacc(2) \Downarrow cls(2,\ [2.72],\ [0],\ 2, dacc(0) + dacc(2)) \end{array}}{\emptyset, \emptyset, 0 \vdash (\lambda^2.\lambda^2.dacc(0) + dacc(2))\ 2.72 \Downarrow cls(2, [2.72], [0], 2, dacc(0) + dacc(2))}$$

$$\dfrac{\begin{array}{c} \emptyset, \emptyset, 0 \vdash (\lambda^2.\lambda^2.dacc(0) + dacc(2))\ 2.72 \Downarrow cls(2,\ [2.72],\ [0],\ 2,\ dacc(0) + dacc(2)) \\ \emptyset, \emptyset, 0 \vdash 3.14 \Downarrow 3.14 \\ [2.72; 3.14], [0; 2], 4 \vdash dacc(0) + dacc(2) \Downarrow 5.86 \end{array}}{\emptyset, \emptyset, 0 \vdash (\lambda^1.\lambda^2.dacc(0) + dacc(2))\ 2.72\ 3.14 \Downarrow 5.86}$$

Figure 5. Example of evaluation in the unboxed calculus

This set of rules is designed so that they can be implemented efficiently in a conventional computer system. In particular, size information is explicitly available whenever size dependent action will be needed. Some explanation is in order in the perspective of implementation. The rule for $dacc(n)$ performs direct memory access by address, and the rule for $iacc(n)$ performs indirect memory access. The rule for function application $M_1\ M_2$ first evaluates M_1 to obtain a closure, evaluates M_2 to obtain a value v, switches the evaluation context to the one stored in the closure and copies the first k words of v to the end of the environment, and finally evaluates the body of the closure. In an actual implementation, some part of the environment may be implemented by several registers, which will be saved in the environment when needed. Such optimization is easily incorporated. This scheme performs correct action if the size $|v|$ of the argument and the size information k stored in the closure are the same. As we shall see later, the type system guarantees that this is always the case for any type-checked programs. The rule for $Pair^{\alpha_1, \alpha_2}(M_1, M_2)$ evaluates M_1, M_2 in this order to obtain values v_1, v_2, evaluates α_1, α_2 to obtain size constants k_1, k_2, allocates a $k_1 + k_2$ word block in the heap, copies first k_1, k_2 words of v_1 and v_2 respectively to the heap, and returns a pointer to the heap. Again the type system guarantees that size constants k_1, k_2 reflects the actual sizes of v_1, v_2 respectively. Π_0 accesses the first element of the pair. Π_α first gets the offset value corresponding to the second element of the pair, and it then accesses the element. An example of evaluation in the unboxed calculus is given in Figure 5.

The operational semantics of declaration $M_1; \cdots; M_n$ is defined as follows:

$$(M_1^{k_1}; \cdots; M_n^{k_n}) \Downarrow v \iff \emptyset, \emptyset, 0 \vdash (let^{k_1}\ M_1\ in\ let^{k_2}\ M_2\ in\ \cdots\ in\ M_n) \Downarrow v$$

3.3. Soundness of the type system

As seen from the definition, the operational semantics is unsafe – evaluation may return *wrong* due to type inconsistency or size inconsistency. However, if a program is type correct then it is guaranteed that its evaluation is always type consistent and size consistent, and therefore "does not go wrong". We show this desirable property by showing the soundness of the type system.

We write $\models v : \sigma$ if canonical value v has type σ. For a closed monotype, this relation is defined as follows:

- $\models c^{b^k} : b^k$

- $\models k : size(\tau)$ if $|\tau| = k$

- $\models cls(k, E, A, i, M) : \tau_1 \to \tau_2$ if $|\tau_1| = k$
  and for all v_1 if $\models v_1 : \tau_1$ and $E; v_1, A; i, i + k \vdash M \Downarrow v_2$ then $\models v_2 : \tau_2$

- $\models pair(v_1, v_2) : \tau_1 * \tau_2$ if $\models v_1 : \tau_1$ and $\models v_2 : \tau_2$

This relation is extended to general polytypes by the following rules:

- $\models v : \tau$ if $\models v : S(\tau)$ for all total ground substitutions S

- $\models v : \forall t_1 \ldots t_n. \tau$ if $\models v : \tau$

We write $Elms(L)$ for the number of elements in L. For a value environment E, we write $Size(E)$ for the size of E, i.e. $Size(E) = \Sigma_{1 \le i \le Elms(E)} |E.i|$. We say that E, A, i *respect* a closed type assignment Δ, denoted by $E, A, i \models \Delta$, if the following conditions are satisfied:

- $i = Size(E)$,

- $Elms(\Delta) = Elms(E) = Elms(A)$,

- $\models E.j : \Delta.j$ for all $1 \le j \le Elms(E)$,

- $A.1 = 0$, $A.(j + 1) = A.j + |E.j|$ for all $1 \le j \le Elms(E) - 1$,

For general Δ, we define $E, A, i \models \Delta$ if $E, A, i \models S(\Delta)$ for any total ground substitution S.

The purpose of the rest of this section is to prove the type soundness theorem for the polymorphic unboxed calculus. We first show some lemmas.

LEMMA 4   *1. If $\models v : \sigma$ and $\sigma > \tau$ then $\models v : \tau$.*

2. *If $\models v : \sigma$ then $|v| = |\sigma|$.*

3. *If $E, A, i \models \Delta$ then $A.1 = 0$, $A.j = |E.1| + |E.2| + \cdots + |E.(j - 1)|$ for all $2 \le j \le Elms(A)$, and $|E.j| = |\Delta.j|$ for all $j \le Elms(A)$.*

Proof: By definitions. ∎

LEMMA 5 *Let S be any substitution. If $\Delta \vdash M : \tau$ then $S(\Delta) \vdash M : S(\tau)$.*

Proof: By induction on the structure of M. We show the cases for size constants and variables. The case for let is shown similarly to the corresponding lemma for Core ML. Other cases follow directly from the induction hypothesis.

Case k. Suppose $\Delta \vdash k : size(\sigma)$. Then $k = |\sigma|$. By lemma 3, $|\sigma| = |S(\sigma)|$. Thus $S(\Delta) \vdash k : size(S(\sigma))$.

Case $dacc(n)$. Suppose $\Delta \vdash dacc(n) : \tau$. Then $\Delta[n] > \tau$. This implies that $\Delta[n] = \Delta.j$ for some j such that $n = |\Delta.1| + \cdots + |\Delta.j - 1|$. By lemma 3, $|\Delta.j| = |S(\Delta.j)|$. Then it is shown by a simple induction on j that $S(\Delta)[n]$ is defined and equals to $S(\Delta.j)$. By lemma 1, $S(\Delta.j) > S(\tau)$. The desired result then follows from the typing rule.

Case $iacc(n)$. By using lemma 1. ∎

Using these lemmas, we prove the following type soundness theorem.

THEOREM 1 *If $\Delta \vdash M : \tau$, $E, A, i \models \Delta$ and $E, A, i \vdash M \Downarrow v$ then $\models v : \tau$.*

Proof: By induction on the structure of the term in the style of [25]. The cases for constants and size constants are trivial.

Case $dacc(n)$. Suppose $\Delta \vdash dacc(n) : \tau$. Then there is some j such that $\Delta[n] = \Delta.j$, $n = |\Delta.1| + \cdots + |\Delta.(j - 1)|$, and $\Delta.j > \tau$. Suppose $E, A, i \vdash dacc(n) \Downarrow v$. Then $v = E[n]$. By lemma 4 $|E.i| = |\Delta.i|$ for all $1 \le i \le Elms(\Delta)$, and therefore $n = |E.1| + \cdots + |E.(j - 1)|$. Thus $E[n] = E.j$. $E, A, i \models \Delta$ implies $\models E.j : \Delta.j$. Then by lemma 4, $\models E.j : \tau$ and therefore $\models v : \tau$ as desired.

Case $iacc(n)$. Suppose $\Delta \vdash iacc(n) : \tau$. By the typing rule, $\Delta.n > \tau$. Suppose $E, A, i \vdash iacc(n) \Downarrow v$. Then $v = E[A.n]$. $E, A, i \models \Delta$ implies $A.n = |E.1| + \cdots + |E.(n - 1)|$ and $\models E.n : \Delta.n$. So $E[A.n] = E.n$. Then by lemma 4, $\models E.n : \tau$, and therefore $\models v : \tau$ as desired.

Case $\lambda^{\alpha}. M'$. Suppose $\Delta \vdash \lambda^{\alpha}. M' : \tau_1 \to \tau_2$. By the typing rules, $\Delta; \tau_1 \vdash M' : \tau_2$ and $\Delta \vdash \alpha : size(\tau_1)$. Let S be any total ground substitution. By lemma 5,

$S(\Delta); S(\tau_1) \vdash M' : S(\tau_2)$ and $S(\Delta) \vdash \alpha : size(S(\tau_1))$. Suppose $E, A, i \vdash \alpha \Downarrow v_\alpha$. By the induction hypothesis, $\models v_\alpha : size(S(\tau_1))$. Now suppose $E, A, i \vdash \lambda^\alpha. M' \Downarrow v$. v must be of the form $cls(v_\alpha, E, A, i, M')$. Let v_1 be any value such that $\models v_1 : S(\tau_1)$. Then $v_\alpha = |S(\tau_1)| = |v_1|$. Suppose $E; v_1, A; i, i + v_\alpha \vdash M' \Downarrow v_2$. By definition, $E, A, i \models \Delta$ implies $E, A, i \models S(\Delta)$. We also have: $Size(E; v_1) = Size(E) + |v_1| = Size(E) + |S(\tau_1)| = i + v_\alpha$, and $i = Size(E) = A.Elms(A) + |E.Elms(E)|$. Therefore $E; v_1, A; i, i + v_\alpha \models S(\Delta); S(\tau_1)$. Then by the induction hypothesis for M', $\models v_2 : S(\tau_2)$. This proves $\models cls(v_\alpha, E, A, i, M') : S(\tau_1) \rightarrow S(\tau_2)$. Since S is arbitrary, we have $\models cls(v_\alpha, E, A, i, M') : \tau_1 \rightarrow \tau_2$ as desired.

Case $M_1 \ M_2$. Suppose $\Delta \vdash M_1 \ M_2 : \tau$. Then there must be some τ' such that $\Delta \vdash M_1 : \tau' \rightarrow \tau$ and $\Delta \vdash M_2 : \tau'$. Let S be any total ground substitution. By lemma 5, $S(\Delta) \vdash M_1 : S(\tau') \rightarrow S(\tau)$ and $S(\Delta) \vdash M_2 : S(\tau')$. By definition, $E, A, i \models \Delta$ implies $E, A, i \models S(\Delta)$. Suppose $E, A, i \vdash M_1 \ M_2 \Downarrow v$. By the evaluation rule, there is a v_1 such that $E, A, i \vdash M_1 \Downarrow v_1$. By the induction hypothesis for M_1, $\models v_1 : S(\tau') \rightarrow S(\tau)$. Then there must exist a v_2 such that $E, A, i \vdash M_2 \Downarrow v_2$. By the induction hypothesis for M_2, $\models v_2 : S(\tau')$. By lemma 4, $|v_2| = |S(\tau')|$. By the definition of value typing, v_1 must be of the form $cls(k, E', A', i', M')$ such that $|S(\tau')| = k$. Then by the evaluation rule, $E'; v_2, A'; i', i' + k \vdash M' \Downarrow v$. Since $\models cls(k, E', A', i', M') : S(\tau') \rightarrow S(\tau)$, $\models v_2 : S(\tau')$ and $k = |v_2|$, we have $\models v : S(\tau)$. Since this holds for any S, we have $\models v : \tau$ as desired.

Case $Pair^{\alpha_1 \alpha_2}(M_1, M_2)$. Suppose $\Delta \vdash Pair^{\alpha_1, \alpha_2}(M_1, M_2) : \tau$. Then there are some τ_1, τ_2 such that $\tau = \tau_1 * \tau_2$, $\Delta \vdash M_1 : \tau_1$, $\Delta \vdash M_2 : \tau_2$, and $\Delta \vdash \alpha_i : size(\tau_i)$ ($i \in \{1, 2\}$). Let S be any total ground substitution. By lemma 5, $S(\Delta) \vdash M_i : S(\tau_i)$ and $S(\Delta) \vdash \alpha_i : size(S(\tau_i))$ ($i \in \{1, 2\}$.) $E, A, i \models \Delta$ implies $E, A, i \models S(\Delta)$. Suppose $E, A, i \vdash Pair^{\alpha_1, \alpha_2}(M_1, M_2) \Downarrow v$. By the evaluation rule, there must be v_1 such that $E, A, i \vdash M_1 \Downarrow v_1$. By the induction hypothesis for M_1, we have $\models v_1 : S(\tau_1)$. Then there must exist a v_2 such that $E, A, i \vdash M_2 \Downarrow v_2$. By the induction hypothesis for M_2, we have $\models v_2 : S(\tau_2)$. Similarly, there must exist v_{α_i} such that $E, A, i \vdash \alpha_i \Downarrow v_{\alpha_i}$. Again by the induction hypothesis, $\models v_{\alpha_i} : size(S(\tau_i))$. By the definition of typing of canonical values, $v_{\alpha_i} = |S(\tau_i)|$. By lemma 4 $|v_i| = |S(\tau_i)|$. Then $|v_i| = v_{\alpha_i}$. Then by the evaluation rule, $v = pair(v_1, v_2)$. Therefore $\models v : S(\tau_1) * S(\tau_2)$ by the definition of typing of canonical values. Since this holds for any S, we have proved $\models v : \tau_1 * \tau_2$.

Case $\Pi_\alpha M'$. Suppose $\Delta \vdash \Pi_\alpha M : \tau$. There are two cases to be considered.

Case $\alpha = 0$. There are some τ_1, τ_2 such that $\Delta \vdash M : \tau_1 * \tau_2$, and $\tau = \tau_1$. Let S be any total ground substitution. By lemma 5, $S(\Delta) \vdash M : S(\tau_1) * S(\tau_2)$.

By definition, $E, A, i \models \Delta$ implies $E, A, i \models S(\Delta)$. By the evaluation rule, there must exist a v' such that $E, A, i \vdash M \Downarrow v'$. By the induction hypothesis for M, we have $\models v' : S(\tau_1) * S(\tau_2)$. By the definition of typing of canonical values, v' must be of the form $pair(v_1, v_2)$ such that $\models v_1 : S(\tau_1)$, $\models v_2 : S(\tau_2)$. Now suppose $E, A, i \vdash \Pi_0 M \Downarrow v$. Then by the definition of evaluation, we have $v = v_1$ and therefore $\models v_1 : S(\tau_1)$. Since this holds for any S, we have proved $\models v : \tau_1$.

Case where α is one of $k, dacc(n)$ or $iacc(n)$. By the typing rule for Π, there are some τ_1, τ_2 such that $\Delta \vdash M : \tau_1 * \tau_2, \Delta \vdash \alpha : size(\tau_1)$, and $\tau = \tau_2$. Let S be any total ground substitution. By lemma 5, we have $S(\Delta) \vdash M : S(\tau_1) * S(\tau_2)$ and $S(\Delta) \vdash \alpha : size(S(\tau_1))$. By definition, $E, A, i \models \Delta$ implies $E, A, i \models S(\Delta)$. Suppose $E, A, i \vdash \Pi_\alpha M \Downarrow v$. By the definition of the evaluation rule, there must exist a v' such that $E, A, i \vdash M \Downarrow v'$. By the induction hypothesis for M, we have $\models v' : S(\tau_1) * S(\tau_2)$. By the definition of typing of canonical values, v' must be of the form $pair(v_1, v_2)$ such that $\models v_1 : S(\tau_1)$ and $\models v_2 : S(\tau_2)$. Then there must exist a v_α such that $E, A, i \vdash \alpha \Downarrow v_\alpha$. By the induction hypothesis for α, we have $\models v_\alpha : size(S(\tau_1))$. So it must be that $v_\alpha = |S(\tau_1)| = |v_1|$. Then by the evaluation rule, $v = v_2$ and therefore $\models v : S(\tau_2)$. Since this holds for any S, we have proved $\models v : \tau_2$.

Case let^α M_1 *in* M_2. Suppose $\Delta \vdash let^\alpha$ M_1 *in* $M_2 : \tau$. Then there is some τ_1 such that $\Delta \vdash M_1 : \tau_1$, $\Delta \vdash \alpha : size(\tau_1)$, and $\Delta; Clos(\Delta, \tau_1) \vdash M_2 : \tau$. Let S be any total ground substitution and $\{t_1, \ldots, t_n\} = FTV(\tau_1) \setminus FTV(\Delta)$. Let $t'_1, \ldots, t'_n$ be type variables not in $FTV(\Delta) \cup FTV(\tau_1) \cup FTV(\tau)$, and let S_1 be the substitution such that $doma(S_1) = dom(S)$, $S_1(t) = S(t)$ if $t \in dom(S) \setminus \{t_1, \ldots, t_n\}$, and $S_1(t_i) = t'_i (i \in \{1, \ldots, n\})$. By lemma 5, $S_1(\Delta) \vdash M_1 : S_1(\tau_1)$, and $S(\Delta; Clos(\Delta, \tau_1)) \vdash M_2 : S(\tau)$. By the construction of S_1, we also have $S_1(\Delta) = S(\Delta)$ and $S(\Delta; Clos(\Delta, \tau_1)) = S(\Delta); S(Clos(\Delta, \tau_1)) = S(\Delta); Clos(S(\Delta), S_1(\tau_1))$. Then $S(\Delta) \vdash M_1 : S_1(\tau_1)$, and $S(\Delta); Clos(S(\Delta), S_1(\tau_1)) \vdash M_2 : S(\tau)$. Now suppose $E, A, i \vdash let^\alpha$ M_1 *in* $M_2 \Downarrow v$. By the definition of the evaluation rule, there is some v_1 such that $E, A, i \vdash M_1 \Downarrow v_1$. By definition, $E, A, i \models \Delta$ implies $E, A, i \models S(\Delta)$. By the induction hypothesis for M_1, we have $\models v_1 : S_1(\tau_1)$. Then by the evaluation rule, there must be v_α such that $E, A, i \vdash \alpha \Downarrow v_\alpha$. Since $S(\Delta) \vdash \alpha : size(S_1(\tau_1))$, by the induction hypothesis, $\models v_\alpha : size(S_1(\tau_1))$. By definition, $v_\alpha = |S_1(\tau_1)|$. By lemma 4, $|v_1| = |S_1(\tau_1)|$. Then we have $E; v_1, A; i, i + v_\alpha \vdash M_2 \Downarrow v$. Since $\models v_1 : S_1(\tau_1)$ we have $\models v_1 : Clos(S(\Delta), S_1(\tau_1))$. By the definition of evaluation, $E; v_1, A; i, i + v_\alpha \models S(\Delta); Clos(S(\Delta), S_1(\tau_1))$. Then $\models v : S(\tau)$ follows from the induction hypothesis for M_2. Since S is arbitrary, we have proved $\models v : \tau$.

∎

4.   Type Inference and Compilation

The next step in our development is to give a translation algorithm of Core ML into the polymorphic unboxed calculus we have just defined. The basic idea is to use the type information obtained by type inference to produce a code for a polymorphic function so that it performs an appropriate action specialized to the run-time representation of an argument according to its type. The translation algorithm is divided into two stages – first to translate a given raw term to an explicitly typed term, and then to compile the explicitly typed term into a term of the unboxed polymorphic calculus.

4.1.   Explicitly typed ML

We define an explicitly typed version of Core ML similar to Core XML [7]. The set of explicitly typed terms (ranged over by X) and declarations (ranged over by B) are given by the syntax:

$$
\begin{aligned}
X \quad ::= \quad & (x \ \tau_1 \ \tau_2 \ \cdots \ \tau_n) \mid c^{b^k} \mid \lambda x : \tau. \ X \mid X \ X \mid Pair^{\tau_1 \tau_2}(X_1, X_2) \\
& \mid \ \Pi_i^{\tau_1 \tau_2} X \mid let \ x : \sigma \ = \ X \ in \ X \\
B \quad ::= \quad & \emptyset \mid B \ ; \ val \ x : \sigma \ = \ X
\end{aligned}
$$

$(x \ \tau_1 \ \cdots \ \tau_n)$ corresponds to nested type application in the second-order lambda calculus. The set of free type variables of X, denoted by $FTV(X)$, is defined by a routine induction, except for the case of let expression whose definition is given below:

$$
\begin{aligned}
& FTV(let \ x : \forall t_1 \ldots \forall t_n . \tau_1 = X_1 \ in \ X_2) \\
& = \ ((FTV(\tau_1) \cup FTV(X_1)) \setminus \{t_1, \ldots, t_n\}) \cup FTV(X_2)
\end{aligned}
$$

The set of free type variables of a declaration is defined inductively as follows:

$$
\begin{aligned}
& FTV(B; val \ x : \forall t_1 \ldots \forall t_n . \tau_1 = X_1) \\
& = \ ((FTV(\tau_1) \cup FTV(X_1)) \setminus \{t_1, \ldots, t_n\}) \cup FTV(B)
\end{aligned}
$$

with $FTV(\emptyset) = \emptyset$. Substitutions extend uniquely to explicitly typed terms.

  The type system of explicitly typed Core ML is defined in Figure 6.

  For this calculus, the following expected property holds.

LEMMA 6 *Let S be any substitution. If $\Gamma \vdash X : \tau$ then $S(\Gamma) \vdash S(X) : S(\tau)$.*

Typing rules for declarations:

$$\vdash \emptyset : \emptyset \qquad \qquad \frac{\vdash B : \Gamma \qquad \qquad \Gamma \vdash X : \tau}{\vdash (B; val\ x : Clos(\Gamma, \tau) = X) : (\Gamma; \{x : Clos(\Gamma, \tau)\})}$$

Typing rules for terms:

$$\Gamma \vdash c^{b^k} : b^k \qquad\qquad \frac{(x : \forall t_1 \ldots t_n.\ \tau) \in \Gamma}{\Gamma \vdash (x\ \tau_1 \ldots \tau_n) : \tau[\tau_1/t_1, \ldots, \tau_n/t_n]}$$

$$\frac{\Gamma; \{x : \tau_1\} \vdash X : \tau_2}{\Gamma \vdash \lambda x : \tau_1.\ X : \tau_1 \to \tau_2} \qquad\qquad \frac{\Gamma \vdash X_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash X_2 : \tau_1}{\Gamma \vdash X_1\ X_2 : \tau_2}$$

$$\frac{\Gamma \vdash X_1 : \tau_1 \quad \Gamma \vdash X_2 : \tau_2}{\Gamma \vdash Pair^{\tau_1 \tau_2}(X_1, X_2) : \tau_1 * \tau_2} \qquad\qquad \frac{\Gamma \vdash X : \tau_1 * \tau_2}{\Gamma \vdash \Pi_i^{\tau_1 \tau_2} X : \tau_i}\ i = 1, 2$$

$$\frac{\Gamma \vdash X_1 : \tau_1 \quad \Gamma; \{x : Clos(\Gamma, \tau_1)\} \vdash X_2 : \tau_2}{\Gamma \vdash let\ x : Clos(\Gamma, \tau_1)\ =\ X_1\ in\ X_2 : \tau_2}$$

Figure 6. Typing Rules for explicitly Typed Core ML

This can be proved similarly to lemma 1.

The main reason for using this calculus as an intermediate calculus for our type dependent translation is that a term in this calculus encodes all type information of a typing derivation of a given raw Core ML term. The following result is shown by Harper and Mitchell [7].

PROPOSITION 1 *There is one-to-one correspondence between typings of the explicitly typed calculus and typing derivations of Core ML.*

We define an algorithm to produce a typing of this explicitly typed calculus. The algorithm is an adaptation of Milner's type inference algorithm $\mathcal{W}$ with the following two refinements. First, we distinguish two kinds of type variables: *size sensitive type variables*, denoted by s, and size insensitive ones denoted by u. Intuitively, a type is size sensitive if its size information will be needed in translation. For example, the type of the variable x in $\lambda x.x$ is size sensitive since its size is needed to translate this term into a term in the unboxed calculus. The type inference algorithm therefore produces an explicitly typed term of the form $\lambda x : s.x$ to indicate that the size of type variable s will be needed in the translation. We use a meta variable t when we are not concerned about the difference. The unification algorithm is refined so that when unifying s and u, it produces a type sensitive variable, i.e. it returns a

substitution that substitutes s for u. This refinement can be easily formulated as a simple case of kinded type variables [20].

The second refinement is the elimination of *vacuous type variables* [20], i.e. those that appear in an explicitly typed term but not in its type or its type assignment. For example, consider the type variable t in the following typing in the explicitly typed calculus.

$$\emptyset \vdash (\lambda f : t \to t.1)\ (\lambda x : t.x) : int$$

Since t does not appear in the result type or the type assignment, it will never be further instantiated, and therefore the size of x cannot be statically determined. This is an example of a vacuous type variable. Our solution to this problem is to replace these type variables with a predefined (virtual) base type b^0 of size 0. Type safety of this is seen by observing that these type variables are associated with no typing constraint, and therefore the derivation tree obtained by replacing them with b^0 still remains a valid derivation for the same typing. For example,

$$\emptyset \vdash (\lambda f : b^0 \to b^0.1)\ (\lambda x : b^0.x) : int$$

remains a valid typing. See [20] for more discussion on vacuous type variables, and [18] where it has been shown that these type variables are the source of failure of coherence of translation from ML typings to Core XML terms, but it has also been shown that they do not affect the meaning of ML typings.

The soundness of this transformation is stated more formally as follows.

COROLLARY 1 *If* $\Gamma \vdash X : \tau$ *then* $\Gamma \vdash [b^0/t_1, \ldots, b^0/t_n](X) : \tau$ *where* $\{t_1, \ldots, t_n\} = FTV(X) \setminus (FTV(\Gamma) \cup FTV(\tau))$.

Proof: By lemma 6. ∎

Figures 7 shows the algorithm $\mathcal{W}$ which takes a type environment Γ and a term e of Core ML, and returns a substitution S, an explicitly typed term X and a type τ. Figure 8 shows the algorithm *Infer* which takes a Core ML declaration d and returns a pair (B, Γ) of a declaration of explicitly typed ML and a type environment.

For an explicitly typed term X, its *type erasure*, denoted by $erase(X)$, is the term defined below:

$$\begin{aligned} erase((x\ \tau_1 \ldots \tau_n)) &= x \\ erase(c^{b^k}) &= c^{b^k} \\ erase(\lambda x : \tau.\ X) &= \lambda x.\ erase(X) \end{aligned}$$

$\mathcal{W}(\Gamma, x) =$ if $x \notin dom(\Gamma)$ then *fail*

else let $\forall t_1 \ldots t_n. \tau = \Gamma(x)$

$\quad t'_1 \ldots t'_n$ be new s.t. if $t_i = s_i$ then $t'_i = s'_i$ and if $t_i = u_i$ then $t'_i = u'_i$

$\quad$ in $(\emptyset, (x, t'_1, \ldots, t'_n), [t'_1/t_1, \ldots, t'_n/t_n](\tau))$

$\mathcal{W}(\Gamma, c^{b^k}) = (\emptyset, c^{b^k}, b^k)$

$\mathcal{W}(\Gamma, \lambda x.\, e_1) = (S, X_1, \tau_1) = \mathcal{W}(\Gamma; \{x : s\}, e_1)$ in $(S, \lambda x : S(s).\, X_1, S(s) \to \tau_1)$ (s fresh)

$\mathcal{W}(\Gamma, e_1\, e_2) =$ let $(S_1, X_1, \tau_1) = \mathcal{W}(\Gamma, e_1)$

$\quad (S_2, X_2, \tau_2) = \mathcal{W}(S_1(\Gamma), e_2)$

$\quad S_3 = \mathcal{U}(S_2(\tau_1), \tau_2 \to u) \quad$ (u fresh)

$\quad$ in $(S_3 S_2 S_1, S_3((S_2(X_1))X_2), S_3(u))$

$\mathcal{W}(\Gamma, Pair(e_1, e_2)) =$ let $(S_1, X_1, \tau_1) = \mathcal{W}(\Gamma, e_1)$

$\quad (S_2, X_2, \tau_2) = \mathcal{W}(S_1(\Gamma), e_2)$

$\quad S_3 = \mathcal{U}(S_2(\tau_1) * \tau_2, s_1 * s_2) \quad$ (s_1, s_2 fresh)

$\quad$ in $(S_3 S_2 S_1, Pair^{S_3(s_1), S_3(s_2)}(S_3 S_2(X_1), S_3(X_2)), S_3(s_1) * S_3(s_2))$

$\mathcal{W}(\Gamma, \Pi_i e) =$ let $(S_1, X, \tau) = \mathcal{W}(\Gamma, e)$

$\quad S_2 = \mathcal{U}(\tau, s * u) \quad$ (s,u fresh)

$\quad$ in $(S_2 S_1, \Pi_i^{S_2(s) S_2(u)} S_2(X), S_2(t))$ where $t = s$ if $i = 1$ or $t = u$ if $i = 2$

$\mathcal{W}(\Gamma, let\ x = e_1\ in\ e_2) =$

$\quad$ let $(S_1, X_1, \tau_1) = \mathcal{W}(\Gamma, e_1)$

$\quad\quad S_2 = \mathcal{U}(\tau_1, s) \quad$ (s fresh)

$\quad\quad (S_3, X_2, \tau_2) = \mathcal{W}(S_2 S_1(\Gamma); \{x : \sigma\}, e_2)$ where $\sigma = Clos(S_2 S_1(\Gamma), (S_2(\tau_1)))$

$\quad\quad \sigma = \forall t_1 \ldots t_n.\, \tau$

$\quad\quad \{t'_1, \ldots, t'_k\} = FTV(X_1) \setminus (FTV(\Gamma) \cup \{t_1, \ldots, t_n\})$

$\quad\quad S_4 = [b^0/t'_1, \ldots, b^0/t'_k]$

$\quad$ in $(S_3 S_2 S_1, let\ x : S_4 S_3(\sigma) = S_4 S_3 S_2(X_1)\ in\ S_4(X_2), \tau_2)$

where $\mathcal{U}$ is the unification algorithm.

Figure 7. Type Inference Algorithm for Terms

$Infer(\emptyset) = (\emptyset, \emptyset)$
$Infer(d; val\ x = e) =$
    let $(B, \Gamma) = Infer(d)$
        $(S_1, X, \tau) = \mathcal{W}(\Gamma, e)$
        $S_2 = \mathcal{U}(\tau, s)$ (s fresh)
        $\forall t_1. \ldots . \forall t_k. S_2(\tau) = Clos(S_2 S_1(\Gamma), S_2(\tau))$
        $\{t'_1, \ldots, t'_n\} = FTV(S_2(X)) \setminus \{t_1, \ldots, t_k\}$
        $S_3 = [b^0/t'_1, \ldots, b^0/t'_n]$
    in $(B\ ;\ val\ x : \forall t_1. \ldots . \forall t_k. S_2(\tau) = S_3 S_2(X),\ \ \Gamma\ ;\ \{x : \forall t_1. \ldots . \forall t_k. S_2(\tau)\})$

Figure 8. Type Inference Algorithm for Declarations

$$erase(X_1\ X_2) = erase(X_1)\ erase(X_2)$$
$$erase(Pair^{\tau_1, \tau_2}(X_1, X_2)) = Pair(erase(X_1), erase(X_2))$$
$$erase(\Pi_i^{\tau_1 \tau_2} X) = \Pi_i erase(X)$$
$$erase(let\ x : \sigma\ = X_1\ in\ X_2) = let\ x = erase(X_1)\ in\ erase(X_2)$$

For the declaration B, its type erasure, $erase(d)$, is defined as follows:

$$erase(\emptyset) = \emptyset$$
$$erase(B; val\ x : \sigma = X) = erase(B); val\ x = erase(X)$$

We say that an occurrence of type τ in a typing $\Gamma \vdash X : \tau'$ of the explicitly typed calculus is *size sensitive* if one of the following holds:

- τ is τ_1 in a sub-term of the form $\lambda x : \tau_1. X_1$,

- τ is one of $\{\tau_1, \tau_2\}$ in a sub-term of the form $Pair^{\tau_1, \tau_2}(X_1, X_2)$,

- τ is τ_1 in a sub-term of the form $\Pi_i^{\tau_1, \tau_2} X_1$,

- τ is τ_1 in a sub-term of the form $let\ x : \forall t_1 \ldots t_n. \tau_1 = X_1\ in\ X_2$, or

- τ is one of $\{\tau_1, \ldots, \tau_n\}$ in a sub-term of the form $(x\ \tau_1 \cdots \tau_n)$ such that $\Gamma(x) = \forall t_1 \ldots t_n. \tau_0$ and t_i is a size sensitive type variable.

A typing $\Gamma \vdash X : \tau$ in the explicit calculus is *well formed* if any occurrence of a type variable is size sensitive then it is a type sensitive type variable. This definition is extended to declarations.

The algorithm $\mathcal{W}$ and *Infer* are sound and complete in the following sense.

PROPOSITION 2   *1. If $\mathcal{W}(\Gamma, e) = (S, X, \tau)$, then $S(\Gamma) \vdash e : \tau$ is a principle typing in Core ML, $S(\Gamma) \vdash X : \tau$ is a derivable well formed typing in the explicitly typed Core ML and $erase(X) = e$; if it returns failure, then e has no typing in Core ML.*

2. If $Infer(d) = (B, \Gamma)$ then $\vdash d : \Gamma$ is a principal typing in Core ML, $\vdash B : \Gamma$ is a derivable well formed typing in explicitly typed Core ML that does not contain vacuous type variables, $erase(B) = d$, and Γ is closed; if it returns failure then d has no typing.

The proof is essentially the same as in ML [3]. The additional property of well formedness is easily verified by checking each case of the algorithm.

4.2.   The translation algorithm

We define an algorithm that translates declarations of explicitly typed Core ML into declarations of the polymorphic unboxed calculus. For the algorithm to work correctly, we require that the given declaration is well formed and has no vacuous type variables. Since the type inference algorithm only produces declarations satisfying these conditions, this assumption does not impose any restriction on terms to be translated. The translation algorithm, $Comp$, is defined as a function which takes a well formed declaration of explicitly typed Core ML, and returns a pair of a type assignment Γ and a declaration D of the polymorphic unboxed calculus, using the algorithm $\mathcal{C}$ which takes a term X and a type assignment Γ of Core XML and produces a term of the polymorphic unboxed calculus. The algorithm $Comp$ and $\mathcal{C}$ are given in Figures 9 and 10.

Since we have already defined an operational semantics of the unboxed calculus, this translation algorithm defines an operational semantics of Core ML. To show the soundness of the Core ML type system with respect to this operational semantics, we define several auxiliary notions. The *size completion* of a polytype $\forall t_1 \ldots t_n. \tau$ of Core ML, denoted by $(\forall t_1 \ldots t_n. \tau)^*$, is the type $\forall t_1 \ldots t_n. size(s_1) \rightarrow \ldots \rightarrow size(s_k) \rightarrow \tau$ such that $\{s_1, \ldots s_k\} = \{s_i | s_i \in \{t_1, \ldots, t_n\}\}$. The *size completion* of a *closed* type assignment Γ, denoted by $(\Gamma)^*$, is the type assignment Γ' such that $dom(\Gamma') = dom(\Gamma)$, and $\Gamma'(x) = (\Gamma(x))^*$ for all $x \in dom(\Gamma')$. For a type assignment Γ, the *name erasure of* Γ, denoted by $Range(\Gamma)$, is the list of types obtained from Γ by erasing all variables.

We now prove that the algorithm preserves typings.

$Comp(\emptyset) = (\emptyset, \emptyset)$
$Comp(B \; ; \; val \; x : \forall t_1 \ldots t_n. \, \tau = X)$
$\quad = let \; (\Gamma, D) = Comp(B)$
$\qquad \{s_1, \ldots, s_l\} = \{s_i | s_i \in \{t_1, \ldots, t_n\}\}$
$\qquad M = \mathcal{C}(\Gamma', X) \; where \; \Gamma' = \Gamma; \{x_{s_1} : size(s_1), \ldots, x_{s_l} : size(s_l)\}$
$\qquad k = |\forall t_1 \ldots t_n. \, size(s_1) \rightarrow \cdots \rightarrow size(s_l) \rightarrow \tau|$
$\quad\;\; in \; (\Gamma; \{x : \forall t_1 \ldots t_n. \, size(s_1) \rightarrow \cdots \rightarrow size(s_l) \rightarrow \tau\},$
$\qquad D; (\underbrace{\lambda^1 \ldots \lambda^1}_{l}. \, M)^k)$

Figure 9. The Translation Algorithm for Declarations

THEOREM 2 *If $\vdash d : \Gamma$ is a well formed derivable declaration of the explicitly typed Core ML containing no vacuous type variables, then $Comp(d) = (\Gamma', D)$ such that $\vdash D : Range(\Gamma')$ and $\Gamma' = (\Gamma)^*$.*

Proof: We first show the following property on the algorithm $\mathcal{C}$:

> If $\Gamma \vdash X : \tau$ is a well formed typing of the explicitly typed Core ML and if Γ' satisfies the property that for all $x \in dom(\Gamma)$, $\Gamma'(x) = (\Gamma(x))^*$ and for any $s \in FTV(X)$, $(x_s : size(s)) \in \Gamma'$, then $\mathcal{C}(\Gamma', X) = M$ such that $Range(\Gamma') \vdash M : \tau$.

Proof is by induction on the structures of X. The case for constants is trivial, and the case for application follows directly from the induction hypothesis.

Case $(x \, \tau_1 \ldots \tau_n)$. Suppose $\Gamma \vdash (x \, \tau_1 \ldots \tau_n) : \tau$. Then $\Gamma(x)$ must be of the form $\forall t_1 \ldots t_n. \, \tau_0$ such that $\tau = S(\tau_0)$, where $S = [\tau_1/t_1, \ldots, \tau_n/t_n]$. Let Γ' satisfy the property for $\Gamma \vdash (x \, \tau_1 \ldots \tau_n) : \tau$. Also let $\{x_1 : \sigma_1, \ldots, x_i : \sigma_i, x : \sigma, \ldots\} = \Gamma'$. Then $\sigma = \forall t_1 \ldots t_n. \, size(s_1) \rightarrow \cdots \rightarrow size(s_l) \rightarrow \tau_0$, where $\{s_1, \ldots, s_l\}$ are size sensitive type variables in $\{t_1, \ldots, t_n\}$. There are two cases to be considered. First, suppose $|\sigma_j|$ is defined for all $1 \leq j \leq i$. Then $ACC(x, \Gamma') = dacc(n)$ such that $Range(\Gamma')[n] = \forall t_1 \ldots t_n. \, size(s_1) \rightarrow \cdots \rightarrow size(s_l) \rightarrow \tau_0$. Secondly, suppose $|\sigma_j|$ is undefined for some $1 \leq j \leq i$. Then $ACC(x, \Gamma') = iacc(n)$ such that $Range(\Gamma').n = \forall t_1 \ldots t_n. \, size(s_1) \rightarrow \cdots \rightarrow size(s_l) \rightarrow \tau_0$. In either case, $Range(\Gamma') \vdash ACC(x, \Gamma') : size(S(s_1)) \rightarrow \cdots \rightarrow size(S(s_l)) \rightarrow S(\tau_0)$. By the property of Γ', if $S(s_i) = s$ then $(x_s : size(s)) \in \Gamma'$. Then each α_i mentioned in the algorithm is defined and $Range(\Gamma') \vdash \alpha_i : size(S(s_i))$. Therefore we have $Range(\Gamma') \vdash ACC(x, \Gamma') \, \alpha_1 \cdots \alpha_l : \tau$.

$\mathcal{C}(\Gamma, (x \ \tau_1 \ldots \tau_n)) = $ let $(x : \forall t_1 \ldots t_n. \ size(s_1) \to \cdots \to size(s_l) \to \tau) \in \Gamma$

$$\alpha_i = \begin{cases} k & if \ |[\tau_1/t_1, \ldots, \tau_n/t_n](s_i)| = k \\ ACC(x_t, \Gamma) & if \ [\tau_1/t_1, \ldots, \tau_n/t_n](s_i) = t \ \text{and} \\ & \quad (x_t : size(t)) \in \Gamma \end{cases}$$

in $ACC(x, \Gamma) \ \alpha_1 \ldots \alpha_l$

$\mathcal{C}(\Gamma, c^{b^k}) = c^{b^k}$

$\mathcal{C}(\Gamma, \lambda x : \tau. \ X) = $ let $M = \mathcal{C}(\Gamma; \{x : \tau\}, X)$

$$\alpha = \begin{cases} k & if \ |\tau| = k \\ ACC(x_t, \Gamma) & if \ \tau = t, (x_t : size(t)) \in \Gamma \end{cases}$$

in $\lambda^\alpha. \ M$

$\mathcal{C}(\Gamma, X_1 X_2) = \mathcal{C}(\Gamma, X_1)\mathcal{C}(\Gamma, X_2)$

$\mathcal{C}(\Gamma, Pair^{\tau_1 \tau_2}(X_1, X_2)) = $ let $M_i = \mathcal{C}(\Gamma, X_i) \ (i = 1, 2)$

$$\alpha_i = \begin{cases} k & if \ |\tau_i| = k \\ ACC(x_t, \Gamma) & if \ \tau_i = t, (x_t : size(t)) \in \Gamma \end{cases}$$

in $Pair^{\alpha_1 \alpha_2}(M_1, M_2)$

$\mathcal{C}(\Gamma, \Pi_i^{\tau_1 \tau_2} X) = $ let $M = \mathcal{C}(\Gamma, X)$

in case $i = 1 : \Pi_0 M$

case $i = 2 :$ let $\alpha = \begin{cases} k & if \ |\tau_1| = k \\ ACC(x_t, \Gamma) & if \ \tau_1 = t, (x_t : size(t)) \in \Gamma \end{cases}$

in $\Pi_\alpha M$

$\mathcal{C}(\Gamma, let \ x = \forall t_1 \ldots t_n. \ \tau = X_1 \ in \ X_1) =$

let $\{s_1 \ldots s_l\} = \{s_i | s_i \in \{t_1 \ldots t_n\}\}$

$\quad M_1 = \mathcal{C}(\Gamma; \{x_{s_1} : size(s_1), \ldots, x_{s_l} : size(s_l)\}, X_1)$

$\quad M_2 = \mathcal{C}(\Gamma; \{x : \forall t_1 \ldots t_n. \ size(s_1) \to \ldots \to size(s_l) \to \tau\}, X_2)$

$$\alpha = \begin{cases} k & if \ |\forall t_1 \ldots t_n. \ size(s_1) \to \ldots \to size(s_l) \to \tau| = k \\ ACC(x_t, \Gamma) & if \ \forall t_1 \ldots t_n. \ \tau = t \ \text{and} \ (x_t : size(t)) \in \Gamma \end{cases}$$

in $let^\alpha \underbrace{\lambda^1 \ldots \lambda^1}_{l}. \ M_1 \ in \ M_2$

where $ACC(x, \Gamma) = \begin{cases} dacc(n) & if \ \Gamma = [x_1 : \sigma_1; \ldots; x_k : \sigma_k x : \sigma; \ldots], \ \text{and} \\ & \quad n = |\sigma_1| + \cdots + |\sigma_k| \\ iacc(n) & if \ \Gamma = [x_1 : \sigma_1; \ldots; x_k : \sigma_k; x : \sigma; \ldots], \\ & \quad \exists i (1 \le i \le k) \ |\sigma_i| \ \text{is undefined and} \ n = k + 1 \end{cases}$

Figure 10. The Translation Algorithm for Typings

Case $\lambda x : \tau_1.\, X'$. Suppose $\Gamma \vdash \lambda x : \tau_1.\, X' : \tau_1 \to \tau_2$. Then $\Gamma; \{x : \tau_1\} \vdash X' : \tau_2$. Let Γ' satisfy the condition for $\Gamma \vdash \lambda x : \tau_1.\, X' : \tau_1 \to \tau_2$. Then $\Gamma'; \{x : \tau_1\}$ satisfies the condition for $\Gamma; \{x : \tau_1\} \vdash X' : \tau_2$. Let $M' = \mathcal{C}(\Gamma'; \{x : \tau_1\}, X')$. By the induction hypothesis, we get $Range(\Gamma'; \{x : \tau\}) \vdash M' : \tau_2$. By the definition of $\mathcal{C}$, $M = \lambda^\alpha.\, M'$ where $\alpha = |\tau_1|$ if it is defined otherwise τ is a type variable s and $\alpha = ACC(x_s, \Gamma')$. In the latter case, $(x_s : size(s)) \in \Gamma'$ by the assumption on Γ'. Therefore $ACC(x_s, \Gamma')$ is defined. Thus in either case, $Range(\Gamma') \vdash \alpha : size(\tau_1)$. Then we have $Range(\Gamma') \vdash \lambda^\alpha.\, M' : \tau_1 \to \tau_2$ by the typing rules.

Case $Pair^{\tau_1 \tau_2}(X_1, X_2)$. Suppose $\Gamma \vdash Pair^{\tau_1 \tau_2}(X_1, X_2) : \tau_1 * \tau_2$. Then $\Gamma \vdash X_1 : \tau_1$, $\Gamma \vdash X_2 : \tau_2$. Let Γ' satisfy the condition for $\Gamma \vdash Pair^{\tau_1 \tau_2}(X_1, X_2) : \tau_1 * \tau_2$. Let $M_1 = \mathcal{C}(\Gamma', X_1)$ and $M_2 = \mathcal{C}(\Gamma', X_2)$. Since Γ' satisfies the condition for $\Gamma \vdash X_1 : \tau_1$ and $\Gamma \vdash X_2 : \tau_2$, by the induction hypothesis we have $Range(\Gamma') \vdash M_1 : \tau_1$ and $Range(\Gamma') \vdash M_1 : \tau_1$. By the definition of $\mathcal{C}$, $M = Pair^{\alpha_1 \alpha_2}(M_1, M_2)$ where $\alpha_i = |\tau_i|$ if it is defined, otherwise τ_i is a type variable s_i and $\alpha_i = ACC(x_{s_i}, \Gamma')$. In the latter case there must be $(x_{s_i} : size(s_i)) \in \Gamma'$ by the condition on Γ'. In either case we have $Range(\Gamma') \vdash \alpha_i : size(\tau_i)$. $Range(\Gamma) \vdash Pair^{\alpha_1, \alpha_2}(X_1, X_2) : \tau_1 * \tau_2$ follows from the typing rule.

Case $\Pi_i^{\tau_1 \tau_2} X'$. The case for Π_1 follow directly from the induction hypothesis.

Suppose $\Gamma \vdash \Pi_2^{\tau_1 \tau_2} X' : \tau_2$. Then $\Gamma \vdash X' : \tau_1 * \tau_2$. Let Γ' satisfy the condition for $\Gamma \vdash \Pi_2^{\tau_1 \tau_2} X' : \tau_2$. Let $M' = \mathcal{C}(\Gamma', X')$. Γ' also satisfies the condition for $\Gamma \vdash X' : \tau_1 * \tau_2$. By the induction hypothesis, $Range(\Gamma') \vdash M' : \tau_1 * \tau_2$. Now let $M = \mathcal{C}(\Gamma', \Pi_2^{\tau_1 \tau_2} X')$. Then $M = \Pi_\alpha M'$ where $\alpha = |\tau_1|$ if it is defined or τ_1 is a type variable s and $\alpha = ACC(x_s, \Gamma')$. In the latter case there must be $(x_s : size(s)) \in \Gamma'$ by the condition on Γ'. Therefore in either case we have $Range(\Gamma') \vdash \alpha : size(\tau_1)$. Then we have $Range(\Gamma') \vdash \Pi_\alpha M' : \tau_2$.

Case $let\ x : \sigma = X_1\ in\ X_2$. Suppose $\Gamma \vdash let\ x : \sigma = X_1\ in\ X_2 : \tau$. Then there is some τ_1 such that $\Gamma \vdash X_1 : \tau_1$, $\Gamma; \{x : \forall t_1 \ldots t_n.\, \tau_1\} \vdash X_2 : \tau$, and $\forall t_1 \ldots t_n.\, \tau_1 = Clos(\Gamma, \tau_1)$. Let Γ' satisfy the property for $\Gamma \vdash let\ x : \sigma = X_1\ in\ X_2 : \tau$, and let

$$
\begin{aligned}
\{s_1, \ldots, s_l\} &= \{s | s \in \{t_1, \ldots, t_n\}\} \\
M_1 &= \mathcal{C}(\Gamma'\{x_{s_1} : size(s_1), \ldots, x_{s_l} : size(s_l)\}, X_1) \\
M_2 &= \mathcal{C}(\Gamma'\{x : \forall t_1 \ldots t_n.size(s_1) \to \cdots \to size(s_l) \to \tau\}, X_2)
\end{aligned}
$$

Since $\Gamma'\{x_{s_1} : size(s_1), \ldots, x_{s_l} : size(s_l)\}$ satisfies the condition for $\Gamma \vdash X_1 : \tau_1$, by the induction hypothesis $Range(\Gamma'; \{size(s_1), \ldots, size(s_l)\}) \vdash M_1 : \tau_1$. Since $(*\sigma) = \forall t_1 \ldots t_n.\, size(s_1) \to \ldots \to size(s_l) \to \tau_1$, $\Gamma'\{x : \forall t_1 \ldots t_n.size(s_1) \to \cdots \to size(s_l) \to \tau\}$ satisfies the condition for $\Gamma; \{x : \sigma\} \vdash X_2 : \tau_2$. Then by the induction

hypothesis $Range(\Gamma'; \{\forall t_1 \ldots t_n.\, size(s_1) \rightarrow \ldots \rightarrow size(s_l) \rightarrow \tau_1\}) \vdash M_2 : \tau_2$. Now suppose $M = \mathcal{C}(\Gamma', let\ x : \sigma = X_1\ in\ X_2)$. Then $M = let^\alpha \underbrace{\lambda^1 \ldots \lambda^1}_{l}.M_1\ in\ M_2$ where

$\alpha = |\forall t_1 \ldots t_n.size(s_1) \rightarrow \ldots \rightarrow size(s_l) \rightarrow \tau_1|$ if it is defined, otherwise $\forall t_1 \ldots t_n.\tau_1$ is a type variable s and $\alpha = ACC(x_s, \Gamma')$. In this case, $(x_s : size(s)) \in \Gamma'$ by the condition on Γ'. The desired conclusion then follows by the typing rule.

Using this property of the algorithm $\mathcal{C}$, we prove the theorem by induction on the length of the declaration. The basis is trivial.

Let the given declaration be of the form $\vdash (B; val\ x : \forall t_1 \ldots t_n.\, \tau = X)\ :$ $(\Gamma; \forall t_1 \ldots t_n.\, \tau)$. Then $\vdash B : \Gamma$, $\Gamma \vdash X : \tau$, and $\forall t_1 \ldots t_n.\, \tau = Clos(\Gamma, \tau)$. Let $(\Gamma', D) = Comp(B)$. By the induction hypothesis, $\vdash D : Range(\Gamma')$ and $\Gamma' = (\Gamma)^*$. Let $\{s_1, \ldots, s_l\} = \{s | s \in \{t_1, \ldots, t_n\}\}$. Since Γ is closed, $\{t_1, \ldots, t_n\} = FTV(\tau)$. Then since $B; val\ x : \forall t_1 \ldots t_n.\, \tau = X$ does not contain vacuous type variables, $\Gamma'\{x_{s_1} : size(s_1), \ldots, x_{s_l} : size(s_l)\}$ satisfies the condition of proposition we have just proved for $\Gamma \vdash X : \tau$. Therefore $\mathcal{C}(\Gamma'\{x_{s_1} : size(s_1), \ldots, x_{s_l} : size(s_l)\}, X) = M$ such that $Range(\Gamma'\{x_{s_1} : size(s_1), \ldots, x_{s_l} : size(s_l)\}) \vdash M : \tau$. Then the algorithm for $B; val\ x : \forall t_1 \ldots t_n.\tau = X$ succeeds with $(\Gamma'; \{x : \forall t_1 \ldots t_n.size(s_1) \rightarrow \cdots \rightarrow size(s_l) \rightarrow \tau\}, D; \underbrace{\lambda^1 \ldots \lambda^1}_{l}.\, M)$. By the typing rule,

$$Range(\Gamma') \vdash \underbrace{\lambda^1 \ldots \lambda^1}_{l}.\, M : size(s_1) \rightarrow \cdots \rightarrow size(s_l) \rightarrow \tau$$

Since Γ is closed, $Clos(\Gamma', size(s_1) \rightarrow \cdots \rightarrow size(s_l) \rightarrow \tau) = \forall t_1 \ldots t_n.size(s_1) \rightarrow \cdots \rightarrow size(s_l) \rightarrow \tau$. Then we have

$$\vdash D; \underbrace{\lambda^1 \ldots \lambda^1}_{l}.\, M\ :\ Range(\Gamma'\{x : \forall t_1 \ldots t_n.size(s_1) \rightarrow \cdots \rightarrow size(s_l) \rightarrow \tau\})$$

as desired.   ∎

Since $\tau^* = \tau$ for any ground monotype, this theorem guarantees that the translation preserves all the monotypes of declarations. By combining the soundness theorem (Theorem 1) and the definition of evaluation of declaration, we have the following desirable property.

COROLLARY 2 *Let* $\vdash B : (\Gamma; \{x : \sigma\})$ *be a well formed declaration of the explicitly typed Core ML containing no vacuous type variable. Then* $Comp(B) = ((\Gamma'; \{x : \sigma'\}), D)$ *such that* $(\Gamma; \{x : \sigma\})^* = \Gamma'; \{x : \sigma'\}$ *and if* $D \Downarrow v$ *then* $\models v : \sigma'$.

Since the type inference algorithm always produces a declaration containing no vacuous type variable, this implies the following desirable property.

COROLLARY 3 *Core ML type system is sound with respect to the operational se-mantics realized by the combination of the translation and the operational semantics of the polymorphic unboxed calculus.*

We end this section by showing some examples of translation. The first is a simple polymorphic declaration.

$$\{val\ twice = \lambda x.\lambda f.f(f\ x)\}\ :\ \{twice : \forall s.s \to (s \to s) \to s\}$$

which is compiled into the following declaration in the polymorphic unboxed cal-culus:

$$\{\lambda^1.\lambda^{dacc(0)}.\lambda^1.(iacc(3)\ (iacc(3)\ dacc(1)))\}\ :\ \{\forall s.size(s) \to s \to (s \to s) \to s\}$$

The next one involves size applications. Consider the following declaration:

$$\{val\ twice = \lambda x.\lambda f.f(f\ x); val\ it = twice\ 3.14\ (\lambda x.x)\}$$

This is converted into the following declaration of the explicitly typed calculus:

$$\{\ val\ twice : \forall s.\ s \to (s \to s) \to s = \lambda x : s.\lambda f : s \to s.f(f\ x)\ ;$$
$$val\ it : real = (twice\ real)\ 3.14\ \lambda x : real.x\ \}$$

This is translated into the the unboxed calculus as follows:

$$\{\lambda^1.\lambda^{dacc(0)}.\lambda^1.(iacc(3)\ (iacc(3)\ dacc(1))); dacc(0)\ 2\ 3.14\ (\lambda^2.dacc(1))\}$$
$$: \{\forall s.size(s) \to s \to (s \to s) \to s;\ real\}$$

When this declaration is evaluated, it produces the expected result.

5.   Further Refinements

There are a number of ways to refine the polymorphic unboxed calculus and the compilation method presented in this paper. In this section, we discuss some of them.

5.1.   Recursive data types and other type constructors

The described method can be easily extended to ML-style recursive data types. The only necessary assumption is that the size of each top-level type constructor in a recursive type is the same. Since the usual use of recursive types is to define dynamic data structures such as lists or trees where each top-level constructor is a pointer, this condition is naturally satisfied. Under this restriction, for example, list type of the form

```
datatype 'a list = nil | 'a * ('a list)
```

can be introduced without any problem. Note that in this type, pair constructor (i.e. `cons` constructor) is required to be boxed while the component type `'a` has its natural representation. We can then have, for example, `real list` as a list containing unboxed real values, and those lists can be directly handled with polymorphic functions such as `fold` without incurring boxing-unboxing coercion overhead.

5.2. Recursive functions

Recursive functions can be introduced by extending Core ML and the polymorphic unboxed calculus with fixed point constructors of the forms $fix(f, x).M$ and $fix^\alpha.M$, respectively. Their typing rules are given below.

$$\frac{\Gamma; \{f : \tau_1 \to \tau_2\}; \{x : \tau_1\} \vdash M : \tau_2}{\Gamma \vdash fix(f, x).M : \tau_1 \to \tau_2} \qquad \frac{\Delta; \tau_1 \to \tau_2; \tau_1 \vdash M : \tau_2 \quad \Delta \vdash \alpha : size(\tau_1)}{\Delta \vdash fix^\alpha.M : \tau_1 \to \tau_2}$$

By implementing canonical values of *fix-closures* representing $fix^\alpha.M$ to have the same size as ordinary closures, the operational semantics of the unboxed calculus can be extended. More specifically, the operational semantics of the polymorphic unboxed calculus can be extended with the following rules.

$$\frac{E, A, i \vdash \alpha \Downarrow k}{E, A, i \vdash fix^\alpha.\, M \Downarrow fix(k, E, A, i, M)}$$

$$\frac{\begin{array}{c} E_1, A_1, i_1 \vdash M_1 \Downarrow fix(k, E_2, A_2, i_2, M_3) \\ E_1, A_1, i_1 \vdash M_2 \Downarrow v_1 \\ E_2; fix(k, E_2, A_2, i_2, M_3); v_1, A_2; i_2; i_2 + 1, i_2 + 1 + k \vdash M_3 \Downarrow v_2 \end{array}}{E_1, A_1, i_1 \vdash M_1 \ M_2 \Downarrow v_2} \quad \text{if } |v_1| = k$$

where $fix(k, E, A, i, M)$ is a fix-closure. For this extended calculus, the type soundness theorem can be proved by using the technique of [25] or [13]. Type inference and compilation can be directly extended for fix construct without much difficulty.

5.3. More refined size information

One feature that has contributed to simplify the method presented here is that the size of a given type is determined by its topmost type constructor. Because of this

property we had only to insert size abstraction for each type variable s, since once s is instantiated to some type, its size is statically computed. This assumption no longer holds if we include some type constructors, such as an unboxed product, whose size depends on the size of component types. One problem in allowing such a type constructor is the treatment of polytypes. If we allow such type constructors, then computation of the size of a type in compiling polymorphic functions may require arithmetic operations. For example, consider the term

$$\lambda x.\lambda y.\lambda f.f(x,y) : \forall t_1.\forall t_2.\forall t_3.t_1 \rightarrow t_2 \rightarrow (t_1 \times t_2 \rightarrow t_3) \rightarrow t_3$$

where (x,y) is assumed to be unboxed product whose size is the sum of the sizes of x and y. In our approach, compilation of this term would require to treat $size(t_1) + size(t_2)$ as a type. It is an interesting future investigation to extend our framework so that the set of types can contain not only atomic values but also an algebra.

A simpler alternative approach is to restrict polytypes to be those $\forall t_1.\ldots.\forall t_n.\tau$ such that the size of τ does not depend on instantiations of $t_1, \ldots, t_n$. Except for a few polymorphic constants, we believe that this condition is satisfied in most of the cases. We can then obtain a simple and practical solution by placing this condition on general polymorphic functions and handling those small number of constants in a special way. Under this restriction, we believe that the method presented in this paper can be extended to those type constructors in the following way. Instead of coupling size abstractions with type variables, we maintain the set of types for which size information is needed. To do this, we change the algorithm $\mathcal{W}$ so that it infers an explicitly typed term X together with the set of types T whose size information is needed for compiling X. Then at the time of translating a top level declaration or a let binding, we insert a size abstraction of type $size(\tau)$ for each τ in T that contains some type variables that are bound by the type scheme.

5.4. Preservation of the order of evaluation

To make our presentation simple, we have represented both of the ordinary function abstraction and the size abstraction by the same lambda abstraction. This causes the problem of changing the order of evaluation of let bound values under the usual call-by-value evaluation, since our translation algorithm may insert size abstraction for let bound values. One way to solve this problem without making any restrictions on polymorphic programs is to introduce a new term constructor,

say $\mu.M$, for size abstraction, and to change the operational semantics of the unboxed calculus in such a way that it evaluate inside of the size abstraction. The interested reader is referred to [20], where this approach has been formalized in the context of compiling a polymorphic record calculus, and it has shown that under this strategy the compilation preserves the order of evaluation.

6. Related Works

This work was motivated by the work of Morrison et. al. [17] for an ad-hoc approach to implement polymorphic function, and a method for mixed representation optimization of Peyton Jones and Launchbury [22] and of Leroy [14] (see also [10, 24] for further refinements.) As already mentioned in the introduction, the new contribution of our work compared to these works is a method to specialize polymorphic functions themselves without resorting to run-time type analysis. It does not require coercion functions between boxed and unboxed representations when using polymorphic functions. Another advantage of our approach is that it works well for bulk data types such as lists or trees.

In our method, we used size abstraction and size application to specialize polymorphic code for efficient execution. This idea and its technical development are based on one of the authors' work [20] of compiling polymorphic record calculus. In a more general perspective, this method can be characterized as type dependent specialization of polymorphic functions. In this perspective, our work shares the same motivation as the paradigm of "intensional type analysis" [8, 9]. In their framework, a polymorphic function is translated to a function that takes a type as an extra parameter and performs type analysis of the actual argument at run-time. This method is powerful enough to express a wide range of type dependent behavior. It is, however, not immediately obvious whether or not type passing system contributes to efficient implementation. For example, if a type is passed at run-time, then the run-time system can certainly computes the offset of a field in a record, but this strategy is equivalent to find the position of a label in a labeled record at run-time and does not necessary contribute to efficient code. The important point is to pass not a type itself but only those attributes of a type that are relevant for efficient execution. Developing a systematic method to achieve this effect is far from trivial. In the case of unboxed execution, the most relevant information is the size of the type. The major technical contribution of the present paper is to have established a systematic method to use this information in evaluating a polymorphic function, i.e. an unboxed operational semantics.

There are also several recent papers on efficient implementation of polymorphic languages using explicit type information. Examples include tag-free garbage collection [26] by translating a raw term to an explicitly typed second-order term, specialization of Haskell type classes [27] using type information [6, 21], and list representation optimization [5, 23]. Since program translations in all of these approaches and ours appear to share some general structure, a detailed comparison among them may shed the light on better understanding of the general property of a type inference approach to program specialization.

7.   Conclusion and Implementation Consideration

We have presented an unboxed operational semantics for an ML-style polymorphic programming language. We have first defined a polymorphic unboxed calculus as an abstract machine for unboxed implementation of a polymorphic language, given its formal operational semantics, and shown its type soundness. An important new feature of this unboxed calculus is that it allows polymorphic functions to be evaluated efficiently without using boxed representations nor run-time type information. This has been achieved by the technique of representing a variable as an index that corresponds to the actual offset to the value in a run-time environment containing unboxed objects, and an abstraction mechanism over size information. We have then given a translation algorithm from ML into the unboxed calculus, and have shown that the translation preserves typings. Combined with the type soundness of the unboxed calculus, these results establish the type soundness of ML with respect to the operational semantics realized by the translation.

We believe that the unboxed semantics serves as a basis for more efficient compilation of ML-style polymorphic languages. In the perspective of implementation, there are two sources of run-time overhead of our method. The first is the extra function applications to pass the necessary size information. This is done only when a polymorphic function is instantiated. Since in a loop or recursion, a polymorphic function has the same instance type and therefore the instantiation is done once outside of the loop or recursion, we expect that this overhead is negligible. Another run-time overhead of our approach is that some variables (those of the form $iacc(n)$) are accessed indirectly through an offset table. This indirection is necessary only for those local lambda variables that are defined inside of a polymorphic function whose argument size may vary. Accesses of predefined identifiers in an environment or lambda variables that are not inside of a polymorphic function are

compiled into directly accessible variables (of the form $dacc(n)$). So again we think that this overhead is not significant.

In order to use the presented method in an actual optimizing compiler, it is important to analyze the interaction between the method and various existing optimization techniques. As a future work, we would like to develop a compiler using the presented method with other existing optimization techniques. Jointly with Oki Electric Industry, we have started a compiler construction project for an ML-style polymorphic programming language incorporating various advanced features such as the unboxed semantics presented in this article and record polymorphism [20].

Acknowledgments

References

1. 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. Lecture Notes in Computer Science, vol. 528. Springer-Verlag, Berlin, 1991.

2. G. Cousineau, P-L. Curien, and M. Mauny. The categorical abstract machine. *Science of Computer Programming*, 8(2):173–202, 1987.

3. L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 207–212. ACM, New York, 1982.

4. N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indag. Math.*, 34:381–392, 1972.

5. C. Hall. Using Hindley-Milner type inference to optimize list representation. In *Proceedings of the ACM Conference on Lisp and Functional Programming*. ACM, New York, 1994.

6. C. Hall, K. Hammond, S. Peyton Jones, and P. Wadler. Type class in Haskell. Technical report, University of Glasgow, 1994.

7. R. Harper and J. C. Mitchell. On the type structure of Standard ML. *ACM Trans. Program. Lang. Syst.*, 15(2):211–252, 1993.

8. R. Harper and G. Morrisett. Compiling with non-parametric polymorphism. Technical Report CMU-CS-94-122, School of Computer Science, Carnegie Mellon University, 1994.

9. R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 130–141. ACM, New York, 1995.

10. F. Henglein and J. Jørgensen. Formally optimal boxing. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 213–226. ACM, New York, 1994.

11. 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. *SIG-PLAN Notices, Haskell special issue*, 27(5), 1992.

12. G. Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, pages 22–39. Springer Verlag, Berlin, 1987.

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

14. X. Leroy. Unboxed objects and polymorphic typing. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 177–188. ACM, New York, 1992.

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

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

17. R. Morrison, A. Dearle, R.C.H. Connor, and A.L. Brown. An ad hoc approach to the implementation of polymorphism. *ACM Trans. Program. Lang. Syst.*, 13(3):342–371, 1991.

18. A. Ohori. A simple semantics for ML polymorphism. In *Proceedings of the ACM/IFIP Conference on Functional Programming Languages and Computer Architecture*, pages 281–292. ACM, New York, 1989.

19. A Ohori. A compilation method for ML-style polymorphic record calculi. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 154–165. ACM, New York, 1992.

20. A. Ohori. A polymorphic record calculus and its compilation. *ACM Trans. Program. Lang. Syst.*, 17(6):844–895, 1995.

21. John Peterson and Mark Jones. Implementing type classes. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 227-236. ACM, New York, 1993.

22. S.L. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a nonstrict functional language. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 636–666. Lecture Notes in Computer Science, Vol 523. Springer-Verlag, Berlin, 1991.

23. Z. Shao, J.H. Reppy, and A. W. Apple. Unrolling lists. In *Proceedings of the ACM Conference on Lisp and Functional Programming*. ACM, New York, 1994.

24. P. Thiemann. Unboxed values and polymorphic typing revisited. In *Proceedings of the International Conference on Functional Programming Languages and Computer Architecture*, pages 24–35, 1995.

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

26. A Tolmach. Tag-free garbage collection using explicit type parameters. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 1–11. ACM, New York, 1994.

27. P. Wadler and S. Blott. How to make ad hoc polymorphism less ad hoc. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, New York, 1989.