Compiling Polymorphism Using Intensional Type Analysis

Robert Harper Greg Morrisett

September 2, 1994

CMU–CS–94–185

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Also published as Fox Memorandum CMU–CS–FOX–94–07

This research was sponsored by the Defense 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.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Keywords: type theory,polymorphism,lambda calculus,compilation

Abstract

Types have been used to describe the size and shape of data structures at compile time. In polymorphic languages or languages with abstract types, this is not possible since the types of some objects are not known at compile time. Consequently, most implementations of polymorphic languages *box* data (*i.e.*, represent an object as a pointer), leading to inefficiencies. We introduce a new compilation method for polymorphic languages that avoids the problems associated with boxing data. The fundamental idea is to relax the requirement that code selection for primitive, polymorphic operations, such as pairing and projection, must be performed at compile time. Instead, we allow such operations to defer code selection until link- or even run-time when the types of the values are known.

We formalize our approach as a translation into an explicitly-typed, predicative polymorphic λ-calculus with *intensional* polymorphism. By "intensional polymorphism", we mean that constructors and terms can be constructed via structural recursion on types. The form of intensional analysis that we provide is sufficiently strong to perform non-trivial type-based code selection, but it is sufficiently weak that termination of operations that analyze types is assured. This means that a compiler may always "open code" intensionally polymorphic operations as soon as the type argument is known — the properties of the target language ensure that the specialization will always terminate. We illustrate the use of intensional polymorphism by considering a "flattening" translation for tuples and a "marshalling" operation for distributed computing. We briefly consider other applications including type classes, Dynamic types, and "views".

1 Introduction

Types may be thought of as descriptions of data. Compilers for monomorphic languages have considerable leeway in choosing data representations, using types at compile time to guide code selection. For example, a Pascal or C compiler typically uses a "flattened" representation of structures (records) in which consecutive fields are physically adjacent, and in which nested structures are layed out "in line". Access to these structures is determined by the type which determines the size and location of the components of the structure. This allows the programmer to gain considerable control over the representation of data structures, facilitating interaction with ambient hardware and software systems. It is also easy to support a type-safe form of cast whereby a compound data structure may be viewed as a value of a number of different types, provided that all such types describe the same sequence of atomic values.

Extending this flexibility to languages like Modula-3 or Standard ML (SML) is rather more difficult because the type of a value is not always statically apparent. For example, in Modula-3 it is possible to manipulate values of an abstract type that is defined in a separate compilation unit. The compiler cannot determine the representation of the value because the implementation type of the abstraction is unavailable (at least until link time). Similarly, in Standard ML unknown types arise not only because of separate compilation, but also because of the module system polymorphism. For example, when compiling the body of a functor whose parameter declares a type and operations on that type, it is unknown (and fundamentally unknowable!) what is the representation of that type. Similar problems arise with ML-style polymorphism — the type of a variable may be only partially constrained, leaving the exact shape of its value underdetermined.

As a result current compiler technology for polymorphic languages precludes affording the programmer the same degree of control over data representation that is routinely provided in monomorphic languages. Modula-3 imposes the restriction that values of unknown types must be pointers in order to ensure that the representation of values of unknown type is uniform across instances. Most implementations of ML impose a similar restriction, requiring that values of unknown type be "boxed" (stored on the heap and represented by a pointer). Early implementations used a LISP-like representation in which *all* values are boxed [5]; later implementations [31, 32, 30, 24, 43] seek to minimize boxing by taking advantage of whatever type information is manifest in the program. Despite these recent improvements, current implementations still resort to pointer representations for unknown types. Furthermore, current implementations make use of *tag bits* on values to assist garbage collection [5] and to define *polymorphic equality* [5, 6, 18]. Thus representations are further compromised by making it impossible to have 32-bit integers or tag-free tuples with contiguous layout of components.

In this paper we introduce a new compilation method for polymorphic languages that avoids the difficulties introduced by boxing and tagging techniques. The fundamental idea is to relax the requirement that code selection must be performed at compile time. In a monomorphic language code generation for primitive operations such as pairing or projection is determined by the type. For example, different code is generated for the second projection at type $\mathsf{float} * \mathsf{float}$ than for $\mathsf{int} * \mathsf{int}$ since float's typically take more space than int's. In a polymorphic language it is necessary to compile functions such as $\lambda x.\lambda y.\langle x, y\rangle$, in which the types of x and y are unknown. Which pairing operation should be used? Using boxing the compiler ensures that x and y are represented by pointers, for which pairing can be compiled uniformly. We propose instead to *defer* code selection to link- or even run-time when the types of x and y are known. This requires a type-passing interpretation of polymorphism (as suggested by Harper and Mitchell [21]), together with suitable operations for performing code selection based on type parameters.

Our approach is formalized as a translation into an explicitly-typed, predicative polymorphic

λ-calculus with *intensional* or *structural* [18] polymorphism. By "predicative" we mean that monotypes and polytypes are separated, with quantifiers ranging only over monotypes. By "intensional polymorphism" we mean that type parameters are not necessarily treated uniformly, as in the parametric case [45], but rather can significantly affect the course of computation. Following Constable [13, 14] we consider primitive operations for performing *intensional type analysis* [13, 14] in the form of structural recursion on types at both the term and the type level. Intensional type analysis is required at the type, as well as the term, level in order to track the type of intensionally polymorphic operations. This feature distinguishes our approach from other approaches based on typecase [49, 28].

The form of intensional analysis that we provide is sufficiently strong to perform non-trivial type-based code selection, but it is sufficiently weak that termination of operations that analyze types is assured. This means that the compiler may always "open code" intensionally polymorphic operations as soon as the type argument is known — the properties of the target language ensure that the specialization will always terminate. We illustrate the use of intensional polymorphism by considering a "flattening" translation for tuples and a "marshalling" operation for distributed computing (based on Ohori and Kato [42]).

This paper is organized as follows. In Section 2 we describe our approach to compilation as a type-based translation from the source language, Mini-ML, to the target language, λ_i^{ML}. The basic properties of λ_i^{ML} are stated, and a few illustrative examples are given. In Section 3 we give a translation from Mini-ML to λ_i^{ML} in which nested binary products are represented as right-associated binary products. In Section 4, we consider the controlled re-introduction of boxing into our framework. In Section 5 we cast Ohori and Kato's distributed ML compilation in our setting, using intensional polymorphism to determine external representations of types. In Section 6 we briefly consider other applications including type classes, dynamic types and "views". In Section 7 we discuss related work, and in section 8 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 intermediate language, following the interpretation of polymorphism suggested by Harper and Mitchell [21]. The source language is based on Mini-ML [12], 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 [22], a predicative variant of Girard's F_ω [15, 16], enriched with primitives for intensional type analysis. A compiler 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 and in general there are many typing derivations of an expression, 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 [8, 29, 21].[1] Our translations will have the property that $|\Delta|; |\Gamma| \vdash e_t : |\tau|$ is derivable in λ_i^{ML} for a suitable translation of contexts and types into λ_i^{ML}. This allows us to track the typing properties of the translation, and admits consideration of multi-stage type-directed compilation. The exact definitions of the term and type translations will vary from case to case, but the general flavor is to make type abstraction and type instantiation explicit, and to exploit this type-passing interpretation through the use of intensional type analysis in both types and terms.

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

2.1 Source Language: Mini-ML

The source language for our translations is a variant of Mini-ML [12]. The syntax of Mini-ML is defined by the following grammar:

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

Monotypes (τ) are either type variables (t), int, arrow types, or binary product types. *Polytypes* (σ) (also known as *type schemes*) are prenex quantified types. We write $\forall t_1, t_2, \ldots, t_n.\tau$ to represent the polytype $\forall t_1.\forall t_2.\cdots.\forall t_n.\tau$. 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.

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'. Similarly, we use $\Gamma \uplus \{x : \sigma\}$ 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.

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 $\Delta; \Gamma \rhd e : \tau$ where Δ is a set of free type variables and Γ is a type assignment mapping identifiers to polytypes.

The two most interesting rules are the *var* and *let* rules. The *var* rule allows us to conclude that the variable x has type τ' under Γ and Δ if Γ assigns to x the polytype $\forall t_1, \ldots, t_n.\tau$ and τ' is obtained from τ by subtstituting "well-formed" types for $t_1, \ldots, t_n$. These types are well-formed if their free type variables are bound in some outer scope. The scope of type variables is tracked explicitly using Δ, so the type is well-formed if its free type variables are contained in Δ. The *let* rule allows us to assign a polytype ($\forall t_1, \ldots, t_n.\tau$) to the variable x within the expression e provided the following conditions hold: First, the expression bound to the variable x must type-check with type τ under the context that extends the type variables in Δ with $t_1, \ldots, t_n$. Second, the variable x must be bound to a *value*, v, instead of an arbitrary expression. This "value restriction" on polymorphism [20, 33, 52] is needed for our translation. Wright has determined empirically that the value restriction does not affect the vast majority of ML programs [52].

2.2 Target Language: λ_i^{ML}

The target language of our translations, λ_i^{ML}, is based on λ^{ML} [21], a predicative variant of Girard's F_ω [15, 16, 44]. 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 [21] for further discussion of this point.) The language λ_i^{ML} extends λ^{ML} with *intensional* (or *structural* [18]) 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

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

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

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

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

Figure 1: Mini-ML Typing Rules

given below:

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

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 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 carefully distinguish constructors from types, writing $T(\mu)$ for the type corresponding to the monotype 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 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[2] and is used to determine the calling sequence for the function. 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 $\mathsf{Typerec}$ and $\mathsf{typerec}$ 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

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

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 [9, 22].) At the term level typerec may be thought of as a generalization of the typecase operation associated with the type dynamic [1] 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. As will become clear below, it is crucial to provide type recursion at both the constructor and term level so that the type of an intensionally polymorphic operation can itself be defined by intensional type analysis.

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.

$$
\begin{array}{ll}
\Delta \triangleright \mu :: \kappa & \mu \textit{ is a constructor of kind } \kappa \\
\Delta \triangleright \mu_1 \equiv \mu_2 :: \kappa & \mu_1 \textit{ and } \mu_2 \textit{ are equivalent constructors} \\
\Delta \triangleright \sigma & \sigma \textit{ is a valid type} \\
\Delta \triangleright \sigma_1 \equiv \sigma_2 & \sigma_1 \textit{ and } \sigma_2 \textit{ are equivalent types} \\
\Delta; \Gamma \triangleright e : \sigma & e \textit{ is a term of type } \sigma
\end{array}
$$

The formation and equivalence rules for constructors are given in Figures 2 and 3. The formation rules are largely standard, with the exception of the Typerec form. The constructor $\mathsf{Typerec}(\mu; \mu_{\mathsf{i}}; \mu_\times; \mu_\to)$ has kind κ if 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 (Figure 3) axiomatize *definitional equality* [47, 34] of constructors to consist of β-conversion together with recursion equations governing the Typerec form. The level of constructors and kinds is a variation of Gödel's **T** [17]. 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 3 [47]. This reduction relation is confluent, from which it follows that constructor equivalence is decidable [47].

The type formation and equivalence rules for λ_i^{ML} are given in Figure 4. The rules of type equivalence define the interpretation $T(\mu)$ of the constructor μ as a type. The term formation rules are standard (see Figure 5) with the exception of the typerec form, which is governed by the following rule:

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

The argument constructor μ must be of kind Ω, and the result type of the typerec expression is determined as function of the argument constructor. 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.

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 (Decidability) *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 $\rho \vdash e \Rightarrow v$ where e is a λ_i^{ML} expression, ρ is an environment mapping variables to *semantic values*, and v is a semantic value. Semantic values and environments are defined as follows:

$$
\begin{array}{llll}
(\textit{semantic values}) & v & ::= & n \mid \langle v_1, v_2 \rangle \mid (\rho, \lambda x{:}\sigma.\, e) \mid (\rho, \Lambda t{::}\kappa.\, e) \\
(\textit{environments}) & \rho & ::= & \{x_1 \mapsto v_1, \cdots, x_n \mapsto v_n\}
\end{array}
$$

$$\Delta \uplus \{t :: \kappa\} \triangleright t :: \kappa \qquad \Delta \triangleright \mathsf{Int} :: \Omega$$

$$\frac{\Delta \triangleright \mu_1 :: \Omega \qquad \Delta \triangleright \mu_2 :: \Omega}{\Delta \triangleright \to\!(\mu_1, \mu_2) :: \Omega} \qquad \frac{\Delta \triangleright \mu_1 :: \Omega \qquad \Delta \triangleright \mu_2 :: \Omega}{\Delta \triangleright \times(\mu_1, \mu_2) :: \Omega}$$

$$\frac{\Delta \uplus \{t :: \kappa_1\} \triangleright \mu :: \kappa_2}{\Delta \triangleright \lambda t :: \kappa_1.\mu :: \kappa_1 \to \kappa_2} \qquad \frac{\Delta \triangleright \mu_1 :: \kappa' \to \kappa \qquad \Delta \triangleright \mu_2 :: \kappa'}{\Delta \triangleright \mu_1[\mu_2] :: \kappa}$$

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

Figure 2: Formation Rules for Constructors

$$\frac{\Delta \uplus \{t :: \kappa'\} \triangleright \mu_1 :: \kappa \qquad \Delta \triangleright \mu_2 :: \kappa'}{\Delta \triangleright (\lambda t :: \kappa'.\mu_1)[\mu_2] \equiv [\mu_2/t]\mu_1 :: \kappa}$$

$$\frac{\begin{array}{c}\Delta \triangleright \mu_{\mathsf{i}} :: \kappa \\ \Delta \triangleright \mu_{\to} :: \Omega \to \Omega \to \kappa \to \kappa \to \kappa \\ \Delta \triangleright \mu_{\times} :: \Omega \to \Omega \to \kappa \to \kappa \to \kappa\end{array}}{\Delta \triangleright \mathsf{Typerec}(\mathsf{Int}; \mu_{\mathsf{i}}; \mu_{\times}; \mu_{\to}) \equiv \mu_{\mathsf{i}} :: \kappa}$$

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

Figure 3: Equivalence Rules for Constructors

$$\frac{\Delta \triangleright \mu :: \Omega}{\Delta \triangleright T(\mu)} \qquad \Delta \triangleright \mathsf{int}$$

$$\frac{\Delta \triangleright \sigma_1 \qquad \Delta \triangleright \sigma_2}{\Delta \triangleright \sigma_1 \times \sigma_2} \qquad \frac{\Delta \triangleright \sigma_1 \qquad \Delta \triangleright \sigma_2}{\Delta \triangleright \sigma_1 \to \sigma_2} \qquad \frac{\Delta \uplus \{t :: \kappa\} \triangleright \sigma}{\Delta \triangleright \forall t{::}\kappa.\sigma}$$

$$\Delta \triangleright T(\mathsf{Int}) \equiv \mathsf{int} \qquad \frac{\Delta \triangleright \mu_1 :: \Omega \qquad \Delta \triangleright \mu_2 :: \Omega}{\Delta \triangleright T(\to\!(\mu_1, \mu_2)) \equiv T(\mu_1) \to T(\mu_2)} \qquad \frac{\Delta \triangleright \mu_1 :: \Omega \qquad \Delta \triangleright \mu_2 :: \Omega}{\Delta \triangleright T(\times(\mu_1, \mu_2)) \equiv T(\mu_1) \times T(\mu_2)}$$

Figure 4: Type Formation and Equivalence

The semantic values differ from λ_i^{ML} *syntactic* values in that no type information is needed on data structures, such as pairs, and closures ($(\rho, \lambda x{:}\sigma.\, e)$ and $(\rho, \Lambda t{::}\kappa.\, e)$) are used instead of meta-level substitution for value application. Figure 6 defines the evaluation relation using a series of axioms and inference rules. We use $\rho \uplus \{x \mapsto v\}$ to denote the extension of environment ρ so that x is mapped to v, assuming that x is not in the domain of ρ.

The semantics is standard except for the evaluation of a $\mathsf{typerec}$ expression. First, the normal form of the constructor argument is determined. For a well-formed program, we only need to determine normal forms of closed constructors of kind Ω and these are never of the form $\mathsf{Typerec}(\ldots)$, so finding the normal form amounts to evaluating the argument constructor. 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 $\mathsf{typerec}$ at each of the argument constructors.

In order to state a type preservation property for the static semantics with respect to our dynamic semantics, we define a typing judgement for semantic values, $\rhd\, v : \sigma$, and a judgement for environments, $\rhd\, \rho : \Gamma$, as follows:

$$(\mathit{int}) \quad \rhd\, n : \mathsf{int} \qquad\qquad (\mathit{pair}) \quad \frac{\rhd\, v_1 : \sigma_1 \qquad \rhd\, v_2 : \sigma_2}{\rhd\, \langle v_1, v_2\rangle : \sigma_1 \times \sigma_2}$$

$$(\mathit{clos}) \quad \frac{\rhd\, \rho : \Gamma \qquad \emptyset; \Gamma \rhd \lambda x{:}\sigma.\, e : \sigma_1 \to \sigma_2}{\rhd\, (\rho, \lambda x{:}\sigma.\, e) : \sigma_1 \to \sigma_2}$$

$$(t\text{-}\mathit{clos}) \quad \frac{\rhd\, \rho : \Gamma \qquad \emptyset; \Gamma \rhd \Lambda t{::}\kappa.\, e : \forall t{::}\kappa.\sigma}{\rhd\, (\rho, \Lambda t{::}\kappa.\, e) : \forall t{::}\kappa.\sigma}$$

$$(\mathit{env}) \quad \frac{\rhd\, v_1 : \sigma_1 \qquad \cdots \qquad \rhd\, v_n : \sigma_n}{\rhd\, \{x_1 \mapsto v_1, \cdots, x_n \mapsto v_n\} : \{x_1 : \sigma_1, \cdots, x_n : \sigma_n\}}$$

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

By inspection of the semantic value typing rules, only appropriate values occupy appropriate types and thus evaluation will not "go wrong". Furthermore, programs written in pure λ_i^{ML} (*i.e.*, without recursion operators or recursive types) always terminate.

Proposition 2.3 (Termination) *If e is an expression such that $\emptyset; \emptyset \rhd e : \sigma$, then there exists a semantic value v such that $\emptyset \vdash e \Rightarrow v$ and $\rhd\, v : \sigma$.*

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

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

where

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

(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.

$$(var)\quad \frac{\Delta \triangleright \sigma}{\Delta; \Gamma \uplus \{x : \sigma\} \triangleright x : \sigma} \qquad (int)\quad \Delta; \Gamma \triangleright \bar{n} : \mathsf{int}$$

$$(pair)\quad \frac{\Delta; \Gamma \triangleright e_1 : \sigma_1 \qquad \Delta; \Gamma \triangleright e_2 : \sigma_2}{\Delta; \Gamma \triangleright \langle e_1, e_2 \rangle^{\sigma_1, \sigma_2} : \sigma_1 \times \sigma_2} \qquad (\pi)\quad \frac{\Delta; \Gamma \triangleright e : \sigma_1 \times \sigma_2}{\Delta; \Gamma \triangleright \pi_i^{\sigma_1, \sigma_2}\, e : \sigma_i} \quad (i = 1, 2)$$

$$(abs)\quad \frac{\Delta \triangleright \sigma_1 \qquad \Delta; \Gamma \uplus \{x : \sigma_1\} \triangleright e : \sigma_2}{\Delta; \Gamma \triangleright \lambda x{:}\sigma_1.\, e : \sigma_1 \to \sigma_2}$$

$$(app)\quad \frac{\Delta; \Gamma \triangleright e_1 : \sigma' \to \sigma \qquad \Delta; \Gamma \triangleright e_2 : \sigma'}{\Delta; \Gamma \triangleright @^{\sigma'}\, e_1\, e_2 : \sigma}$$

$$(tabs)\quad \frac{\Delta \uplus \{t{::}\kappa\}; \Gamma \triangleright e : \sigma}{\Delta; \Gamma \triangleright \Lambda t{::}\kappa.\, e : \forall t{::}\kappa.\sigma} \qquad (tapp)\quad \frac{\begin{array}{c}\Delta \triangleright \mu{::}\kappa \\ \Delta; \Gamma \triangleright e : \forall t{::}\kappa.\sigma\end{array}}{\Delta; \Gamma \triangleright e[\mu] : [\mu/t]\sigma}$$

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

Figure 5: Term Formation

$$(var)\quad \rho \vdash x \Rightarrow \rho(x) \qquad (int)\quad \rho \vdash \bar{n} \Rightarrow n$$

$$(pair)\quad \frac{\rho \vdash e_1 \Rightarrow v_1 \qquad \rho \vdash e_2 \Rightarrow v_2}{\rho \vdash \langle e_1, e_2 \rangle^{\sigma_1, \sigma_2} \Rightarrow \langle v_1, v_2 \rangle} \qquad (proj)\quad \frac{\rho \vdash e \Rightarrow \langle v_1, v_2 \rangle}{\rho \vdash \pi_i^{\sigma_1, \sigma_2}\, e \Rightarrow v_i} \quad (i = 1, 2)$$

$$(fn)\quad \rho \vdash \lambda x{:}\sigma.\, e \Rightarrow (\rho, \lambda x{:}\sigma.\, e) \qquad (t\text{-}fn)\quad \rho \vdash \Lambda t{::}\kappa.\, e \Rightarrow (\rho, \Lambda t{::}\kappa.\, e)$$

$$(app)\quad \frac{\begin{array}{c}\rho \vdash e_1 \Rightarrow (\rho', \lambda x{:}\sigma.\, e) \qquad \rho \vdash e_2 \Rightarrow v' \\ \rho' \uplus \{x \mapsto v'\} \vdash e \Rightarrow v\end{array}}{\rho \vdash @^{\sigma}\, e_1\, e_2 \Rightarrow v} \qquad (t\text{-}app)\quad \frac{\begin{array}{c}\rho \vdash e \Rightarrow (\rho', \Lambda t{::}\kappa.\, e') \\ \rho' \vdash [\mu/t]e' \Rightarrow v\end{array}}{\rho \vdash e[\mu] \Rightarrow v}$$

$$(trec\text{-}int)\frac{\rho \vdash e_{\mathsf{i}} \Rightarrow v}{\rho \vdash \mathsf{typerec}[t.\sigma](\mu; e_{\mathsf{i}}; e_{\times}; e_{\to}) \Rightarrow v} \quad (NF(\mu) = \mathsf{Int})$$

$$(trec\text{-}pair)\frac{\begin{array}{l}\rho \vdash @^{[\mu_2/t]\sigma}(@^{[\mu_1/t]\sigma}(e_{\times}[\mu_1][\mu_2]) \\ \qquad (\mathsf{typerec}[t.\sigma](\mu_1; e_{\mathsf{i}}; e_{\times}; e_{\to})))(\mathsf{typerec}[t.\sigma](\mu_1; e_{\mathsf{i}}; e_{\times}; e_{\to})) \Rightarrow v\end{array}}{\rho \vdash \mathsf{typerec}[t.\sigma](\mu; e_{\mathsf{i}}; e_{\times}; e_{\to}) \Rightarrow v} \quad (NF(\mu) = \times(\mu_1, \mu_2))$$

$$(trec\text{-}fn)\frac{\begin{array}{l}\rho \vdash @^{[\mu_2/t]\sigma}(@^{[\mu_1/t]\sigma}(e_{\to}[\mu_1][\mu_2]) \\ \qquad (\mathsf{typerec}[t.\sigma](\mu_1; e_{\mathsf{i}}; e_{\times}; e_{\to})))(\mathsf{typerec}[t.\sigma](\mu_1; e_{\mathsf{i}}; e_{\times}; e_{\to})) \Rightarrow v\end{array}}{\rho \vdash \mathsf{typerec}[t.\sigma](\mu; e_{\mathsf{i}}; e_{\times}; e_{\to}) \Rightarrow v} \quad (NF(\mu) = \to(\mu_1, \mu_2))$$

Figure 6: Natural Dynamic Semantics for λ_i^{ML}

As another example, Girard's formulation of System F [15] 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 $\mathsf{typerec}$ as follows:

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

where

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

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 $\mathsf{typerec}$ 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 $\mathsf{typerec}$ expression. The definitions of zero and sizeof are given in this form as follows:

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

Whenever a definition is presented in this form we tacitly assert that it can be formalized using $\mathsf{typerec}$.

3 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 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 giving a translation from Mini-ML monotypes to λ_i^{ML} constructors, written $|\tau|_t$:

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

Here Prod is a constructor of kind $\Omega \to \Omega \to \Omega$ defined below. The translation is extended to polytypes as follows:

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

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 $|\Gamma(x)|$ for each $x \in \mathrm{dom}(\Gamma)$.

Proposition 3.1 *The type translation commutes with substitution:*

$$|[\tau_1, \ldots, \tau_n/t_1, \ldots, t_n]\tau|_t = [|\tau_1|_t, \ldots, |\tau_n|_t/t_1, \ldots, t_n]\,|\tau|_t.$$

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

$$(int) \quad \Delta;\Gamma \rhd \bar{n} : \mathsf{int} \Rightarrow \bar{n}$$

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

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

$$(abs) \quad \frac{\Delta;\Gamma \uplus \{x : \tau_1\} \rhd e : \tau_2 \Rightarrow e'}{\Delta;\Gamma \rhd \lambda x.\, e : \tau_1 \to \tau_2 \Rightarrow \lambda x : |\tau_1|_s.\, e'}$$

$$(app) \quad \frac{\Delta;\Gamma \rhd e_1 : \tau' \to \tau \Rightarrow e_1' \qquad \Delta;\Gamma \rhd e_2 : \tau' \Rightarrow e_2'}{\Delta;\Gamma \rhd e_1\, e_2 : \tau \Rightarrow @^{|\tau'|_s}\, e_1'\, e_2'}$$

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

Figure 7: Flattening Term Translation

The translation maps Mini-ML types to their counterpart constructors in λ_i^{ML}, except that product types are computed using the constructor Prod, which is defined as follows:

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

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

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

and

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

and the equation

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

is derivable in λ_i^{ML}.

The term translation is given in Figure 3 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. The *let* rule makes the implicit type abstraction explicit. The translation of the primitive operations for product types 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:

$$\begin{array}{rcl}\mathsf{mkpair}[\mathsf{Int}][\tau_2] & = & \lambda x : T(\mathsf{Int}).\, \lambda y : T(\tau_2).\, \langle x, y\rangle \\ \mathsf{mkpair}[\rightarrow(\tau_a,\tau_b)][\tau_2] & = & \lambda x : T(\rightarrow(\tau_a,\tau_b)).\, \lambda y : T(\tau_2).\, \langle x, y\rangle \\ \mathsf{mkpair}[\times(\tau_a,\tau_b)][\tau_2] & = & \lambda x : T(\times(\tau_a,\tau_b)).\, \lambda y : T(\tau_2).\, \langle \pi_1\, x, \mathsf{mkpair}[\tau_b][\tau_2](\pi_2\, x)\ y\rangle\end{array}$$

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 $\mathsf{mkpair}[\mathsf{Int}][\tau]$ has type

$$T(\mathsf{Int}) \rightarrow T(\tau) \rightarrow T(\mathsf{Prod}[\mathsf{Int}][\tau])$$

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

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

Similarly, we must check that $\mathsf{mkpair}[\times(\tau_a,\tau_b)][\tau]$ has type

$$T(\times(\tau_a,\tau_b)) \rightarrow T(\tau) \rightarrow T(\mathsf{Prod}[\times(\tau_a,\tau_b)][\tau]$$

which follows from its definition, the derivability of the equation

$$T(\mathsf{Prod}[\times(\tau_a,\tau_b)][\tau]) \equiv T(\tau_a) \times T(\mathsf{Prod}[\tau_b][\tau]),$$

and, inductively, the fact that $\mathsf{mkpair}[\tau_b][\tau]$ has type $\tau_b \rightarrow \tau \rightarrow \mathsf{Prod}[\tau_b][\tau]$.

The operations proj_1 and proj_2 are defined as follows:

$$\begin{array}{rcl}\mathsf{proj}_1[\mathsf{Int}][\tau_2] & = & \lambda x : T(\mathsf{Prod}[\mathsf{Int}][\tau_2]).\, \pi_1\, x \\ \mathsf{proj}_1[\rightarrow(\tau_a,\tau_b)][\tau_2] & = & \lambda x : T(\mathsf{Prod}[\rightarrow(\tau_a,\tau_b)][\tau_2]).\, \pi_1\, x \\ \mathsf{proj}_1[\times(\tau_a,\tau_b)][\tau_2] & = & \lambda x : T(\mathsf{Prod}[\times(\tau_a,\tau_b)][\tau_2]).\, \langle \pi_1\, x, \mathsf{proj}_1[\tau_b][\tau_2](\pi_2\, x)\rangle\end{array}$$

$$\begin{array}{rcl}\mathsf{proj}_2[\mathsf{Int}][\tau_2] & = & \lambda x : T(\mathsf{Prod}[\mathsf{Int}][\tau_2]).\, \pi_2\, x \\ \mathsf{proj}_2[\rightarrow(\tau_a,\tau_b)][\tau_2] & = & \lambda x : T(\mathsf{Prod}[\rightarrow(\tau_a,\tau_b)][\tau_2]).\, \pi_2\, x \\ \mathsf{proj}_2[\times(\tau_a,\tau_b)][\tau_2] & = & \lambda x : T(\mathsf{Prod}[\times(\tau_a,\tau_b)][\tau_2]).\, \mathsf{proj}_2[\tau_b][\tau_2](\pi_2\, x)\end{array}$$

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

The translation given in Figure 3 may be characterized by the following type preservation property.

Theorem 3.2 *If $\Delta;\Gamma \triangleright e : \tau \Rightarrow e'$, then $|\Delta|;|\Gamma| \triangleright e' : |\tau|_t$.*

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 as follows:

$$\mathsf{Prod}[\mu_1][\mu_2] = \mathsf{Append}[\mathsf{Tuple}(\mathsf{ToList}\ \mu_1)][\mathsf{Tuple}(\mathsf{ToList}\ \mu_1)]$$

The Tuple constructor has kind $\Omega^\star \rightarrow \Omega$, where $\kappa^\star$ is the kind of lists whose elements are constructors of kind κ. The Prod constructor coalesces the product of two tuple types into a single tuple type whose components are obtained by appending the fields of the two tuples. Otherwise the ordinary pair (*i.e.*, 2-tuple) of the types is formed. The constructors Append and ToList are defined using Typerec and Listrec as follows:

$$\begin{array}{l}\mathsf{Append}[\mathsf{Nil}][\mu] = \mu \\ \mathsf{Append}[\mathsf{Cons}(\mu_1, \mu_2)][\mu] = \mathsf{Cons}(\mu_1, \mathsf{Append}[\mu_2][\mu]) \\ \\ \mathsf{ToList}[\mathsf{Int}] = \mathsf{Cons}(\mathsf{Int}, \mathsf{Nil}) \\ \mathsf{ToList}[\rightarrow(\mu_1, \mu_2)] = \mathsf{Cons}(\rightarrow(\mu_1, \mu_2), \mathsf{Nil}) \\ \mathsf{ToList}[\mathsf{Tuple}(\mu)] = \mu\end{array}$$

A rigorous formulation of the target language extended with n-tuples is tedious, but appears to be straightforward.

4 Boxing

When type arguments to polymorphic functions are passed explicitly, it is no longer necessary to use boxing to implement polymorphism. For example, the polymorphic function $\lambda x.\lambda y.\langle x, y\rangle$ compiles to $\Lambda t_1{::}\Omega.\Lambda t_2{::}\Omega.\lambda x_1{:}t_1.\lambda x_2{:}t_2.\langle x, y\rangle^{T(t_1),T(t_2)}$, where the pairing primitive is indexed by the types of the components. When using "flat" representations for types, the components of a pair can be large, and the cost of creation or projection can be considerable. An advantage of a "boxed" over a "flat" representation is that large aggregates can be handled atomically. It would seem, then, that the type-passing interpretation of polymorphism is more costly than the boxing interpretation for some applications.

Fortunately, boxing is not incompatible with type-passing. In particular, we can make boxing *explicit* in the source and/or target languages (as suggested by Peyton Jones and Launchbury [30] and Leroy [32]). This allows the programmer (or compiler) to make controlled use of boxing to satisfy either layout requirements (at the cost of certain operations being more expensive) or access requirements (at the cost of introducing indirections).

Boxing may be made explicit in λ_i^{ML} by introducing the following primitives:

$$\begin{array}{l}\mathsf{Box} :: \Omega \rightarrow \Omega \\ \\ \mathsf{box} : \forall t{::}\Omega.T(t) \rightarrow T(\mathsf{Box}[t]) \\ \mathsf{unbox} : \forall t{::}\Omega.T(\mathsf{Box}[t]) \rightarrow T(t)\end{array}$$

In addition we enrich the type language with types of the form $\mathsf{boxed}(\sigma)$ and define $T(\mathsf{Box}[\mu]) \equiv \mathsf{boxed}(T(\mu))$. The Typerec and typerec forms are extended to include a case for "boxed" types as follows:

$$\mathsf{Typerec}(\mathsf{Box}[\mu]; \mu_{\mathsf{i}}; \mu_{\times}; \mu_{\rightarrow}; \mu_{\mathsf{b}}) \quad \equiv \quad \mu_{\mathsf{b}}\, \mu\, (\mathsf{Typerec}(\mu; \mu_{\mathsf{i}}; \mu_{\times}; \mu_{\rightarrow}; \mu_{\mathsf{b}}))$$

$$E[\mathsf{typerec}[t.\sigma](\mathsf{Box}[\mu]; e_{\mathsf{i}}; e_{\times}; e_{\rightarrow}; e_{\mathsf{b}})] \quad \longmapsto \quad E[@^{[\mu/t]\sigma}\, (e_{\mathsf{b}}[\mu])\, (\mathsf{typerec}[t.\sigma](\mu; e_{\mathsf{i}}; e_{\times}; e_{\rightarrow}; e_{\mathsf{b}}))]$$

with the obvious associated kind and type rules.

In the presence of explicit boxing we gain precise control over data layout. For example, we may introduce two forms of product types in Mini-ML, a "flat" form, $\tau_1 \times^\flat \tau_2$, and a "non-flat" form, $\tau_1 \times \tau_2$, with the following translations:

$$\begin{array}{rcl}|\tau_1 \times \tau_2|_t & = & \mathsf{Prod}[\mathsf{Box}[|\tau_1|_t]][\mathsf{Box}[|\tau_2|_t]] \\ |\tau_1 \times^\flat \tau_2|_t & = & \mathsf{Prod}[|\tau_1|_t][|\tau_2|_t]\end{array}$$

The constructor Prod is extended to treat boxed types atomically:

$$\mathsf{Prod}[\mathsf{Box}[\mu_1]][\mu_2] = \times(\mathsf{Box}[\mu_1], \mu_2)$$

Through the use of boxing we may control the trade-off between time and layout constraints.

The interpretation of the boxing and unboxing primitives is left unspecified. The simplest interpretation is heap allocation — values of type $\mathsf{boxed}(\sigma)$ are pointers to values of type σ. As pointed out by Leroy [32, Section 4], this simple interpretation is not always adequate. The "recursive" wrap and unwrap operations considered by Leroy may be defined as follows:

$$\begin{array}{rcl}
\mathsf{wrap}[\mathsf{Int}] & = & \mathsf{box}[\mathsf{Int}] \\
\mathsf{wrap}[\mathsf{Box}[\mu]] & = & \mathsf{identity}[\mathsf{Box}[\mu]] \\
\mathsf{wrap}[\times(\mu_1, \mu_2)] & = & \mathsf{box}[\times(\mathsf{Wrap}[\mu_1], \mathsf{Wrap}[\mu_2])] \circ (\mathsf{wrap}[\mu_1] \times \mathsf{wrap}[\mu_2]) \\
\mathsf{wrap}[\rightarrow(\mu_1, \mu_2)] & = & \mathsf{box}[\rightarrow(\mathsf{Wrap}[\mu_1], \mathsf{Wrap}[\mu_2])] \circ (\mathsf{unwrap}[\mu_1] \rightarrow \mathsf{wrap}[\mu_2]) \\
\\
\mathsf{unwrap}[\mathsf{Int}] & = & \mathsf{unbox}[\mathsf{Int}] \\
\mathsf{unwrap}[\mathsf{Box}[\mu]] & = & \mathsf{identity}[\mathsf{Box}[\mu]] \\
\mathsf{unwrap}[\times(\mu_1, \mu_2)] & = & (\mathsf{unwrap}[\mu_1] \times \mathsf{unwrap}[\mu_2]) \circ \mathsf{unbox}[\times(\mathsf{Wrap}[\mu_1], \mathsf{Wrap}[\mu_2])] \\
\mathsf{unwrap}[\rightarrow(\mu_1, \mu_2)] & = & (\mathsf{wrap}[\mu_1] \rightarrow \mathsf{unwrap}[\mu_2]) \circ \mathsf{unbox}[\rightarrow(\mathsf{Wrap}[\mu_1], \mathsf{Wrap}[\mu_2])]
\end{array}$$

(where $\circ$ is function composition and product and function spaces are extended to functions in the usual way). These definitions can be encoded in a single $\mathsf{typerec}$ that returns a pair consisting of the two functions. The constructor $\mathsf{Wrap} :: \Omega \rightarrow \Omega$ is defined as follows:

$$\begin{array}{rcl}
\mathsf{Wrap}[\mathsf{Int}] & = & \mathsf{Box}[\mathsf{Int}] \\
\mathsf{Wrap}[\mathsf{Box}[\mu]] & = & \mathsf{Box}[\mu] \\
\mathsf{Wrap}[\times(\mu_1, \mu_2)] & = & \times(\mathsf{Wrap}[\mu_1], \mathsf{Wrap}[\mu_2]) \\
\mathsf{Wrap}[\rightarrow(\mu_1, \mu_2)] & = & \rightarrow(\mathsf{Wrap}[\mu_1], \mathsf{Wrap}[\mu_2])
\end{array}$$

With this definition in mind, it is easy to check that

$$\begin{array}{rcl}
\mathsf{wrap} & : & \forall t{::}\Omega. T(t) \rightarrow T(\mathsf{Wrap}[t]) \\
\mathsf{unwrap} & : & \forall t{::}\Omega. T(\mathsf{Wrap}[t]) \rightarrow T(t)
\end{array}$$

5 Marshalling

Ohori and Kato give an extension of ML with primitives for distributed computing in a hetrogenous environment [42]. 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 hetrogenous environment) and are thus represented by proxy using unique identifiers. These identifiers are associated with their functions by servers 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 constructor Id of kind $\Omega \rightarrow \Omega$ 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 $\mathsf{Id}[\mu]$:

$$\mathsf{Typerec}(\mathsf{Id}[\mu]; \mu_{\mathsf{i}}; \mu_{\times}; \mu_{\rightarrow}; \mu_{\mathsf{id}}) \quad \equiv \quad \mu_{\mathsf{id}}\, \mu\, \mathsf{Typerec}(\mu; \mu_{\mathsf{i}}; \mu_{\times}; \mu_{\rightarrow}; \mu_{\mathsf{id}})$$

$$E[\mathsf{typerec}[t.\sigma](\mathsf{Id}[\mu]; e_{\mathsf{i}}; e_{\times}; e_{\rightarrow}; e_{\mathsf{id}})] \quad \longmapsto \quad E[@^{[\mu/t]\sigma}\,(e_{\mathsf{id}}[\mu])\,(\mathsf{typerec}[t.\sigma](\mu; e_{\mathsf{i}}; e_{\times}; e_{\rightarrow}; e_{\mathsf{id}}))]$$

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

$$\begin{array}{rcl}
\mathsf{newid} & : & \forall t_1{::}\Omega.\forall t_2{::}\Omega.(T(\mathsf{Trans}[t_1]) \to T(\mathsf{Trans}[t_2])) \to T(\mathsf{Trans}[\rightarrow(t_1,t_2)]) \\
\mathsf{rpc} & : & \forall t_1{::}\Omega.\forall t_2{::}\Omega.(T(\mathsf{Trans}[\rightarrow(t_1,t_2)])) \to T(\mathsf{Trans}[t_1]) \to T(\mathsf{Trans}[t_2])
\end{array}$$

From an abstract perspective, newid maps a function on representations to a representation of the function and rpc is its (left) inverse. The name newid stems from the representation scheme, which is defined as follows:

$$\begin{array}{rcl}
\mathsf{Trans}[\mathsf{Int}] & = & \mathsf{Int} \\
\mathsf{Trans}[\rightarrow(\mu_1,\mu_2)] & = & \mathsf{Id}[\rightarrow(\mathsf{Trans}[\mu_1], \mathsf{Trans}[\mu_2])] \\
\mathsf{Trans}[\times(\mu_1,\mu_2)] & = & \times(\mathsf{Trans}[\mu_1], \mathsf{Trans}[\mu_2]) \\
\mathsf{Trans}[\mathsf{Id}[\mu]] & = & \mathsf{Id}[\mu]
\end{array}$$

A value of type $T(\mathsf{Trans}[\mu])$ has no arrow types. Instead, $\rightarrow(\mu_1,\mu_2)$ is replaced with an $\mathsf{Id}[-]$ constructor. It is easy to check that Trans is a constructor of kind $\Omega \to \Omega$.

Operationally, rpc takes a proxy identifier of a remote function, and a transmissible argument value. The argument value is sent to the remote server, 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 newid operation takes a function between transmissible values, generates a new, globally unique identifier and associates that identifier with the function.

The compilation of Ohori and Kato's distribution primitives into this extension of λ_i^{ML} relies critically on "marshalling" and "unmarshalling" operations that convert values from a type to its transmissible representation and vice-versa. These are defined simultaneously as follows using the unofficial syntax:

$$\mathsf{M} : \forall t :: \Omega.T(t) \to T(\mathsf{Trans}[t])$$

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

$$\mathsf{U} : \forall t :: \Omega.T(\mathsf{Trans}[t]) \to T(t)$$

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

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 of arrow type and a marshalled argument, performs an rpc on the identifier and argument, takes the result and unmarshals 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 [42] for further details.

6 Other Applications

In this section, we sketch several other applications of intensional polymorphism.

6.1 Type Classes

The language Haskell [25] provides the ability to define a class of types with associated operations called methods. (See [51, 27, 49, 7] for various papers related to type classes.) The canonical example is the class of types that admit equality (also known as equality types in SML [36]).

Consider adding a distinguished type void (with associated constructor Void) in such a way that void is "empty". By empty, we mean that 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:

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

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

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 $\rightarrow(-,-)$.

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

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

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 $\mathsf{eq}[\rightarrow(\mu_1, \mu_2)]$ 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 used.

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

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

Figure 8: Typing Rules for Existentials

6.2 Dynamics

In the presence of intensional polymorphism a predicative form of the type **dynamic** [2] may be defined to be the existential type $\exists t{::}\Omega.T(t)$. 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, $\mathsf{typecase}(d; e_{\mathsf{i}}; e_\times; e_\to)$, where $d : \mathsf{dynamic}$, $e_{\mathsf{i}} : \sigma$, and $e_\times, e_\to : \forall t_1, t_2{::}\Omega.\sigma$, is defined as follows:

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

Here $e'_\times = \Lambda t_1{::}\Omega.\Lambda t_2{::}\Omega.\lambda x_1{:}\sigma.\lambda x_2{:}\sigma.e_\times[t_1][t_2]$, and similarly for $e'_\to$. (The typing rules for **pack** and **abstype** are given in Figure 8.)

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

6.3 Views

One advantage of controlling data representation is that it becomes possible to support a type-safe form of casting which 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 — ie, iff $|\tau_1|_t$ is definitionally equivalent to $|\tau_2|_t$ 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, 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.

Let us extend the source language with a construct for imposing views. If e has type τ and $\tau \approx \tau'$, then the expression $\mathsf{view}\ e\ \mathsf{as}\ \tau'$ has type τ'. By our definition of similarity, no coercion or copying is implied by the imposition of a view. This follows from the fact that similar Mini-ML types are represented by definitionally equal λ_i^{ML} types, and the fact that types are passed to primitive operations to determine their behavior. For example, in the case of the right-associative representation of tuples, we may change views by merely changing the ascribed type, for then the projection operations are given the type of the view, and adjust their behavior according to the imposed view.

In contrast to coercion-based interpretations of type equivalence, such an approach to views is compatible with **ref** types in the sense that τ_1 **ref** is equivalent to τ_2 **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.

7 Related Work

There has traditionally been two interpretations of polymorphism, the *explicit* style (due to Reynolds [44]), in which types are passed to polymorphic operations, and the *implicit* style (due to Milner [35]), in which types are erased prior to execution. In their study of the type theory of Standard ML Harper and Mitchell [21] 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 subsequently refined by Harper, Mitchell, and Moggi [22], and provides the basis for this work. The idea of intensional type analysis exploited here was inspired by the work of Constable [14, 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.* [40] 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. Ohori's work on compiling record operations [41] 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 [26] 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 it is not clear what is the exact relationship between the two approaches. Recently Thatte [49] has suggested a semantics for type classes that is similar in spirit to the present proposal, but lacks the capability to perform intensional type analysis at the constructor level, a crucial feature for tracking the typing properties of intensionally polymorphic operations.

A number of authors have considered problems pertaining to representation analysis in the presence of poylmorphism. The boxing interpretation of polymorphism has been studied by Peyton Jones & Launchbury [30], by Leroy [32], by Poulsen [43], and by Henglein & Jørgensen [24], with the goal of minimizing the overhead of boxing and unboxing at run time. Of a broadly similar nature is the work on "soft" type systems [3, 11, 23, 48, 53] which 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 [50] 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[3]. Morrisett, Harper, and Felleisen [39] give a semantic framework for discussing garbage collection, and provide a proof of correctness of Tolmach's algorithm.

[3]However, types are passed indpendently as data and associated with code.

8 Directions for Future Research

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. The "value restriction" on polymorphic functions, together with the well-founded-ness 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 $\mathsf{typerec}$ 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 does not address analysis of recursive types. Recursive types may be added to λ_i^{ML} by enriching the constructor level with a constant Rec of kind $(\Omega \to \Omega) \to \Omega$, and adding constants representing the isomorphism between $\mathsf{Rec}[\mu]$ and $\mu(\mathsf{Rec}[\mu])$. Extending $\mathsf{typerec}$ and $\mathsf{Typerec}$ to handle recursive types is problematic because of the negative occurrence of Ω in the kind of Rec. In particular, termination can no longer be guaranteed. For the application to data layout, this difficulty is not prohibitive because values of recursive types are "boxed" (by the isomorphism mediating the recursion) and hence not further analyzed. However, it may be important in other applications to analyze recursive types. The most obvious approach is to define evaluation of $\mathsf{typerec}$ at a Rec constructor so that the unrolling is done "lazily". In the case of well-founded recursive types such as lists and trees, this approach is viable because the values themselves are well-founded. However, in general, we lose termination, which presents problems not only for optimization but also for type checking (since $\mathsf{Typerec}$ would no longer terminate).

The restriction to predicative polymorphism is sufficient for compiling ML programs. More recent languages such as Quest [10] extend the expressive power to admit impredicative polymorphism, in which quantified types may be instantiated by quantified types. (Both Girard's [15] and Reynolds's [44] calculi exhibit this kind of poylmorphism.) 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 [50], 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 [38] 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* [46, 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.

It is important to investigate further the relationship between intensional polymorphism and type classes [51, 27]. The primary difference between the two approaches appears to be a trade-off between passing types, from which methods can be chosen based on intensional type analysis, and passing the methods themselves. Passing types seems to give a better handle on the typing properties of non-parametric operations (through the use of Typerec at the constructor level), but it is not clear what are the exact costs and benefits of each approach.

Acknowledgements

We are grateful to Martín Abadi, Andrew Appel, Lars Birkedal, Luca Cardelli, Matthias Felleisen, Andrzej Filinski, Mark Jones, Simon Peyton Jones, and Zhong Shao for their comments and suggestions.

References

[1] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon 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] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically-typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 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, January 1994.

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

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

[6] Andrew W. Appel. A critique of Standard ML. Technical Report CS–TR–364–92, Princeton University, Princeton, NJ, February 1992.

[7] Lennart Augustsson. Implementing Haskell overloading. In *FPCA '93: Conference on Functional Programming Languages and Computer Architecture*, Copenhagen, Denmark, June 1993. ACM Press.

[8] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991.

[9] Luca Cardelli. Phase distinctions in type theory. unpublished manuscript.

[10] Luca Cardelli. Typeful programming. Technical Report 45, DEC Systems Research Center, 1989.

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

[12] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ML. In *1986 ACM Conference on LISP and Functional Programming*, 1986.

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

[14] Robert L. Constable and Daniel R. Zlatin. The type theory of PL/CV3. *ACM Transactions on Programming Languages and Systems*, 7(1):72–93, January 1984.

[15] Jean-Yves 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 Scanindavian Logic Symposium*, Studies in Logic and the Foundations of Mathematics, pages 63–92. North-Holland, 1971.

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

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

[18] Carl A. Gunter, Elsa L. Gunter, and David B. MacQueen. Computing ML equality kinds using abstract interpretation. *Information and Computation*, 107(2):303–323, December 1993.

[19] Robert Harper and Mark Lillibridge. Polymorphic type assignment and CPS conversion. In Olivier Danvy and Carolyn Talcott, editors, *Proceedings of the ACM SIGPLAN Workshop on Continuations CW92*, pages 13–22, Stanford, CA 94305, June 1992. Department of Computer Science, Stanford University. Published as technical report STAN–CS–92–1426.

[20] Robert Harper and Mark Lillibridge. Polymorphic type assignment and CPS conversion. *LISP and Symbolic Computation*, 6(4):361–380, November 1993. (See also [19].).

[21] Robert Harper and John C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, April 1993. (See also [37].).

[22] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Seventeenth ACM Symposium on Principles of Programming Languages*, San Francisco, CA, January 1990.

[23] Nevin 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] Fritz Henglein and Jesper Jørgensen. Formally optimal boxing. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 213–226, Portland, OR, January 1994. ACM.

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

[26] Mark 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.

[27] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *FPCA '93: Conference on Functional Programming Languages and Computer Architecture*, Copenhagen, Denmark, June 1993. ACM Press.

[28] Mark P. Jones. ML typing, explicit polymorphism and qualified types. In *TACS '94: Conference on theoretical aspects of computer software*, Sendai, Japan, April 1994. Springer-Verlag. Lecture Notes in Computer Science, to appear.

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

[30] Simon Peyton Jones and John 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.

[31] Xavier Leroy. The ZINC experiment: an economical implementation of the ML language. Technical Report 117, INRIA, 1990.

[32] Xavier 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.

[33] Xavier Leroy. Polymorphism by name. In *Twentieth ACM Symposium on Principles of Programming Languages*, January 1993.

[34] Per 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.

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

[36] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[37] John Mitchell and Robert Harper. The essence of ML. In *Fifteenth ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1988.

[38] John C. Mitchell and Gordon Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.

[39] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. (Submitted for publication, POPL '95), July 1994.

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

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

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

[43] Eigil Rosager Poulsen. Representation analysis for efficient implementation of polymorphism. Technical report, Department of Computer Science (DIKU), University of Copenhagen, April 1993. Master Dissertation.

[44] John 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.

[45] John C. Reynolds. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing '83*, pages 513–523. Elsevier Science Publishers B. V., 1983.

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

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

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

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

[50] Andrew 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.

[51] Philip Wadler and Stephen Blott. How to make *ad hoc* polymorphism less *ad hoc*. In *Sixteenth ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, January 1989.

[52] Andrew K. Wright. Polymorphism for imperative languages without imperative types. Technical Report TR93–200, Department of Computer Science, Rice University, Houston, TX, February 1993.

[53] Andrew K. Wright and Robert 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.