Compiling Polymorphism Using Intensional Type Analysis*

Robert Harper† Greg Morrisett‡
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891

Abstract

Traditional techniques for implementing polymorphism use a universal representation for objects of unknown type. Often, this forces a compiler to use universal representations even if the types of objects are known. We examine an alternative approach for compiling polymorphism where types are passed as arguments to polymorphic routines in order to determine the representation of an object. This approach allows monomorphic code to use natural, efficient representations, supports separate compilation of polymorphic definitions and, unlike coercion-based implementations of polymorphism, natural representations can be used for mutable objects such as refs and arrays.

We are particularly interested in the typing properties of an intermediate language that allows run-time type analysis to be coded within the language. This allows us to compile many representation transformations and many language features without adding new primitive operations to the language. In this paper, we provide a core target language where type-analysis operators can be coded within the language and the types of such operators can be accurately tracked. The target language is powerful enough to code a variety of useful features, yet type checking remains decidable. We show how to translate an ML-like language into the target language so that primitive operators can analyze types to produce efficient representations. We demonstrate the power of the "user-level" operators by coding flattened tuples, marshalling, type classes, and a form of type dynamic within the language.

1 Introduction

Many compilers assume a universal or "boxed" representation of a single machine word if the type of a value is unknown. This allows the compiler to generate one simple piece of code to manipulate the value. But boxed representations often require more space and provide less efficient access than natural representations. For example, an array of small unknown objects, such as booleans or characters, is represented as an array of words, wasting the majority of the space. An object larger than a word, such as a double-precision floating-point value, is allocated and a pointer is used in place of the value. Consequently, accessing the value requires an additional memory access. As word sizes increase from 32 to 64-bits, and memory latencies increase, it becomes increasingly important to minimize boxing.

In modern programming languages such as Modula-3, Standard ML (SML), and Haskell, unknown types and thus boxed representations arise because of two key language features: types imported from a separately compiled program unit and types within polymorphic routines. Polymorphic values are particularly troublesome because we can simultaneously *view* them as having any one of an infinite number of monomorphic types. For example, a polymorphic routine that maps a function across the elements of an array can be simultaneously seen as a function that works on boolean arrays and a function that works on real arrays. The routine can thus be used in place of a function that was compiled knowing whether the argument array contains booleans or reals. Consequently, monomorphic routines are forced to use the same representations as polymorphic routines and the entire program pays the price of the increased space and execution-time overheads of the universal representations.

1.1 Coercion Implementations

The problem with polymorphism stems from the assumption that *viewing* a polymorphic value as a monomorphic value should have no computational effect. Recent work by Leroy [30] and others [41, 24, 43] has suggested that the instantiation of a polymorphic value should correspond to a *run-time coercion* from the universal representation to the appropriate specialized representation. At function types, this requires the dual coercion (for the function argument) that converts specialized representations to the universal representation. For example, when the identity function of type $\forall\alpha.\alpha \rightarrow \alpha$ is instantiated to have type $\mathsf{int} \rightarrow \mathsf{int}$, a coercion is generated that takes an integer argument, boxes it, passes it to the identity function, and unboxes the result. This approach allows monomorphic code to use the natural, efficient representations.

Leroy's coercions produce an isomorphic copy of a data structure. For example, to coerce a tuple, we project the

*This work was sponsored by the Advanced Research Projects Agency, CSTO, under the title "The Fox Project: Advanced Development of Systems Software", ARPA Order No. 8313, issued by ESD/AVS under Contract No. F19628–91–C–0168.

†E-mail: rwh@cs.cmu.edu
‡E-mail: jgmorris@cs.cmu.edu

components of the tuple, box/unbox them, and then form a new tuple. Unfortunately, copying coercions are impractical for large data structures since the cost of making the copy often outweighs the benefits of the unboxed representation (as pointed out by Leroy [30, page 184]). More problematically, copying coercions do not work for mutable data structures such as arrays. If we make a copy of the value to box the components then updates to the copy will not be reflected in the original array and vice versa.

1.2 Type Passing

An alternative approach to coercions, first suggested by the Napier '88 implementation [37], is to pass the types that are unknown at compile-time to primitive operations at link-time or even run-time. Then the primitive operations can analyze the type in order to select the appropriate code to manipulate the natural representation of an object. For example, a polymorphic subscript function for arrays might be compiled into the following pseudo-code:

$$\begin{array}{l}\mathsf{sub} = \Lambda\alpha.\mathsf{typecase}\ \alpha\ \mathsf{of} \\ \qquad\qquad\quad \mathsf{bool} \Rightarrow \mathsf{boolsub} \\ \qquad\qquad\quad | \,\mathsf{real} \Rightarrow \mathsf{realsub} \\ \qquad\qquad\quad | \,\tau \Rightarrow \mathsf{boxedsub}[\tau]\end{array}$$

Here, sub is a function that takes a *type* argument (α), and does a case analysis to determine the appropriate specialized subscript function that should be returned. For example, sub[bool] returns the boolean subscript function that expects an array of bits, while sub[real] returns the floating point subscript function that expects a double-word aligned array of floating point values. For all other types, we assume the array has boxed components and thus return the boxed subscript function.

If the sub operation is instantiated with a type that is known at compile-time (or link-time), then the overhead of the case analysis can be eliminated by duplicating and specializing the definition of sub at the appropriate type. For example, the source expression "$\mathsf{sub}\langle x, 4\rangle + 3.14$" will be compiled to the target expression "$\mathsf{sub}[\mathsf{real}]\langle x, 4\rangle + 3.14$" since the result of the sub operation is constrained to be a real. If the definition of sub is inlined into the target expression and some simple reductions are peformed, this yields the optimized expression "$\mathsf{realsub}\langle x, 4\rangle + 3.14$". Thus, parameterizing the primitive operations by type provides a single, consistent methodology for type analysis at compile-time, link-time, and run-time.

In languages where polymorphic definitions are restricted to "computational values" (essentially constants and functions), polymorphic definitions can always be duplicated and specialized or even inlined. Lazy languages such as Haskell satisfy this constraint, and Wright has determined empirically that such a restriction does not effect the vast majority of SML programs [52]. Languages like core-SML and Haskell only allow polymorphic values to arise as the result of a "let" binding and restrict the type of such values to be prenex-quantified. That is, the type must be of the form $\forall\alpha_1, \ldots, \alpha_n.\tau$ where τ contains no quantifier. Thus, the only thing that can be done to a polymorphic value is to instantiate it. Since the scope of a let is closed, it is possible to determine all of the instantiations of the polymorphic value at compile time and eliminate *all* polymorphism through duplication andd specialization. Such an approach is used, for instance, by Blelloch *et al.* in their NESL compiler [6] and more recently by Jones to eliminate Haskell overloading [27]. Furthermore, Jones reports that this approach does not lead to excessive code-blowup.

Unfortunately, eliminating all of the polymorphism in a program is not always possible or pratical. In particular, there is no way to eliminate the polymorphism when separately compiling a definition from its uses because it is impossible to determine the types at which the definition will potentially be used. This prevents us from separately compiling polymorphic libraries or polymorphic definitions entered at a top-level loop. Furthermore, in languages that allow polymorphic values to be "first-class" such as XML [21] and Quest [9], it is impossible to eliminate all polymorphism at compile-time. Therefore, we view duplication and specialization as an important optimization, but consider some run-time type analysis to still be necessary for practical language implementation.

1.3 Type-Checking Type Analysis

In this paper, we show how to compile ML-like polymorphic languages to a target language where run-time type analysis may be used by the primitive operations to determine the representation of a data structure. We are particularly interested in the *typing* properties of a language that allows run-time type analysis. The sub definition above is ill-typed in ML because it must simultaneously have the types $\mathsf{boolarray} \times \mathsf{int} \to \mathsf{bool}$, $\mathsf{realarray} \times \mathsf{int} \to \mathsf{real}$, as well as $\forall\alpha.(\alpha)\mathsf{boxedarray} \times \mathsf{int} \to \alpha$. Since boolarray and realarray are nullary constructors and *not* instantiations of (α)boxedarray, it is clear that there is no ML type that unifies all of these types.

Our approach to this problem is to consider a type system that provides analysis of types via a type-level "Typecase" construct. For example, the sub definition above can be assigned a type of the form:

$$\forall\alpha.\mathsf{SpclArray}[\alpha] \times \mathsf{int} \to \alpha$$

where the specialized array constructor SpclArray is defined using Typecase as follows:

$$\begin{array}{lcl}\mathsf{SpclArray}[\alpha] & = & \mathsf{Typecase}\ \alpha\ \mathsf{of} \\ & & \quad \mathsf{bool} \Rightarrow \mathsf{boolarray} \\ & & \quad | \,\mathsf{real} \Rightarrow \mathsf{realarray} \\ & & \quad | \,\tau \Rightarrow (\tau)\mathsf{boxedarray}\end{array}$$

The definition of the constructor parallels the definition of the term: If the parameter α is instantiated to bool, then the resulting type is boolarray and if the parameter is instantiated to real, the resulting type is realarray.

In its full generality, our target language allows types to be analyzed not just by case analysis, but also via primitive recursion. This allows more sophisticated transformations to be coded within the language, yet type checking for the target language remains decidable. An example of a more sophisticated translation made possible by primitive recursion is one where arrays of tuples are represented as tuples of arrays. For example, an array of $\mathsf{bool} \times \mathsf{real}$ is represented as a pair of a boolarray and a realarray. This representation allows the boolean components of the array to be packed and allows the real components to be naturally aligned. The subscript operation for this representation is defined using a recursive typecase construct called typerec in the following manner:

$$\begin{aligned}
&\mathsf{typerec\ sub}\,[\mathsf{bool}] = \mathsf{boolsub}\\
&\quad \mid \mathsf{sub}\,[\mathsf{real}] = \mathsf{realsub}\\
&\quad \mid \mathsf{sub}\,[\tau_1 \times \tau_2] =\\
&\qquad \lambda\langle\langle x,y\rangle,i\rangle.\langle \mathsf{sub}[\tau_1]\,\langle x,i\rangle, \mathsf{sub}[\tau_2]\,\langle y,i\rangle\rangle\\
&\quad \mid \mathsf{sub}\,[\tau] = \mathsf{boxedsub}[\tau]
\end{aligned}$$

If sub is given a product type, $\tau_1 \times \tau_2$, it returns a function that takes a pair of arrays ($\langle x, y\rangle$) and an index (i) and returns the pair of values from both arrays at that index, recursively calling the sub operation at the types τ_1 and τ_2.

The type of this sub operation is:

$$\forall\alpha.\mathsf{RecArray}[\alpha] \times \mathsf{int} \to \alpha$$

where the recursive, specialized array constructor RecArray is defined using a type-level "Typerec":

$$\begin{aligned}
&\mathsf{Typerec\ RecArray}\,[\mathsf{bool}] = \mathsf{boolarray}\\
&\quad \mid \mathsf{RecArray}\,[\mathsf{real}] = \mathsf{realarray}\\
&\quad \mid \mathsf{RecArray}\,[\tau_1 \times \tau_2] =\\
&\qquad \mathsf{RecArray}[\tau_1] \times \mathsf{RecArray}[\tau_2]\\
&\quad \mid \mathsf{RecArray}[\tau] = (\tau)\mathsf{boxedarray}
\end{aligned}$$

Again, the definition of the constructor parallels the definition of the sub operation. If the parameter is instantiated to bool, then the resulting type is boolarray. If the parameter is instantiated with $\tau_1 \times \tau_2$, then the resulting type is the product of $\mathsf{RecArray}[\tau_1]$ and $\mathsf{RecArray}[\tau_2]$.

Run-time type analysis can be used to provide other useful language mechanisms besides efficient representations. In particular, *ad hoc* polymorphic operators, such as the equality operator of SML, or an overloaded operator exported from a Haskell *type class*, can be directly implemented in our target language without the need to tag values. Furthermore, the static constraints of SML's equality types and Haskell's type classes may be coded using our Typerec construct. Our target language is also able to express "marshalling" of data structures and a form of type dynamic.

In Section 2 we describe the type-analysis approach to compilation as a type-based translation from a source language, Mini-ML, to our target language, λ_i^{ML}. The key properties of λ_i^{ML} are stated, and a few illustrative examples involving typerec and Typerec are given. In Section 3 we show how many interesting and useful language constructs can be coded using typerec, including flattened representations, marshalling, type classes, and type dynamic. In Section 4 we discuss related work, and in Section 5 we summarize and suggest directions for future research.

2 Type-Directed Compilation

In order to take full advantage of type information during compilation, we consider translations of typing derivations from the implicitly-typed ML core language to an explicitly-typed target language, following the interpretation of polymorphism suggested by Harper and Mitchell [20]. The source language is based on Mini-ML [11], which captures many of the essential features of the ML core language. The target language, λ_i^{ML}, is an extension of λ^{ML}, also known as XML [21], a predicative variant of Girard's F_ω [16, 17, 42], enriched with primitives for intensional type analysis.

2.1 Source Language: Mini-ML

The source language for our translations is a variant of Mini-ML [11]. The syntax of Mini-ML is defined by the following grammar:

$$\begin{array}{llll}
(\textit{monotypes}) & \tau & ::= & t \mid \mathsf{int} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2\\
(\textit{polytypes}) & \sigma & ::= & \tau \mid \forall t.\sigma\\
(\textit{terms}) & e & ::= & x \mid \bar{n} \mid \langle e_1, e_2\rangle \mid \pi_1\, e \mid \pi_2\, e \mid\\
& & & \lambda x.\, e \mid e_1\, e_2 \mid \mathsf{let}\, x = v \,\mathsf{in}\, e\\
(\textit{values}) & v & ::= & x \mid \bar{n} \mid \langle v_1, v_2\rangle \mid \lambda x.\, e
\end{array}$$

Monotypes (τ) are either type variables (t), int, arrow types, or binary product types. *Polytypes* (σ) (also known as *type schemes*) are either monotypes or prenex quantified types. We write $\forall t_1, \ldots, t_n.\tau$ to represent the polytype $\forall t_1.\cdots.\forall t_n.\tau$. The terms of Mini-ML ($e$) consist of identifiers, numerals ($\bar{n}$), pairs, first and second projections, abstractions, applications, and let expressions. Values (v) are a subset of the terms and include identifiers, integer values, pairs of values, and abstractions.

The static semantics for Mini-ML is given in Figure 1 as a series of inference rules. The rules allow us to derive a judgement of the form $\Delta;\Gamma \triangleright e : \tau$ where Δ is a set of free type variables and Γ is a type assignment mapping identifiers to polytypes. We write $[\tau/t]\tau'$ to denote the substitution of the type τ for the type variable t in the type expression τ'. We use $\Delta \uplus \Delta'$ to denote the union of two disjoint sets of type variables, Δ and Δ', and $\Gamma \uplus \{x : \sigma\}$ to denote the type assignment that extends Γ so that x is assigned the polytype σ, assuming x does not occur in the domain of Γ.

Let-bound expressions are restricted to values so that our translation, which makes type abstraction explicit, is correct (see below).

2.2 Target Language: λ_i^{ML}

The target language of our translations, λ_i^{ML}, is based on λ^{ML} [20], a predicative variant of Girard's F_ω [16, 17, 42]. The essential departure from the impredicative systems of Girard and Reynolds is that the quantifier $\forall t.\sigma$ ranges only over "small" types, or "monotypes", which do not include the quantified types. This calculus is sufficient for the interpretation of ML-style polymorphism (see Harper and Mitchell [20] for further discussion of this point.) The language λ_i^{ML} extends λ^{ML} with *intensional* (or *structural* [19]) polymorphism, that allows non-parametric functions to be defined by intensional analysis of types.

The four syntactic classes for λ_i^{ML}, kinds (k), constructors (μ), types (σ), and terms (e), are given below:

$$\begin{array}{llll}
(\textit{kinds}) & \kappa & ::= & \Omega \mid \kappa_1 \to \kappa_2\\
(\textit{con's}) & \mu & ::= & t \mid \mathsf{Int} \mid \to(\mu_1,\mu_2) \mid \times(\mu_1,\mu_2) \mid\\
& & & \lambda t{::}\kappa.\mu \mid \mu_1[\mu_2] \mid\\
& & & \mathsf{Typerec}\ \mu\ \mathsf{of}\ (\mu_{\mathsf{i}}|\mu_{\to}|\mu_{\times})\\
(\textit{types}) & \sigma & ::= & T(\mu) \mid \mathsf{int} \mid \sigma_1 \to \sigma_2 \mid \sigma_1 \times \sigma_2 \mid\\
& & & \forall t{::}\kappa.\sigma\\
(\textit{terms}) & e & ::= & x \mid \bar{n} \mid \lambda x{:}\sigma.\, e \mid @^{\sigma}\, e_1\, e_2 \mid\\
& & & \langle e_1, e_2\rangle^{\sigma_1,\sigma_2} \mid \pi_1^{\sigma_1,\sigma_2}\, e \mid \pi_2^{\sigma_1,\sigma_2}\, e \mid\\
& & & \Lambda t{::}\kappa.\, e \mid e[\mu] \mid\\
& & & \mathsf{typerec}\ \mu\ \mathsf{of}\ [t.\sigma](e_{\mathsf{i}}|e_{\to}|e_{\times})
\end{array}$$

Kinds classify constructors, and types classify terms. Constructors of kind Ω name "small types" or "monotypes". The monotypes are generated from Int and variables by the

$$(var)\quad \frac{FTV([\tau_n/t_n](\cdots([\tau_1/t_1]\tau)\cdots)) \subseteq \Delta}{\Delta;\Gamma \uplus \{x:\forall t_1,\ldots,t_n.\tau\} \triangleright x : [\tau_n/t_n](\cdots([\tau_1/t_1]\tau)\cdots)} \qquad (int)\quad \Delta;\Gamma \triangleright \bar{n} : \mathsf{int}$$

$$(pair)\quad \frac{\Delta;\Gamma \triangleright e_1 : \tau_1 \qquad \Delta;\Gamma \triangleright e_2 : \tau_2}{\Delta;\Gamma \triangleright \langle e_1, e_2\rangle : \tau_1 \times \tau_2} \qquad (\pi)\quad \frac{\Delta;\Gamma \triangleright e : \tau_1 \times \tau_2}{\Delta;\Gamma \triangleright \pi_i\, e : \tau_i}\quad (i=1,2)$$

$$(abs)\quad \frac{\Delta;\Gamma \uplus \{x:\tau_1\} \triangleright e : \tau_2}{\Delta;\Gamma \triangleright \lambda x.e : \tau_1 \to \tau_2} \qquad (app)\quad \frac{\Delta;\Gamma \triangleright e_1 : \tau' \to \tau \qquad \Delta;\Gamma \triangleright e_2 : \tau'}{\Delta;\Gamma \triangleright e_1\, e_2 : \tau}$$

$$(let)\quad \frac{\begin{array}{c}\Delta \uplus \{t_1,\ldots,t_n\};\Gamma \triangleright v : \tau' \\ \Delta;\Gamma \uplus \{x:\forall t_1,\ldots,t_n.\tau'\} \triangleright e : \tau\end{array}}{\Delta;\Gamma \triangleright \mathsf{let}\, x = v \,\mathsf{in}\, e : \tau}$$

Figure 1: Mini-ML Typing Rules

constructors $\to$ and $\times$. The application and abstraction constructors correspond to the function kind $\kappa_1 \to \kappa_2$. Types in λ_i^{ML} include the monotypes, and are closed under products, function spaces, and polymorphic quantification. We distinguish constructors from types, writing $T(\mu)$ for the type corresponding to the constructor μ. The terms are an explicitly-typed λ-calculus with explicit constructor abstraction and application forms.

The official syntax of terms shows that the primitive operations of the language are provided with type information that may be used at run-time. For example, the pairing operation is $\langle e_1, e_2\rangle^{\sigma_1,\sigma_2}$, where $e_i : \sigma_i$, reflecting the fact that there is (potentially) a pairing operation at each pair of types. In a typical implementation, the pairing operation is implemented by computing the size of the components from the types, allocating a suitable chunk of memory, and copying the parameters into that space. However, there is no need to tag the resulting value with type information because the projection operations ($\pi_i^{\sigma_1,\sigma_2}\, e$) are correspondingly indexed by the types of the components so that the appropriate chunk of memory can be extracted from the tuple. Similarly, the application primitive ($@^\sigma\, e_1\, e_2$) is indexed by the domain type of the function[1] and is used to determine the calling sequence for the function. Of course, primitive operations can ignore the type if a universal representation is used. Consequently, the implementor can decide whether to use a natural or universal representation. We use a simplified term syntax without the types when the information is apparent from the context. However, it is important to bear in mind that the type information is present in the fully explicit form of the calculus.

The Typerec and typerec forms provide the ability to define constructors and terms by structural induction on monotypes. These forms may be thought of as eliminatory forms for the kind Ω at the constructor and term level. (The introductory forms are the constructors of kind Ω; there are no introductory forms at the term level in order to preserve the phase distinction [8, 21].) At the term level typerec may be thought of as a generalization of typecase that provides for the definition of a term by induction on the structure of a monotype. At the constructor level Typerec provides a similar ability to define a constructor by induction on the structure of a monotype.

The static semantics of λ_i^{ML} consists of a collection of rules for deriving judgements of the following forms, where Δ is a kind assignment, mapping type variables (t) to kinds, and Γ is a type assignment, mapping term variables to types.

$\Delta \triangleright \mu :: \kappa$	*μ is a constructor of kind κ*
$\Delta \triangleright \mu_1 \equiv \mu_2 :: \kappa$	*μ_1 and μ_2 are equivalent constructors*
$\Delta \triangleright \sigma$	*σ is a valid type*
$\Delta \triangleright \sigma_1 \equiv \sigma_2$	*σ_1 and σ_2 are equivalent types*
$\Delta;\Gamma \triangleright e : \sigma$	*e is a term of type σ*

The formation rules for constructors are largely standard, with the exception of the Typerec form:

$$\frac{\begin{array}{c}\Delta \triangleright \mu :: \Omega \qquad \Delta \triangleright \mu_i :: \kappa \\ \Delta \triangleright \mu_\to :: \Omega \to \Omega \to \kappa \to \kappa \to \kappa \\ \Delta \triangleright \mu_\times :: \Omega \to \Omega \to \kappa \to \kappa \to \kappa\end{array}}{\Delta \triangleright \mathsf{Typerec}\ \mu\ \mathsf{of}\ (\mu_i|\mu_\to|\mu_\times) :: \kappa}$$

The whole constructor has kind κ if the constructor to be analyzed, μ, is of kind Ω (*i.e.*, a monotype), μ_i is of kind κ, and $\mu_\to$ and $\mu_\times$ are each of kind $\Omega \to \Omega \to \kappa \to \kappa \to \kappa$.

The constructor equivalence rules (see Figure 2) axiomatize *definitional equality* [47, 31] of constructors to consist of β-conversion together with recursion equations governing the Typerec form. Conceptually, Typerec selects μ_i, $\mu_\times$, or $\mu_\to$ according to the head-constructor of the normal form of μ and passes it the components of μ and the "unrolling" of the Typerec on the components. The level of constructors and kinds is a variation of Gödel's **T** [18]. Every constructor, μ, has a unique normal form, $NF(\mu)$, with respect to the obvious notion of reduction derived from the equivalence rules of Figure 2 [47]. This reduction relation is confluent, from which it follows that constructor equivalence is decidable [47].

The type formation, type equivalence, and term formation rules for λ_i^{ML} are omitted due to lack of space, but can be found in a previous report [22]. The rules of type equivalence define the interpretation $T(\mu)$ of the constructor μ as a type. For example, $T(\mathsf{Int}) \equiv \mathsf{int}$ and $T(\to(\mu_1,\mu_2)) \equiv T(\mu_1) \to T(\mu_2)$. Thus, T takes us from a constructor which *names* a type to the actual type. The term formation rules are standard with the exception of the typerec form, which

[1] In general, application could also depend upon the range type, but our presentation is simplified greatly by restricting the dependency to the domain type.

$$\frac{\Delta \uplus \{t :: \kappa'\} \triangleright \mu_1 :: \kappa \qquad \Delta \triangleright \mu_2 :: \kappa'}{\Delta \triangleright (\lambda t :: \kappa'.\mu_1)[\mu_2] \equiv [\mu_2/t]\mu_1 :: \kappa} \qquad \frac{\begin{array}{c}\Delta \triangleright \mu_{\mathsf{i}} :: \kappa \\ \Delta \triangleright \mu_{\rightarrow} :: \Omega \to \Omega \to \kappa \to \kappa \to \kappa \\ \Delta \triangleright \mu_{\times} :: \Omega \to \Omega \to \kappa \to \kappa \to \kappa\end{array}}{\Delta \triangleright \mathsf{Typerec\ Int\ of}\ (\mu_{\mathsf{i}}|\mu_{\rightarrow}|\mu_{\times}) \equiv \mu_{\mathsf{i}} :: \kappa}$$

$$\frac{\begin{array}{c}\Delta \triangleright \mu_1 :: \Omega \qquad \Delta \triangleright \mu_2 :: \Omega \qquad \Delta \triangleright \mu_{\mathsf{i}} :: \kappa \\ \Delta \triangleright \mu_{\rightarrow} :: \Omega \to \Omega \to \kappa \to \kappa \to \kappa \qquad \Delta \triangleright \mu_{\times} :: \Omega \to \Omega \to \kappa \to \kappa \to \kappa\end{array}}{\left\{\begin{array}{c}\Delta \triangleright \mathsf{Typerec}\ (\rightarrow(\mu_1,\mu_2))\ \mathsf{of}\ (\mu_{\mathsf{i}}|\mu_{\rightarrow}|\mu_{\times}) \equiv \mu_{\rightarrow}\ \mu_1\ \mu_2\ (\mathsf{Typerec}\ \mu_1\ \mathsf{of}\ (\mu_{\mathsf{i}}|\mu_{\rightarrow}|\mu_{\times}))\ (\mathsf{Typerec}\ \mu_2\ \mathsf{of}\ (\mu_{\mathsf{i}}|\mu_{\rightarrow}|\mu_{\times})) :: \kappa \\ \Delta \triangleright \mathsf{Typerec}\ (\times(\mu_1,\mu_2))\ \mathsf{of}\ (\mu_{\mathsf{i}}|\mu_{\rightarrow}|\mu_{\times}) \equiv \mu_{\times}\ \mu_1\ \mu_2\ (\mathsf{Typerec}\ \mu_1\ \mathsf{of}\ (\mu_{\mathsf{i}}|\mu_{\rightarrow}|\mu_{\times}))\ (\mathsf{Typerec}\ \mu_2\ \mathsf{of}\ (\mu_{\mathsf{i}}|\mu_{\rightarrow}|\mu_{\times})) :: \kappa\end{array}\right\}}$$

Figure 2: Constructor Equivalence

is governed by the following rule:

$$\frac{\begin{array}{c}\Delta \triangleright \mu :: \Omega \qquad \Delta \uplus \{t{::}\Omega\} \triangleright \sigma \qquad \Delta;\Gamma \triangleright e_{\mathsf{i}} : [\mathsf{Int}/t]\sigma \\ \Delta;\Gamma \triangleright e_{\rightarrow} : \forall t_1, t_2{::}\Omega.[t_1/t]\sigma \to [t_2/t]\sigma \to [\rightarrow(t_1,t_2)/t]\sigma \\ \Delta;\Gamma \triangleright e_{\times} : \forall t_1, t_2{::}\Omega.[t_1/t]\sigma \to [t_2/t]\sigma \to [\times(t_1,t_2)/t]\sigma\end{array}}{\Delta;\Gamma \triangleright \mathsf{typerec}\ \mu\ \mathsf{of}\ [t.\sigma](e_{\mathsf{i}}|e_{\rightarrow}|e_{\times}) : [\mu/t]\sigma}$$

The argument constructor μ must be of kind Ω, and the result type of the typerec expression is determined as a function of the argument constructor, namely the substitution of μ for t in the type expression σ. The "$[t.\sigma]$" label provides the type information needed to check the construct without infereence. Typically the constructor variable t occurs in σ as the argument of a Typerec expression so that $[\mu/t]\sigma$ is determined by a recursive analysis of μ. Similar to normalization of a Typerec constructor, the evaluation of a typerec expression selects e_{i}, $e_{\times}$, or $e_{\rightarrow}$ according to the head constructor of the normal form of μ and passes it the components of μ and the "unrolling" of the typerec on the components.

Type checking for λ_i^{ML} reduces to equivalence checking for types and constructors. In view of the decidability of constructor equivalence, we have the following important result:

Proposition 2.1 *It is decidable whether or not $\Delta;\Gamma \triangleright e : \sigma$ is derivable in λ_i^{ML}.*

To fix the interpretation of typerec, we specify a call-by-value, natural semantics for λ_i^{ML} as a relation of the form $e \hookrightarrow v$ where v is a closed (with respect to both type and value variables), syntactic value. Values are derived from the following grammar:

$$v \quad ::= \quad \bar{n} \mid \lambda x{:}\sigma.\, e \mid \langle v_2, v_2\rangle^{\sigma_1,\sigma_2} \mid \Lambda t{::}\kappa.\, e$$

Type abstractions are values, reflecting the fact that evaluation does not proceed under Λ.

Figure 3 defines the evaluation relation with a series of axioms and inference rules. The semantics uses an auxiliary judgment, $\mu \hookrightarrow \mu'$, (not formally defined here) that determines the normal forms of constructors. During evaluation, we only need to determine normal forms of closed constructors of kind Ω. This amounts to evaluating constructors of the form Typerec(...) and $(\mu_1[\mu_2])$ by orienting the equivalences of Figure 2 to the right and adding the appropriate congruences.

The rest of the semantics is standard except for the evaluation of a typerec expression which proceeds as follows: First, the normal form of the constructor argument is determined. Once the normal form is determined, the appropriate subexpression is selected and applied to any argument constructors. The resulting function is in turn applied to the "unrolling" of the typerec at each of the argument constructors. Some simple examples using typerec may be found at the end of this subsection.

The semantics uses meta-level substitution of closed values for variables and closed constructors for type variables. In a lower-level semantics where substitution is made explicit, an environment would be needed not only for value variables, but also for type variables. Tolmach [51] describes many of the issues involved in implementing such a language.

Proposition 2.2 (Type Preservation) *If $\emptyset;\emptyset \triangleright e : \sigma$ and $e \hookrightarrow v$, then $\emptyset;\emptyset \triangleright v : \sigma$.*

By inspection of the value typing rules, only appropriate values occupy appropriate types and thus evaluation will not "go wrong". In particular, it is possible to show that when evaluating well-typed programs, we only use the *proj* evaluation rule when $\sigma_1' \equiv \sigma_1$ and $\sigma_2' \equiv \sigma_2$ and we only use the *app* rule when $\sigma' \equiv \sigma$. Furthermore, programs written in pure λ_i^{ML} (*i.e.*, without general recursion operators or recursive types) always terminate.

Proposition 2.3 (Termination) *If e is an expression such that $\emptyset;\emptyset \triangleright e : \sigma$, then there exists a value v such that $e \hookrightarrow v$.*

A few simple examples will help to clarify the use of typerec. The function sizeof of type $\forall t{::}\Omega.\mathsf{int}$ that computes the "size" of values of a type can be defined as follows.

$$\mathsf{sizeof} = \Lambda t{::}\Omega.\mathsf{typerec}\ t\ \mathsf{of}\ [t'.\mathsf{int}](e_{\mathsf{i}}|e_{\rightarrow}|e_{\times})$$

where

$$\begin{array}{rcl} e_{\mathsf{i}} & = & \bar{1} \\ e_{\rightarrow} & = & \Lambda t_1{::}\Omega.\Lambda t_2{::}\Omega.\lambda x_1{:}\mathsf{int}.\lambda x_2{:}\mathsf{int}.\bar{1} \\ e_{\times} & = & \Lambda t_1{::}\Omega.\Lambda t_2{::}\Omega.\lambda x_1{:}\mathsf{int}.\lambda x_2{:}\mathsf{int}.x_1 + x_2 \end{array}$$

(Here we assume that arrow types are boxed and thus have size one.) It is easy to check that sizeof has the type $\forall t{::}\Omega.\mathsf{int}$. Note that in a parametric setting this type contains only constant functions.

As another example, Girard's formulation of System F [16] includes a distinguished constant $\mathbf{0}_\tau$ of type τ for each type τ (including variable types). We may define an analogue of these constants using typerec as follows:

$$\mathsf{zero} = \Lambda t{::}\Omega.\mathsf{typerec}\ t\ \mathsf{of}\ [t'.T(t')](e_{\mathsf{i}}|e_{\rightarrow}|e_{\times})$$

$$(val)\quad v \hookrightarrow v \qquad (pair)\quad \frac{e_1 \hookrightarrow v_1 \qquad e_2 \hookrightarrow v_2}{\langle e_1, e_2\rangle^{\sigma_1,\sigma_2} \hookrightarrow \langle v_1, v_2\rangle^{\sigma_1,\sigma_2}} \qquad (proj)\quad \frac{e \hookrightarrow \langle v_1, v_2\rangle^{\sigma'_1,\sigma'_2}}{\pi_i^{\sigma_1,\sigma_2}\, e \hookrightarrow v_i} \quad (i = 1, 2)$$

$$(app)\quad \frac{e_1 \hookrightarrow \lambda x{:}\sigma'.\, e \qquad e_2 \hookrightarrow v' \qquad [v'/x]e \hookrightarrow v}{@^{\sigma}\, e_1\, e_2 \hookrightarrow v} \qquad (tapp)\quad \frac{e \hookrightarrow \Lambda t{::}\kappa.\, e' \qquad [\mu/t]e' \hookrightarrow v}{e[\mu] \hookrightarrow v} \qquad (trec\text{-}int)\quad \frac{\mu \hookrightarrow \mathsf{Int} \qquad e_{\mathsf{i}} \hookrightarrow v}{\mathsf{typerec}\ \mu\ \mathsf{of}\ [t.\sigma](e_{\mathsf{i}}|e_{\to}|e_{\times}) \hookrightarrow v}$$

$$(trec\text{-}fn)\quad \frac{\begin{array}{c}\mu \hookrightarrow\ \to(\mu_1, \mu_2)\\ \mathsf{typerec}\ \mu_1\ \mathsf{of}\ [t.\sigma](e_{\mathsf{i}}|e_{\to}|e_{\times}) \hookrightarrow v_1\\ \mathsf{typerec}\ \mu_2\ \mathsf{of}\ [t.\sigma](e_{\mathsf{i}}|e_{\to}|e_{\times}) \hookrightarrow v_2\\ @^{[\mu_2/t]\sigma}(@^{[\mu_1/t]\sigma}(e_{\to}[\mu_1][\mu_2])\ v_1)\ v_2 \hookrightarrow v\end{array}}{\mathsf{typerec}\ \mu\ \mathsf{of}\ [t.\sigma](e_{\mathsf{i}}|e_{\to}|e_{\times}) \hookrightarrow v} \qquad (trec\text{-}pair)\quad \frac{\begin{array}{c}\mu \hookrightarrow\ \times(\mu_1, \mu_2)\\ \mathsf{typerec}\ \mu_1\ \mathsf{of}\ [t.\sigma](e_{\mathsf{i}}|e_{\to}|e_{\times}) \hookrightarrow v_1\\ \mathsf{typerec}\ \mu_2\ \mathsf{of}\ [t.\sigma](e_{\mathsf{i}}|e_{\to}|e_{\times}) \hookrightarrow v_2\\ @^{[\mu_2/t]\sigma}(@^{[\mu_1/t]\sigma}(e_{\times}[\mu_1][\mu_2])\ v_1)\ v_2 \hookrightarrow v\end{array}}{\mathsf{typerec}\ \mu\ \mathsf{of}\ [t.\sigma](e_{\mathsf{i}}|e_{\to}|e_{\times}) \hookrightarrow v}$$

Figure 3: Operational Semantics for λ_i^{ML}

where

$$\begin{array}{rcl} e_{\mathsf{i}} & = & \bar{0} \\ e_{\to} & = & \Lambda t_1{::}\Omega.\Lambda t_2{::}\Omega.\lambda z_1{:}T(t_1).\lambda z_2{:}T(t_2).\lambda x{:}T(t_1).z_2 \\ e_{\times} & = & \Lambda t_1{::}\Omega.\Lambda t_2{::}\Omega.\lambda z_1{:}T(t_1).\lambda z_2{:}T(t_2).\langle z_1, z_2\rangle \end{array}$$

It is easy to check that zero has type $\forall t{::}\Omega.T(t)$, the "empty" type in System F and related systems. The presence of typerec violates parametricity to achieve a more flexible programming language.

To simplify the presentation we usually define terms such as zero and sizeof using recursion equations, rather than as a typerec expression. The definitions of zero and sizeof are given in this form as follows:

$$\begin{array}{rcl} \mathsf{sizeof}[\mathsf{Int}] & = & \bar{1} \\ \mathsf{sizeof}[\times(\mu_1, \mu_2)] & = & \mathsf{sizeof}[\mu_1] + \mathsf{sizeof}[\mu_2] \\ \mathsf{sizeof}[\to(\mu_1, \mu_2)] & = & \bar{1} \\ \\ \mathsf{zero}[\mathsf{Int}] & = & \bar{0} \\ \mathsf{zero}[\times(\mu_1, \mu_2)] & = & \langle \mathsf{zero}[\mu_1], \mathsf{zero}[\mu_2]\rangle \\ \mathsf{zero}[\to(\mu_1, \mu_2)] & = & \lambda x{:}T(\mu_1).\mathsf{zero}[\mu_2] \end{array}$$

Whenever a definition is presented in this form we tacitly assert that it can be formalized using typerec.

2.3 Translating Mini-ML into λ_i^{ML}

A compiler from Mini-ML to λ_i^{ML} is specified by a relation $\Delta; \Gamma \triangleright e_s : \tau \Rightarrow e_t$ that carries the meaning that $\Delta; \Gamma \triangleright e_s : \tau$ is a derivable typing in Mini-ML and that the translation of the source term e_s determined by that typing derivation is the λ_i^{ML} expression e_t. Since the translation depends upon the typing derivation, it is possible to have many different translations of a given expression. However, all of the translation schemes we consider are *coherent* in the sense that any two typing derivations produce observationally equivalent translations [7, 26, 20].[2]

Here, we give a straightforward compiler whose purpose is to make types explicit so that the primitive operations such as pairing and projection can potentially analyze their types at run-time. This simple translation does not utilize typerec or Typerec, but subsequent translations take advantage of these constructs.

We begin by defining a translation from Mini-ML types to λ_i^{ML} constructors, written $|\tau|$:

$$\begin{array}{rcl} |t| & = & t \\ |\mathsf{int}| & = & \mathsf{Int} \\ |\tau_1 \to \tau_2| & = & \to(|\tau_1|, |\tau_2|) \\ |\tau_1 \times \tau_2| & = & \times(|\tau_1|, |\tau_2|) \end{array}$$

The translation is extended to map Mini-ML type schemes to λ_i^{ML} types as follows:

$$\begin{array}{rcl} |\tau|_s & = & T(|\tau|) \\ |\forall t.\sigma|_s & = & \forall t{::}\Omega.|\sigma|_s \end{array}$$

Finally, we write $|\Delta|$ for the kind assignment mapping t to the kind Ω for each $t \in \Delta$, and $|\Gamma|$ for the type assignment mapping x to $|\Gamma(x)|_s$ for each $x \in \mathrm{dom}(\Gamma)$.

Proposition 2.4 *The type translation commutes with substitution:*

$$|[\tau_n/t_n](\cdots([\tau_1/t_1]\tau)\cdots)| = [|\tau_n|/t_n](\cdots([|\tau_1|/t_n]|\tau|)\cdots)$$

The term translation is given in Figure 4 as a series of inference rules that parallel the typing rules for Mini-ML. The *var* rule turns Mini-ML implicit instantiation of type variables into λ_i^{ML} explicit type application. Operationally, this corresponds to passing the types to the polymorphic value at run-time. The *let* rule makes the implicit type abstraction of the bound expression explicit. The translation of λ-abstraction, application, pairing, and projection is straightforward except that these primitive operations are labelled with their types.

The translation may be characterized by the following type preservation property.

Theorem 2.5 *If* $\Delta; \Gamma \triangleright e : \tau \Rightarrow e'$*, then* $|\Delta|; |\Gamma| \triangleright e' : |\tau|$.

Given a standard, call-by-value operational semantics for Mini-ML with the value restriction, and given the stratification between monotypes and polytypes in both Mini-ML and λ_i^{ML}, it is possible to modify a standard binary logical relations-style argument for the simply-typed lambda calculus [48, 15, 40, 45, 46] to show the correctness of the

[2] We omit explicit consideration of the coherence of our translations here.

$$(var)\quad \frac{FTV([\tau_n/t_n](\cdots([\tau_1/t_1]\tau)\cdots)) \subseteq \Delta}{\Delta;\Gamma \uplus \{x:\forall t_1,\ldots,t_n.\tau\} \rhd x : [\tau_n/t_n](\cdots([\tau_1/t_1]\tau)\cdots) \Rightarrow x[|\tau_1|]\cdots[|\tau_n|]} \qquad (int)\quad \Delta;\Gamma \rhd \bar{n} : \mathsf{int} \Rightarrow \bar{n}$$

$$(pair)\quad \frac{\Delta;\Gamma \rhd e_1 : \tau_1 \Rightarrow e_1' \qquad \Delta;\Gamma \rhd e_2 : \tau_2 \Rightarrow e_2'}{\Delta;\Gamma \rhd \langle e_1, e_2\rangle : \tau_1 \times \tau_2 \Rightarrow \langle e_1', e_2'\rangle^{|\tau_1|_s,|\tau_2|_s}} \qquad (\pi)\quad \frac{\Delta;\Gamma \rhd e : \tau_1 \times \tau_2 \Rightarrow e'}{\Delta;\Gamma \rhd \pi_i\, e : \tau_i \Rightarrow \pi_i^{|\tau_1|_s,|\tau_2|_s}\, e'} \quad (i=1,2)$$

$$(abs)\quad \frac{\Delta;\Gamma \uplus \{x:\tau_1\} \rhd e : \tau_2 \Rightarrow e'}{\Delta;\Gamma \rhd \lambda x.\, e : \tau_1 \to \tau_2 \Rightarrow \lambda x : |\tau_1|_s.\, e'} \qquad (app)\quad \frac{\Delta;\Gamma \rhd e_1 : \tau' \to \tau \Rightarrow e_1' \qquad \Delta;\Gamma \rhd e_2 : \tau' \Rightarrow e_2'}{\Delta;\Gamma \rhd e_1\, e_2 : \tau \Rightarrow @^{|\tau'|_s}\, e_1'\, e_2'}$$

$$(let)\quad \frac{\begin{array}{c}\Delta \uplus \{t_1,\ldots,t_n\};\Gamma \rhd v : \tau' \Rightarrow e_1' \\ \Delta;\Gamma \uplus \{x:\forall t_1,\ldots,t_n.\tau'\} \rhd e : \tau \Rightarrow e_2'\end{array}}{\begin{array}{c}\Delta;\Gamma \rhd \mathsf{let}\, x = v\, \mathsf{in}\, e : \tau \Rightarrow \\ @^{|\forall t_1,\ldots,t_n.\tau'|_s}\,(\lambda x : \forall t_1{::}\Omega,\ldots,t_n{::}\Omega.|\tau'|_s.\, e_2')\,(\Lambda t_1{::}\Omega,\ldots,t_n{::}\Omega.\, e_1')\end{array}}$$

Figure 4: Translation from Mini-ML to λ_i^{ML}

translation. That is, we may show that at base type, if a Mini-ML program computes a value, then its λ_i^{ML} translation computes the same value.

In the presence of computational effects such as non-termination, if we did not restrict the bound expression in a let to be a value, then the translation would be incorrect since evaluation in λ_i^{ML} does not proceed under Λ-abstractions. In other words, the expression might diverge (or print "hello") while its translation would not.

3 Applications of Type Analysis

In this section, we show how to implement a variety of useful and interesting constructs by extending the simple translation from Mini-ML to λ_i^{ML} to take advantage of typerec and Typerec. We have already shown how simple operations like sizeof and zero can be defined in λ_i^{ML}. These operations can be exported directly to Mini-ML as constants of the appropriate type. In the following subsections, we define new operations that may be exported to Mini-ML and modify the standard translation to change the representation of various types.

3.1 Flattening

We consider the "flat" representation of Mini-ML tuples in which nested tuples are represented by a sequence of "atomic" values (for the present purposes, any non-tuple is regarded as "atomic"). To simplify the development we give a translation in which nested binary tuples are represented in right-associated form, so that, for example, the Mini-ML type $(\mathsf{int} \times \mathsf{int}) \times \mathsf{int}$ will be compiled to the λ_i^{ML} type $\mathsf{int} \times (\mathsf{int} \times \mathsf{int})$. The compilation makes use of intensional type analysis at both the term and constructor levels.

We begin by modifying the type translation on Mini-ML tuples:

$$|\tau_1 \times \tau_2| \quad = \quad \mathsf{Prod}[|\tau_1|][|\tau_2|]$$

Here Prod is a constructor of kind $\Omega \to \Omega \to \Omega$ defined below as:

$$\begin{array}{rcl}\mathsf{Prod}[\mathsf{Int}][\mu] &=& \times(\mathsf{Int},\mu)\\ \mathsf{Prod}[\to(\mu_a,\mu_b)][\mu] &=& \times(\to(\mu_a,\mu_b),\mu)\\ \mathsf{Prod}[\times(\mu_a,\mu_b)][\mu] &=& \times(\mu_a,\mathsf{Prod}[\mu_b][\mu])\end{array}$$

Informally, the constructor Prod computes the right-associated form of a product of two types. For example,

$$|(\mathsf{int} \times \mathsf{int}) \times \mathsf{int}| = \mathsf{Prod}[\mathsf{Prod}[\mathsf{Int}][\mathsf{Int}]][\mathsf{Int}]$$

and

$$|\mathsf{int} \times (\mathsf{int} \times \mathsf{int})| = \mathsf{Prod}[\mathsf{Int}][\mathsf{Prod}[\mathsf{Int}][\mathsf{Int}]]$$

and the equation

$$\rhd\ \mathsf{Prod}[\mathsf{Prod}[\mathsf{Int}][\mathsf{Int}]][\mathsf{Int}] \equiv \mathsf{Prod}[\mathsf{Int}][\mathsf{Prod}[\mathsf{Int}][\mathsf{Int}]] :: \Omega$$

is derivable in λ_i^{ML}.

The term translation is modified by changing the behavior of the *pair* and π rules:

$$\frac{\Delta;\Gamma \rhd e_1 : \tau_1 \Rightarrow e_1' \qquad \Delta;\Gamma \rhd e_2 : \tau_2 \Rightarrow e_2'}{\Delta;\Gamma \rhd \langle e_1, e_2\rangle : \tau_1 \times \tau_2 \Rightarrow \mathsf{mkpair}[|\tau_1|][|\tau_2|]\ e_1'\ e_2'}$$

$$\frac{\Delta;\Gamma \rhd e : \tau_1 \times \tau_2 \Rightarrow e'}{\Delta;\Gamma \rhd \pi_i\, e : \tau_i \Rightarrow \mathsf{proj}_i[|\tau_1|][|\tau_2|]\ e'} \quad (i=1,2)$$

The modified translation makes use of three auxiliary functions, mkpair, proj_1 and proj_2, with the following types:

$$\begin{array}{l}\mathsf{mkpair} : \forall t_1,t_2 :: \Omega.T(t_1) \to T(t_2) \to T(\mathsf{Prod}[t_1][t_2])\\ \mathsf{proj}_1 : \forall t_1,t_2 :: \Omega.T(\mathsf{Prod}[t_1][t_2]) \to T(t_1)\\ \mathsf{proj}_2 : \forall t_1,t_2 :: \Omega.T(\mathsf{Prod}[t_1][t_2]) \to T(t_2)\end{array}$$

The mkpair operation is defined as follows, using the "unofficial" syntax of the language:

$$\begin{array}{rl}\mathsf{mkpair}[\mathsf{Int}][\mu] = & \lambda x{:}T(\mathsf{Int}).\,\lambda y{:}T(\mu).\,\langle x,y\rangle\\ \mathsf{mkpair}[\to(\mu_a,\mu_b)][\mu] = & \lambda x{:}T(\to(\mu_a,\mu_b)).\\ & \lambda y{:}T(\mu).\,\langle x,y\rangle\\ \mathsf{mkpair}[\times(\mu_a,\mu_b)][\mu] = & \lambda x{:}T(\times(\mu_a,\mu_b)).\,\lambda y{:}T(\mu).\\ & \langle \pi_1\, x, \mathsf{mkpair}[\mu_b][\mu](\pi_2\, x)\ y\rangle\end{array}$$

The verification that mkpair has the required type proceeds by case analysis on the form of its first argument, relying on the defining equations for Prod. For example, we must check that $\mathsf{mkpair}[\mathsf{Int}][\mu]$ has type

$$T(\mathsf{Int}) \to T(\mu) \to T(\mathsf{Prod}[\mathsf{Int}][\mu])$$

which follows from the definition of $\mathsf{mkpair}[\mathsf{Int}][\mu]$ and the fact that

$$T(\mathsf{Prod}[\mathsf{Int}][\mu]) \equiv \mathsf{int} \times T(\mu).$$

Similarly, we must check that $\mathsf{mkpair}[\times(\mu_a, \mu_b)][\mu]$ has type

$$T(\times(\mu_a, \mu_b)) \to T(\mu) \to T(\mathsf{Prod}[\times(\mu_a, \mu_b)][\mu]$$

which follows from its definition, the derivability of the equation

$$T(\mathsf{Prod}[\times(\mu_a, \mu_b)][\mu]) \equiv T(\mu_a) \times T(\mathsf{Prod}[\mu_b][\mu]),$$

and, inductively, the fact that $\mathsf{mkpair}[\mu_b][\mu]$ has type $T(\mu_b) \to T(\mu) \to T(\mathsf{Prod}[\mu_b][\mu])$.

The operations proj_1 and proj_2 are defined as follows:

$$\begin{aligned}
\mathsf{proj}_1[\mathsf{Int}][\mu] &= \lambda x{:}T(\mathsf{Prod}[\mathsf{Int}][\mu]).\, \pi_1\, x \\
\mathsf{proj}_1[\to(\mu_a, \mu_b)][\mu] &= \lambda x{:}T(\mathsf{Prod}[\to(\mu_a, \mu_b)][\mu]).\, \pi_1\, x \\
\mathsf{proj}_1[\times(\mu_a, \mu_b)][\mu] &= \lambda x{:}T(\mathsf{Prod}[\times(\mu_a, \mu_b)][\mu]). \\
&\quad \langle \pi_1\, x, \mathsf{proj}_1[\mu_b][\mu](\pi_2\, x)\rangle
\end{aligned}$$

$$\begin{aligned}
\mathsf{proj}_2[\mathsf{Int}][\mu] &= \lambda x{:}T(\mathsf{Prod}[\mathsf{Int}][\mu]).\, \pi_2\, x \\
\mathsf{proj}_2[\to(\mu_a, \mu_b)][\mu] &= \lambda x{:}T(\mathsf{Prod}[\to(\mu_a, \mu_b)][\mu]).\, \pi_2\, x \\
\mathsf{proj}_2[\times(\mu_a, \mu_b)][\mu] &= \lambda x{:}T(\mathsf{Prod}[\times(\mu_a, \mu_b)][\mu]). \\
&\quad \mathsf{proj}_2[\mu_b][\mu](\pi_2\, x)
\end{aligned}$$

The verification that these constructors have the required type is similar to that of mkpair, keeping in mind the equations governing $T(-)$ and $\mathsf{Prod}[-][-]$.

One advantage of controlling data representation in this manner is that it becomes possible to support a type-safe form of casting that we call a *view*. Let us define two Mini-ML types τ_1 and τ_2 to be *similar*, $\tau_1 \approx \tau_2$, iff they have the same representation — *i.e.*, iff $|\tau_1|$ is definitionally equivalent to $|\tau_2|$ in λ_i^{ML}. If $\tau_1 \approx \tau_2$, then every value of type τ_1 is also a value of type τ_2, and vice-versa. For example, in the case of the right-associative representation of nested tuples above, we have that $\tau_1 \approx \tau_2$ iff τ_1 and τ_2 are equivalent modulo associativity of the product constructor, and a value of a (nested) product type is a value of every other association of that type.

In contrast to coercion implementations of type equivalence, such an approach to views is compatible with mutable types (*i.e.*, arrays and refs) in the sense that $\tau_1\ \mathsf{ref}$ is equivalent to $\tau_2\ \mathsf{ref}$ iff τ_1 is equivalent to τ_2. This means that we may freely intermingle updates with views of complex data structures, capturing some of the expressiveness of **C** casts without sacrificing type safety.

The right-associated representation does not capture all aspects of "flatness". In particular, access to components is not constant time, given a standard implementation of the pairing and projection operations. This may be overcome by extending λ_i^{ML} with n-tuples (tuples of variable arity), and modifying the interpretation of the product type appropriately. A rigorous formulation of the target language extended with n-tuples is tedious, but appears to be straightforward.

3.2 Marshalling

Ohori and Kato give an extension of ML with primitives for distributed computing in a heterogeneous environment [39]. Their extension has two essential features: one is a mechanism for generating globally unique names ("handles" or "capabilities") that are used as proxies for functions provided by servers. The other is a method for representing arbitrary values in a form suitable for transmission through a network. Integers are considered transmissible, as are pairs of transmissible values, but functions cannot be transmitted (due to the heterogeneous environment) and are thus represented by proxy using unique identifiers. These identifiers are associated with their functions by a name server that may be contacted through a primitive addressing scheme. In this section we sketch how a variant of Ohori and Kato's representation scheme can be implemented using intensional polymorphism.

To accommodate Ohori and Kato's primitives the λ_i^{ML} language is extended with a primitive constructor Id of kind $\Omega \to \Omega$ and a corresponding type constructor $\mathsf{id}(\sigma)$, linked by the equation $T(\mathsf{Id}[\mu]) \equiv \mathsf{id}(T(\mu))$. The $\mathsf{Typerec}$ and $\mathsf{typerec}$ primitives are extended in the obvious way to account for constructors of the form $\mathsf{Id}[\mu]$. For example, the following constructor equivalence is added:

$$\frac{\begin{array}{c}\Delta \triangleright \mu :: \Omega \qquad \Delta \triangleright \mu_{\mathsf{i}} :: \Omega \\ \Delta \triangleright \mu_{\to}, \mu_{\times} :: \Omega \to \Omega \to \kappa \to \kappa \to \kappa \\ \Delta \triangleright \mu_{\mathsf{id}} :: \Omega \to \kappa \to \kappa\end{array}}{\begin{array}{c}\mathsf{Typerec}\ \mathsf{Id}[\mu]\ \mathsf{of}\ (\mu_{\mathsf{i}}|\mu_{\to}|\mu_{\times}|\mu_{\mathsf{id}}) \ \equiv \\ \mu_{\mathsf{id}}\, \mu\, (\mathsf{Typerec}\ \mu\ \mathsf{of}\ (\mu_{\mathsf{i}}|\mu_{\to}|\mu_{\times}|\mu_{\mathsf{id}}))\end{array}}$$

The primitives newid and rpc are added with the following types:

$$\begin{aligned}
\mathsf{newid} &: \forall t_1, t_2{::}\Omega.(T(\mathsf{Tran}[t_1]) \to T(\mathsf{Tran}[t_2])) \to \\
&\qquad T(\mathsf{Tran}[\to(t_1, t_2)])
\end{aligned}$$

$$\begin{aligned}
\mathsf{rpc} &: \forall t_1, t_2{::}\Omega.(T(\mathsf{Tran}[\to(t_1, t_2)])) \to \\
&\qquad T(\mathsf{Tran}[t_1]) \to T(\mathsf{Tran}[t_2])
\end{aligned}$$

where Tran is a constructor coded using $\mathsf{Typerec}$ as follows:

$$\begin{aligned}
\mathsf{Tran}[\mathsf{Int}] &= \mathsf{Int} \\
\mathsf{Tran}[\to(\mu_1, \mu_2)] &= \mathsf{Id}[\to(\mathsf{Tran}[\mu_1], \mathsf{Tran}[\mu_2])] \\
\mathsf{Tran}[\times(\mu_1, \mu_2)] &= \times(\mathsf{Tran}[\mu_1], \mathsf{Tran}[\mu_2]) \\
\mathsf{Tran}[\mathsf{Id}[\mu]] &= \mathsf{Id}[\mu]
\end{aligned}$$

The constructor $\mathsf{Tran}[\mu]$ maps μ to a constructor where each arrow is wrapped by an Id constructor. Thus, values of type $T(\mathsf{Tran}[\mu])$ do not contain functions and are therefore *transmissible*. It is easy to check that Tran is a constructor of kind $\Omega \to \Omega$.

From an abstract perspective, newid maps a function on transmissible representations to a transmissible representation of the function and rpc is its (left) inverse. Operationally, newid takes a function between transmissible values, generates a new, globally unique identifier and tells the name server to associate that identifier with the function on the local machine. For example, the unique identifier might consist of the machine's name paired with the address of the function. The rpc operation takes a proxy identifier of a remote function, and a transmissible argument value. The name server is contacted to discover the remote machine where the value actually lives. The argument value is sent to this machine, the function associated with the identifier is applied to the argument, and the result of the function is transmitted back as the result of the operation.

The compilation of Ohori and Kato's distribution primitives into this extension of λ_i^{ML} relies critically on a "marshalling" operation M that converts a value to its transmissible representation and an "unmarshalling" operation U that converts a value from its transmissible representation. The types of these operations can be easily expressed in terms of Tran:

$$\begin{array}{lcl} \mathsf{M} & : & \forall t{::}\Omega.T(t) \to T(\mathsf{Tran}[t]) \\ \mathsf{U} & : & \forall t{::}\Omega.T(\mathsf{Tran}[t]) \to T(t) \end{array}$$

The operations themselves can be defined as follows using the unofficial syntax of typerec:[3]

$$\begin{array}{rl} \mathsf{M}[\mathsf{Int}] = & \lambda x{:}\mathsf{int}.x \\ \mathsf{M}[\to(\mu_1,\mu_2)] = & \lambda f{:}T(\to(\mu_1,\mu_2)). \\ & \quad \mathsf{newid}[\mu_1][\mu_2] \\ & \quad\ (\lambda x{:}T(\mathsf{Tran}[\mu_1]). \\ & \quad\quad \mathsf{M}[\mu_2](f\ (\mathsf{U}[\mu_1]\ x))) \\ \mathsf{M}[\times(\mu_1,\mu_2)] = & \lambda x{:}T(\times(\mu_1,\mu_2)). \\ & \quad \langle \mathsf{M}[\mu_1](\pi_1\ x), \mathsf{M}[\mu_2](\pi_2\ x)\rangle \\ \mathsf{M}[\mathsf{Id}[\mu_1]] = & \lambda x{:}T(\mathsf{Id}[\mu]).x \end{array}$$

$$\begin{array}{rl} \mathsf{U}[\mathsf{Int}] = & \lambda x{:}\mathsf{int}.x \\ \mathsf{U}[\to(\mu_1,\mu_2)] = & \lambda f{:}T(\mathsf{Id}[\to(\mathsf{Tran}[\mu_1],\mathsf{Tran}[\mu_2])]). \\ & \quad \lambda x{:}T(\mu_1). \\ & \quad\quad \mathsf{U}[\mu_2](\mathsf{rpc}[\mu_1][\mu_2]\ f\ (\mathsf{M}[\mu_1]\ x)) \\ \mathsf{U}[\times(\mu_1,\mu_2)] = & \lambda x{:}T(\times(\mathsf{Tran}[\mu_1],\mathsf{Tran}[\mu_2])). \\ & \quad \langle \mathsf{U}[\mu_1](\pi_1\ x), \mathsf{U}[\mu_2](\pi_2\ x)\rangle \\ \mathsf{U}[\mathsf{Id}[\mu]] = & \lambda x{:}T(\mathsf{Id}[\mu]).x \end{array}$$

At arrow types, M converts the function to one that takes and returns transmissible types and then allocates and associates a new identifier with this function via newid. Correspondingly, U takes an identifier and a marshalled argument, performs an rpc on the identifier and argument, takes the result and unmarshalls it.

The M and U functions are used in the translation of client phrases that import a server's function and in the translation of server phrases that export functions. The reader is encouraged to consult Ohori and Kato's paper [39] for further details.

3.3 Type Classes

The language Haskell [25] provides the ability to define a class of types with associated operations called methods. The canonical example is the class of types that admit equality (also known as equality types in SML [33, 19]).

Consider adding a distinguished type void (with associated constructor Void) to λ_i^{ML} in such a way that void is "empty". That is, no closed value has type void. We can encode a type class definition by using Typerec to map types in the class to themselves and types not in the class to void. In this fashion, Typerec may be used to compute a predicate (or in general an n-ary relation) on types. Definitional equality can be used to determine membership in the class.

For example, the class of types that admit equality can be defined using Typerec as follows:

$$\mathsf{Eq} :: \Omega \to \Omega$$

$$\begin{array}{rcl} \mathsf{Eq}[\mathsf{Int}] & = & \mathsf{Int} \\ \mathsf{Eq}[\mathsf{Bool}] & = & \mathsf{Bool} \\ \mathsf{Eq}[\times(\mu_1,\mu_2)] & = & \times(\mathsf{Eq}[\mu_1],\mathsf{Eq}[\mu_2]) \\ \mathsf{Eq}[\to(\mu_1,\mu_2)] & = & \mathsf{Void} \\ \mathsf{Eq}[\mathsf{Void}] & = & \mathsf{Void} \end{array}$$

[3] To compute M and U using the official syntax, we have to use a single typerec that returns a pair holding the two functions for that type.

Here, Eq serves as a predicate on types in the sense that a non-Void constructor μ is definitionally equal to $\mathsf{Eq}[\mu]$ only if μ is a constructor that does not contain the constructor $\to(-,-)$.

The equality method can be coded using typerec as follows, where we assume primitive equality functions for int and bool and omit some type labels for simplicity:

$$\begin{array}{rcl} \mathsf{eq}[\mathsf{Int}] & = & \mathsf{eqint} \\ \mathsf{eq}[\mathsf{Bool}] & = & \mathsf{eqbool} \\ \mathsf{eq}[\times(\mu_1,\mu_2)] & = & \lambda x.\lambda y.\mathsf{eq}[\mathsf{Eq}[\mu_1]](\pi_1\, x)(\pi_1\, y)\ \mathsf{and} \\ & & \quad \mathsf{eq}[\mathsf{Eq}[\mu_2]](\pi_2\, x)(\pi_2\, y) \\ \mathsf{eq}[\to(\mu_1,\mu_2)] & = & \lambda x{:}\mathsf{void}.\lambda y{:}\mathsf{void}.\mathsf{false} \\ \mathsf{eq}[\mathsf{Void}] & = & \lambda x{:}\mathsf{void}.\lambda y{:}\mathsf{void}.\mathsf{false} \end{array}$$

It is straightforward to verify that:

$$\mathsf{eq} : \forall t{::}\Omega.T(\mathsf{Eq}[t]) \to T(\mathsf{Eq}[t]) \to \mathsf{bool}$$

Consequently, $\mathsf{eq}[\mu]\ e_1\ e_2$ can be well typed only if e_1 and e_2 have types that are definitionally equal to $T(\mathsf{Eq}[\mu])$. The encoding is not entirely satisfactory because $\mathsf{eq}[\to(\mu_1,\mu_2)]$ can be a well-typed expression. However, the function resulting from evaluation of this expression can only be applied to values of type void. Since no such values exist, the function can never be applied.

3.4 Dynamics

In the presence of intensional polymorphism a predicative form of the type dynamic [2] may be defined to be the existential type $\exists t{::}\Omega.T(t)$. The typing rules for existential types are as follows:

$$\frac{\Delta \uplus \{t{::}\kappa\} \rhd \sigma \qquad \Delta \rhd \mu :: \kappa \qquad \Delta;\Gamma \rhd e : [\mu/t]\sigma}{\Delta;\Gamma \rhd \mathsf{pack}\, e\ \mathsf{with}\, \mu\ \mathsf{as}\ \exists t{::}\kappa.\sigma : \exists t{::}\kappa.\sigma}$$

$$\frac{\Delta \rhd \sigma \qquad \Delta;\Gamma \rhd e_1 : \exists t{::}\kappa.\sigma' \qquad \Delta \uplus \{t{::}\kappa\};\Gamma \uplus \{x{:}\sigma'\} \rhd e_2 : \sigma}{\Delta;\Gamma \rhd \mathsf{abstype}\, e_1\ \mathsf{is}\ t{::}\kappa, x{:}\sigma'\ \mathsf{in}\ e_2\ \mathsf{end} : \sigma}$$

The pack operation introduces existentials by packaging a type with a value. The abstype operation eliminates existentials by allowing the type and value to be unpacked and used within a certain scope.

Under this interpretation, the introductory form $\mathsf{dynamic}[\tau](e)$ stands for $\mathsf{pack}\, e\ \mathsf{with}\, \tau\ \mathsf{as}\ \exists t{::}\Omega.T(t)$. The eliminatory form, $\mathsf{typecase}\ d\ \mathsf{of}\ (e_{\mathsf{i}}|e_{\to}|e_{\times})$, where d : dynamic, $e_{\mathsf{i}} : \sigma$, and $e_{\to}, e_{\times} : \forall t_1,t_2{::}\Omega.\sigma$, is defined as follows:

$$\mathsf{abstype}\, d\ \mathsf{is}\ t{::}\Omega, x{:}T(t)\ \mathsf{in}\ \mathsf{typerec}\ t\ \mathsf{of}\ [t.\sigma](e_{\mathsf{i}}|e'_{\to}|e'_{\times})\ \mathsf{end}$$

Here $e'_{\to} = \Lambda t_1{::}\Omega.\Lambda t_2{::}\Omega.\lambda x_1{:}\sigma.\lambda x_2{:}\sigma.e_{\to}[t_1][t_2]$, and similarly for $e'_{\times}$.

This form of dynamic type only allows values of monomorphic types to be made dynamic, consistent with the separation between constructors and types in λ_i^{ML}. The possibilities for enriching λ_i^{ML} to admit impredicative quantifiers (and hence account for the full power of dynamic typing including non-termination) are discussed in the conclusion.

4 Related Work

There are two traditional interpretations of polymorphism, the *explicit* style (due to Girard [16, 17] and Reynolds [42]), in which types are passed to polymorphic operations, and the *implicit* style (due to Milner [32]), in which types are erased prior to execution. In their study of the type theory of Standard ML Harper and Mitchell [20] argued that an explicitly-typed interpretation of ML polymorphism has better semantic properties and scales more easily to cover the full language. Harper and Mitchell formulated a predicative type theory, XML, a theory of dependent types augmented with a universe of small types, adequate for capturing many aspects of Standard ML. This type theory was refined by Harper, Mitchell, and Moggi [21], and provides the basis for this work. The idea of intensional type analysis exploited here was inspired by the work of Constable [12, 13], from which the term "intensional analysis" is taken. The rules for typerec, and the need for Typerec, are derived from the "universe elimination" rules in NuPRL (described only in unpublished work of Constable).

The idea of passing types to polymorphic functions is exploited by Morrison *et al.* [37] in the implementation of Napier '88. Types are used at run-time to specialize data representations in roughly the manner described here. The authors do not, however, provide a rigorous account of the type theory underlying their implementation technique. The work of Ohori on compiling record operations [38] is similarly based on a type-passing interpretation of polymorphism, and was an inspiration for the present work. Ohori's solution is *ad hoc* in the sense that no general type-theoretic framework is proposed, but many of the key ideas in his work are present here. Jones [28] has proposed a general framework for passing data derived from types to "qualified" polymorphic operations, called *evidence passing*. His approach differs from ours in that whereas we pass types to polymorphic operations, that are then free to analyze them, Jones passes code corresponding to a proof that a type satisfies the constraints of the qualification. From a practical point of view it appears that both mechanisms can be used to solve similar problems, but the exact relationship between the two approaches is not clear.

Recently Duggan and Ophel [14] and Thatte [50] have independently suggested semantics for type classes that are similar in spirit to our proposal. In particular both approaches represent the restriction of a class as a user-defined, possibly recursive, kind definition in a predicative language. Both sets of authors are concerned with providing a *source*-level overloading facility and consequently examine hard issues such as type inference and open-scoped definitions that do not directly concern us, since we are primarily concerned with a *target*-level type-analysis facility. The implementation technique proposed by Duggan and Ophel is similar to ours in that polymorphic routines are passed type names at run-time and a typecase construct is used to determine the behavior of an overloaded operation. As with type classes and Jones's qualified types, it appears that we can code many of their kind definitions using Typerec with the approach sketched in Section 3.3. However, Typerec can also be used to *transform* types – a facility crucial for representation transformations such as flattening and marshalling. That is, neither Duggan and Ophel nor Thatte provide a facility for coding constructors such as Prod or Tran that map types to types.

A number of authors have considered problems pertaining to representation analysis in the presence of polymorphism. The boxing interpretation of polymorphism has been studied by Peyton Jones and Launchbury [29], by Leroy [30], by Poulsen [41], by Henglein and Jørgensen [24], and by Shao [43] with the goal of minimizing the overhead of boxing and unboxing at run-time. All but the first of these approaches involve copying coercions. Of a broadly similar nature is the work on "soft" type systems [3, 10, 23, 49, 53] that seek to improve data representations through global analysis techniques. All of these methods are based on the use of program analysis techniques to reduce the overhead of box and tag manipulation incurred by the standard compilation method for polymorphic languages. Many (including the soft type systems, but not Leroy's system) rely on global analysis for their effectiveness. In contrast we propose a new approach to compiling polymorphism that affords control over data representation without compromising modularity.

Finally, a type-passing interpretation of polymorphism is exploited by Tolmach [51] in his implementation of a tag-free garbage collection algorithm. Tolmach's results demonstrate that it is feasible to build a run-time system for ML in which no type information is associated with data in the heap[4]. Morrisett, Harper, and Felleisen [36] give a semantic framework for discussing garbage collection, and provide a proof of correctness of Tolmach's algorithm.

5 Summary and Future Directions

We have presented a type-theoretic framework for expressing computations that analyze types at run-time. The key feature of our framework is the use of structural induction on types at both the term and type level. This allows us to express the typing properties of non-trivial computations that perform intensional type analysis. When viewed as an intermediate language for compiling ML programs, much of the type analysis in the translations can be eliminated prior to run-time. In particular, the prenex quantification restriction of ML ensures good binding-time separation between type arguments and value arguments and the "value restriction" on polymorphic functions, together with the well-foundedness of type induction, ensures that a polymorphic instantiation always terminates. This provides important opportunities for optimization. For example, if a type variable t occurring as the parameter of a functor is the subject of intensional type analysis, then the typerec can be simplified when the functor is applied and t becomes known. Similarly, link-time specialization is possible whenever t is defined in a separately-compiled module. Inductive analysis of type variables arising from let-style polymorphism is ordinarily handled at run-time, but it is possible to expand each instance and perform type analysis in each case separately.

The type theory considered here extends readily to inductively defined types such as lists and trees. However, extending typerec and Typerec to handle generally recursive types is problematic because of the negative occurrence of Ω in a recursive constructor. In particular, termination can no longer be guaranteed, which presents problems not only for optimization but also for type checking.

The restriction to predicative polymorphism is sufficient for compiling ML programs. More recent languages such as Quest [9] extend the expressive power to admit impredicative polymorphism, in which quantified types may be

[4]However, types are passed independently as data and associated with code.

instantiated by quantified types. (Both Girard's [16] and Reynolds's [42] calculi exhibit this kind of polymorphism.) It is natural to consider whether the methods proposed here may be extended to the impredicative case. Since the universal quantifier may be viewed as a constant of kind $(\Omega \to \Omega) \to \Omega$, similar problems arise as for recursive types. In particular, we may extend type analysis to the quantified case, but only at the expense of termination, due to the negative occurrence of Ω in the kind of the quantifier. *Ad hoc* solutions are possible, but in general it appears necessary to sacrifice termination guarantees.

Compiling polymorphism using intensional type analysis enables data representations that are impossible using type-free techniques. Setting aside the additional expressiveness of the present approach, it is interesting to consider the performance of a type-passing implementation of ML as compared to the type-free approach adopted in SML/NJ [5]. As pointed out by Tolmach [51], a type-passing implementation need not maintain tag bits on values for the sake of garbage collection. The only remaining use of tag bits in SML/NJ is for polymorphic equality, which can readily be implemented using intensional type analysis. Thus tag bits can be eliminated, leading to a considerable space savings. On the other hand, it costs time and space to pass type arguments at run-time, and it is not clear whether type analysis is cheaper in practice than carrying tag bits. An empirical study of the relative performance of the two approaches is currently planned by the second author, and will be reported elsewhere.

The combination of intensional polymorphism and existential types [35] raises some interesting questions. On the one hand, the type dynamic [2] may be defined in terms of existentials. On the other hand, data abstraction may be violated since a "client" of an abstraction may perform intensional analysis on the abstract type, which is replaced at run-time by the implementation type of the abstraction. This suggests that it may be advantageous to distinguish two kinds of types, those that are analyzable and those that are not. In this way parametricity and representation independence can be enforced by restricting the use of type analysis.

The idea of intensional analysis of types bears some resemblance to the notion of *reflection* [44, 4] — we may think of type-passing as a "reification" of the meta-level notion of types. It is interesting to speculate that the type theory proposed here is but a special case of a fully reflective type theory. The reflective viewpoint may provide a solution to the problem of intensional analysis of recursive and quantified types since, presumably, types would be reified in a syntactic form that is more amenable to analysis — using first-order, rather than higher-order, abstract syntax.

Acknowledgments

We are grateful to Martín Abadi, Andrew Appel, Lars Birkedal, Luca Cardelli, Matthias Felleisen, Andrzej Filinski, Mark Jones, Simon Peyton Jones, Mark Leone, Phil Wadler, Jeannette Wing and the reviewers for their comments and suggestions.

References

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin*. ACM, January 1989.

[2] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. *ACM Transactions on Progamming Languages and Systems*, 13(2):237–268, Apr. 1991. Revised version of [1].

[3] A. Aiken, E. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 163–173, Portland, OR, Jan. 1994.

[4] S. F. Allen, R. L. Constable, D. J. Howe, and W. E. Aitken. The semantics of reflected proof. In *Fifth Symposium on Logic in Computer Science*, pages 95–106, Philadelphia, PA, June 1990. IEEE.

[5] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[6] G. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 102–111, May 1993.

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

[8] L. Cardelli. Phase distinctions in type theory. Unpublished manuscript.

[9] L. Cardelli. Typeful programming. Technical Report 45, DEC Systems Research Center, 1989.

[10] R. Cartwright and M. Fagan. Soft typing. In *Proc. SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 278–292. ACM, June 1991.

[11] D. Clément, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ML. In *1986 ACM Conf. on LISP and Functional Prog.*, 1986.

[12] R. L. Constable. Intensional analysis of functions and types. Technical Report CSR–118–82, Computer Science Department, University of Edinburgh, June 1982.

[13] R. L. Constable and D. R. Zlatin. The type theory of PL/CV3. *ACM Transactions on Progamming Languages and Systems*, 7(1):72–93, Jan. 1984.

[14] D. Duggan and J. Ophel. Kinded parametric overloading. Technical Report CS-94-35, University of Waterloo, Department of Computer Science, September 1994. Supersedes CS-94-15 and CS-94-16, March 1994, and CS-93-32, August 1993.

[15] H. Friedman. Equality between functionals. In R. Parikh, editor, *Logic Colloquium '75*, Studies in Logic and the Foundations of Mathematics, pages 22–37. North-Holland, 1975.

[16] J.-Y. Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, Studies in Logic and the Foundations of Mathematics, pages 63–92. North-Holland, 1971.

[17] J.-Y. Girard. *Interprétation Fonctionnelle et Élimination des Coupures dans l'Arithmétique d'Ordre Supérieure*. PhD thesis, Université Paris VII, 1972.

[18] K. Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. *Dialectica*, 12:280–287, 1958.

[19] C. A. Gunter, E. L. Gunter, and D. B. MacQueen. Computing ML equality kinds using abstract interpretation. *Information and Computation*, 107(2):303–323, Dec. 1993.

[20] R. Harper and J. C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Progamming Languages and Systems*, 15(2):211–252, April 1993. (See also [34].).

[21] R. Harper, J. C. Mitchell, and E. Moggi. Higher-order modules and the phase distinction. In *Seventeenth ACM Symposium on Principles of Programming Languages*, San Francisco, CA, January 1990.

[22] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. Technical Report CMU–CS–94–185, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, September 1994. (Also published as Fox Memorandum CMU–CS–FOX–94–07).

[23] N. Heintze. Set-based analysis of ML programs. In *Proc. 1994 ACM Conf. on LISP and Functional Programming*, pages 306–317, Orlando, FL, June 1994. ACM.

[24] F. Henglein and J. Jørgensen. Formally optimal boxing. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 213–226, Portland, OR, Jan. 1994. ACM.

[25] P. Hudak, S. L. P. Jones, and P. Wadler. Report on the programming language Haskell, version 1.2. *SIGPLAN Notices*, 27(5), May 1992.

[26] M. Jones. Coherence for qualified types. Research Report YALEU/DCS/RR-989, Yale University, New Haven, Connecticut, USA, September 1993.

[27] M. Jones. Partial evaluation for dictionary-free overloading. In *ACM Conference on Partial Evaluation and Semantics-Based Program Manipulation*, 1994.

[28] M. P. Jones. *Qualified Types: Theory and Practice.* PhD thesis, Programming Research Group, Oxford University Computing Laboratory, July 1992. Currently available as Technical Monograph PRG-106, Oxford University Computing Laboratory, Programming Research Group, 11 Keble Road, Oxford OX1 3QD, U.K. email: library@comlab.ox.ac.uk.

[29] S. P. Jones and J. Launchbury. Unboxed values as first-class citizens. In *Proc. Conf. on Functional Programming and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 636–666. ACM, Springer-Verlag, 1991.

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

[31] P. Martin-Löf. About models for intuitionistic type theories and the notion of definitional equality. In S. Kanger, editor, *Proceedings of the Third Scandinavian Logic Symposium*, Studies in Logic and the Foundations of Mathematics, pages 81–109. North-Holland, 1975.

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

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

[34] J. Mitchell and R. Harper. The essence of ML. In *Fifteenth ACM Symposium on Principles of Programming Languages*, San Diego, California, Jan. 1988.

[35] J. C. Mitchell and G. Plotkin. Abstract types have existential type. *ACM Transactions on Progamming Languages and Systems*, 10(3):470–502, 1988.

[36] G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. In preparation, Oct. 1994.

[37] R. Morrison, A. Dearle, R. C. H. Connor, and A. L. Brown. An ad hoc approach to the implementation of polymorphism. *ACM Transactions on Progamming Languages and Systems*, 13(3):342–371, July 1991.

[38] A. Ohori. A compilation method for ML-style polymorphic record calculi. In *Nineteenth ACM Symposium on Principles of Programming Languages*, pages 154–165, Albuquerque, NM, Jan. 1992. Association for Computing Machinery.

[39] A. Ohori and K. Kato. Semantics for communication primitives in a polymorphic language. In *Twentieth ACM Symposium on Principles of Programming Languages*, pages 99–112, Charleston, SC, Jan. 1993. Association for Computing Machinery.

[40] G. Plotkin. Lambda-definability in the full type hierarchy. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 363–373. Academic Press, 1980.

[41] E. R. Poulsen. Representation analysis for efficient implementation of polymorphism. Technical report, Department of Computer Science (DIKU), University of Copenhagen, Apr. 1993. Master Dissertation.

[42] J. C. Reynolds. Towards a theory of type structure. In *Colloq. sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer-Verlag, 1974.

[43] Z. Shao. *Compiling Standard ML for Efficient Execution on Modern Machines.* PhD thesis, Princeton University, Princeton, NJ, November 1994.

[44] B. C. Smith. Reflection and semantics in LISP. In *Eleventh ACM Symposium on Principles of Programming Languages*, pages 23–35, 1984.

[45] R. Statman. Completeness, invariance, and lambda-definability. *Journal of Symbolic Logic*, 47:17–26, 1982.

[46] R. Statman. Logical relations and the typed λ-calculus. *Information and Control*, 65:85–97, 1985.

[47] S. Stenlund. *Combinators, λ-terms and Proof Theory.* D. Reidel, 1972.

[48] W. W. Tait. Intensional interpretation of functionals of finite type. *Journal of Symbolic Logic*, 32(2):187–199, June 1967.

[49] S. R. Thatte. Quasi-static typing. In *Seventeenth ACM Symposium on Principles of Programming Languages*, pages 367–381, San Francisco, CA, Jan. 1990.

[50] S. R. Thatte. Semantics of type classes revisited. In *Proc. 1994 ACM Conference on LISP and Functional Programming*, pages 208–219, Orlando, June 1994. ACM.

[51] A. Tolmach. Tag-free garbage collection using explicit type parameters. In *Proc. 1994 ACM Conference on LISP and Functional Programming*, pages 1–11, Orlando, FL, June 1994. ACM.

[52] A. K. Wright. Polymorphism for imperative languages without imperative types. Technical Report TR93–200, Department of Computer Science, Rice University, Houston, TX, Feb. 1993. To appear, *Lisp and Symbolic Computation.*

[53] A. K. Wright and R. Cartwright. A practical soft type system for Scheme. In *Proc 1994 ACM Conference on LISP and Functional Programming*, pages 250–262, Orlando, FL, June 1994. ACM.