# Compiling with Non-Parametric Polymorphism (Preliminary Report)

Robert Harper and Greg Morrisett

February 1994
CMU-CS-94-122

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Also published as Fox Memorandum CMU-CS-FOX-94-03

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 author 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:** Polymorphism, non-parametric, ad-hoc, type-directed translation, boxing.

**Abstract**

There is a middle ground between parametric and ad-hoc polymorphism in which a computation can depend upon a type parameter but is restricted to being defined at all types in an inductive fashion. We call such polymorphism *non-parametric*. We show how non-parametric polymorphism can be used to implement a variety of useful language mechanisms including overloading, unboxed data representations in the presence of ML-style polymorphism, and canonical representations of equivalent types. We show that, by using a second-order, explicitly typed language extended with non-parametric operations, these mechanisms can be implemented without having to tag data with type information at run-time. Furthermore, this approach retains a "phase distinction" and permits static type checking and separate compilation. Our aim is to provide a unifying language, translation, and proof framework in which a variety of non-parametric mechanisms can be expressed and verified.

# 1 Introduction

Polymorphism is the parameterization of an expression by a type. Traditionally, polymorphism is divided into two classes, parametric and ad-hoc[13]. Roughly speaking, parametric polymorphism is *uniform* in that the computation cannot depend upon the type that instantiates the parameter, while ad-hoc polymorphism can depend upon the type and need not be defined at all types. In particular, ad-hoc operations such as "+" may only be defined to work on a small number of types.

There is a middle ground between parametric and ad-hoc polymorphism in which the computation can depend upon the type parameter but is restricted to being defined at all types in an inductive fashion. We call such polymorphism *non-parametric*. In this paper, we show how non-parametric polymorphism can be used to implement a variety of useful language mechanisms including overloading, unboxed data representations in the presence of ML-style polymorphism, and canonical representations of equivalent types. We show that, by using a second-order, explicitly typed language extended with non-parametric operations, these mechanisms can be implemented without having to tag data with type information at runtime. Finally, we remark that our approach retains a "phase distinction" and permits static type checking and separate compilation.

Using a second-order explicitly polymorphic language to implement non-parametric language features is not a new idea. Morrison et al. describe an implementation of Napier88 that uses type information at run time to determine properties such as layout of an object[11]. Tolmach has shown how to do "almost tag-free" garbage collection for parametric languages such as SML by encoding them into a second-order language[14]. Ohori shows how polymorphic field selection can be implemented by first encoding the source into a second-order language and then translating types to index structures[12]. However, all of these efforts were directed toward a single language mechanism (e.g. garbage collection) and only Ohori presents a formal account. Our aim is to provide a unifying language, translation, and proof framework in which a variety of non-parametric mechanisms can be expressed and verified.

This paper proceeds as follows: In Section 2, we present a simple source language, based on Mini-ML[4]; an explicitly typed target language $\lambda^{ML}$, based on XML[10]; and a translation technique that compiles Mini-ML into $\lambda^{ML}$. In Section 3, we extend $\lambda^{ML}$ to provide a non-parametric operator, $\mathsf{typerec}$ and in Sections 4, 5, and 6, we show how the translation technique can be used to provide non-parametric language extensions for Mini-ML.

# 2 Compiling ML to $\lambda^{ML}$

## 2.1 The Mini-ML Source Language

Below, we give the abstract syntax for the types, type schemes, expressions, and values of Mini-ML.

$$
\begin{array}{lllll}
\tau & \in & \text{type} & ::= & t \mid \mathsf{unit} \mid \tau_1 \times \tau_2 \mid \tau_1 \rightarrow \tau_2 \\
\sigma & \in & \text{scheme} & ::= & \forall t_1, \ldots, t_n.\tau \\
e & \in & \text{term} & ::= & x \mid \langle\rangle \mid \langle e_1, e_2 \rangle \mid \pi_1\, e \mid \pi_2\, e \mid \mathsf{fix}\, x_1(x_2).e \mid e_1\, e_2 \mid \mathsf{let}\ \ x = v\, \mathsf{in}\, e \\
v & \in & \text{value} & ::= & x \mid \langle\rangle \mid \langle v_1, v_2 \rangle \mid \mathsf{fix}\, x_1(x_2).e
\end{array}
$$

In the interests of simplicity, Mini-ML types include only type variables ($t$), $\mathsf{unit}$, binary products, and functions. Type schemes are prenex quantified types. Expressions include identifiers, unit, tuples, projections, applications, let expressions, and fix expressions of the form $\mathsf{fix}\, x_1(x_2).e$. We will use $\lambda\, x.e$ as an abbreviation for $\mathsf{fix}\, x'(x).e$ when $x'$ is not free in $e$. Note that we restrict $\mathsf{let}$-bound expressions to be values for reasons detailed in Section 2.3.

A standard call-by-value operational semantics in the style of Felleisen et al.[5] (see also Wright and Felleisen[16]) can be given to Mini-ML by defining evaluation contexts (expressions with "holes") and one-step reduction rules:

$$E \quad \in \quad \text{eval context} \quad ::= \quad [\,] \mid \langle E,e\rangle \mid \langle v,E\rangle \mid \pi_i\, E \mid E\, e \mid v\, E$$

$$\begin{array}{lrcll}
(\pi) & E[\pi_i\, \langle v_1,v_2\rangle] & \longmapsto & E[v_i] & (i = 1,2) \\
(\mathsf{fix}_v\beta) & E[(\mathsf{fix}\, x_1(x_2).e)\, v] & \longmapsto & E[[v/x_2][\mathsf{fix}\, x_1(x_2).e/x_1]e] & \\
(\mathsf{let}_v\beta) & E[\mathsf{let}\ \ x = v\, \mathsf{in}\, e] & \longmapsto & E[[v/x]e] &
\end{array}$$

In Section 2.3, we define an equivalent dynamic semantics via translation to another language.

Figure 1 gives a static semantics for Mini-ML by defining two judgements:

$$\Delta \vdash \tau \ \ \mathsf{type} \qquad\qquad \Gamma \vdash_\Delta e : \tau$$

where $\Gamma$ is a type environment mapping identifiers to type schemes and $\Delta$ is a set of type variables. The first judgement establishes the validity of a type in the scope of some bound type variables by requiring all free type variables to be in $\Delta$. The second judgement ascribes a type to an expression. We say that the closed expression $e$ has type $\tau$ if and only if $\emptyset \vdash_\emptyset e : \tau$ is derivable from the set of inference rules. The static semantics is essentially the type system for the core of SML in that polymorphic types are introduced by the $\mathsf{let}$ expression[9]. Unlike SML, we restrict the bound expression that is assigned a generalized type scheme to be a value.

We state the following theorem without proof:

**Theorem 1** *(Mini-ML Type Preservation) If* $\emptyset \vdash_\emptyset e : \tau$ *and* $e \longmapsto^* v$, *then* $\emptyset \vdash_\emptyset v : \tau$.

See Wright and Felleisen[16] for an example of how to prove this theorem.

## 2.2 The $\lambda^{ML}$ Base Target Language

In this section, we introduce a base target language, $\lambda^{ML}$, similar to XML[10]. $\lambda^{ML}$ is a *stratified* second-order $\lambda$-calculus in which the polymorphic types live in a "higher universe" than monotypes. Conceptually, the universe of monotypes is generated inductively, so we are able to give elimination forms corresponding to structural induction on monotypes. As we show in Sections 4, 5, and 6, these induction forms allow us to implement a wide variety of interesting and useful language features in $\lambda^{ML}$.

The four basic syntactic classes of $\lambda^{ML}$, kinds ($k$), type constructors ($u$), types ($\sigma$), and

$$\frac{FTV(\tau) \subseteq \Delta}{\tau \ \ \mathsf{type}}$$

$$\frac{\Gamma(x) = \forall t_1, \ldots, t_n.\tau \qquad \Delta \vdash \tau_1 \ \ \mathsf{type} \quad \cdots \quad \Delta \vdash \tau_n \ \ \mathsf{type}}{\Gamma \vdash_\Delta x : [\tau_1/t_1, \ldots, \tau_n/t_n]\tau}$$

$$\frac{\Gamma \vdash_{\Delta, t_1, \ldots, t_n} v : \tau' \qquad \Gamma, x \mapsto \forall t_1, \ldots, t_n.\tau' \vdash_\Delta e : \tau}{\Gamma \vdash_\Delta \mathsf{let} \ \ x = v \,\mathsf{in}\, e : \tau} \quad (t_i \notin \Delta)$$

$$\frac{}{\Gamma \vdash_\Delta \langle\rangle : \mathsf{unit}}$$

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

$$\frac{\Gamma, x_1 \mapsto \forall.(\tau_1 \to \tau_2), x_2 \mapsto \forall.\tau_1 \vdash_\Delta e : \tau_2}{\Gamma \vdash_\Delta \mathsf{fix}\, x_1(x_2).e : \tau_1 \to \tau_2} \quad (x_1, x_2 \notin \Gamma)$$

$$\frac{\Gamma \vdash_\Delta e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash_\Delta e_2 : \tau_1}{\Gamma \vdash_\Delta e_1 \, e_2 : \tau_2}$$

Figure 1: Static Semantics for Mini-ML

terms ($e$), along with values ($v$) that are a sub-class of terms, are given below:

$$\begin{array}{lllll} k & \in & \text{kind} & ::= & \Omega \mid k_1 \to k_2 \mid k_1 \times k_2 \\ u & \in & \text{constr} & ::= & t \mid \dot{\mathsf{unit}} \mid \dot{\to} \mid \dot{\times} \mid (u_1, u_2) \mid \dot{\pi}_1\ u \mid \dot{\pi}_2\ u \mid \dot{\lambda}\, t : k.u \mid u_1\ u_2 \\ \sigma & \in & \text{type} & ::= & T(\tau) \mid \sigma_1 \times \sigma_2 \mid \sigma_1 \to \sigma_2 \mid \forall t : k.\sigma \\ e & \in & \text{term} & ::= & x \mid \langle\rangle \mid \langle e_1, e_2 \rangle \mid \pi_1\, e \mid \pi_2\, e \mid \Lambda\, t : k\ .e \mid e[u] \mid \mathsf{fix}\, x_1(x_2 : \sigma).e \mid e_1\, e_2 \\ v & \in & \text{value} & ::= & x \mid \langle\rangle \mid \langle v_1, v_2 \rangle \mid \Lambda\, t : k\ .e \mid \mathsf{fix}\, x_1(x_2 : \sigma).e \end{array}$$

The kinds include $\Omega$, the collection of all monotypes, and are closed under products and function spaces. We use $\tau$, a subset of constructors, to range over monotypes:

$$\tau \quad \in \quad \text{monotype} \quad \dot{\mathsf{unit}} \mid \dot{\times}(\tau_1, \tau_2) \mid \dot{\to}(\tau_1, \tau_2)$$

The constructors include monotypes such as $\dot{\mathsf{unit}}$ and type constructors such as $\dot{\to}$. The types of $\lambda^{ML}$, whose elements are terms, include binary products, function spaces, and polymorphic types. In addition, types include explicitly "injected" monotypes ($T(\tau)$). Finally, terms correspond to the basic expression forms of Mini-ML but are written in an explicitly typed syntax. As in Mini-ML, we use $\lambda\, x : \sigma.e$ as an abbreviation for $\mathsf{fix}\, x'(x : \sigma).e$ where $x'$ is not free in $e$. There is no need for a $\mathsf{let}$-construct since this is definable using the $\lambda$-abbreviation together with type abstraction, but we continue to use $\mathsf{let}$ as a syntactic convenience.

The dynamic semantics for $\lambda^{ML}$ is defined by defining evaluation contexts and one-step reduction rules:

$$E \quad \in \quad \text{eval context} \quad ::= \quad [\,] \mid \langle E, e \rangle \mid \langle v, E \rangle \mid \pi_i\, E \mid E[u] \mid E\, e \mid v\, E$$

$$\begin{array}{lrcll} (\pi) & E[\pi_i \langle v_1, v_2 \rangle] & \longmapsto & E[v_i] & (i = 1, 2) \\ (\Lambda\beta) & E[(\Lambda\, t : k\ .e)[u]] & \longmapsto & E[[u/t]e] & \\ (\mathsf{fix}_v\beta) & E[(\mathsf{fix}\, x_1(x_2 : \sigma).e)\, v] & \longmapsto & E[[v/x_2][\mathsf{fix}\, x_1(x_2 : \sigma).e/x_1]e] & \end{array}$$

Note in particular that type abstractions are considered values and type application has an operational reduction.

The static semantics for $\lambda^{ML}$ is broken into a set of formation judgements and equivalence judgements. Full details are given in Appendix A. Throughout, we use $\Gamma$ to denote a type assignment mapping identifiers ($x$) to types ($\sigma$) and we use $\Delta$ to denote a context mapping type variables ($t$) to kinds ($k$).

The formation judgements include constructor formation ($\Delta \vdash u : k$), type formation ($\Delta \vdash \sigma\ \ \mathsf{type}$), and term formation ($\Gamma \vdash_\Delta e : \sigma$). The constructor formation rules are the typing rules for the simply typed $\lambda$-calculus with products. Though $\lambda^{ML}$ types are similar to Mini-ML type schemes, quantification in $\lambda^{ML}$ is not restricted to be prenex and types are required to be closed with respect to quantification over all kinds (not just the kind of monotypes) and function spaces.

Term formation is given by a standard set of typing rules for a second-order calculus, with the exception of a type equivalence rule:

$$\frac{\Gamma \vdash_\Delta e : \sigma_1 \qquad \Delta \vdash \sigma_1 \equiv \sigma_2\ \ \mathsf{type}}{\Gamma \vdash_\Delta e : \sigma_2}$$

This rule is needed to reason about explicitly injected monotypes obtained from constructors (i.e. $T(u)$). Equivalence for constructors is defined using $\beta\eta$-conversion. Hereafter, we shall elide the differences between constructors and their corresponding explicitly injected types.

We state the following theorem without proof:

**Theorem 2** *($\lambda^{ML}$ Type-Preservation): If $\emptyset \vdash_\emptyset e : \sigma$ and $e \longmapsto^* v$, then $\emptyset \vdash_\emptyset v : \sigma$.*

A key property of $\lambda^{ML}$ is that it maintains a *phase* distinction. That is, it is not necessary to reason about term equality in order to show constructor equality. Furthermore, since $\lambda^{ML}$ is explicitly typed, type-checking can be reduced to checking constructor equivalence. Finally, since $\lambda^{ML}$ types are essentially a simply-typed lambda calculus extended with products and a single, inductively generated base kind ($\Omega$), constructor equivalence is decidable and consequently, so is type checking of $\lambda^{ML}$ [8].

## 2.3 Type-Directed Translation

In this section, we present a methodology for compiling Mini-ML-like languages to $\lambda^{ML}$-like languages. Our approach, similar to the one used by Leroy[7], is to use a *type-directed translation* where we first present a translation from Mini-ML types to $\lambda^{ML}$ constructors and then use a term's typing derivation to generate its translation into the appropriate $\lambda^{ML}$ term. Here, we give an example translation (the "standard" translation) that simply demonstrates the technique. In subsequent sections, we extend $\lambda^{ML}$ and use different, but similar translations to handle difficult implementation issues that we gloss over here.

The standard translation on types and schemes is defined by $|\tau|_S$ and $|\sigma|_S$ respectively:

$$\begin{array}{rcl} |t|_S & = & t \\ |\mathsf{unit}|_S & = & \dot{\mathsf{unit}} \\ |\tau_1 \times \tau_2|_S & = & \dot{\times}(|\tau_1|_S, |\tau_2|_S) \\ |\tau_1 \rightarrow \tau_2|_S & = & \dot{\rightarrow}(|\tau_1|_S, |\tau_2|_S) \end{array}$$

$$|\forall t_1, \ldots, t_n.\tau|_S = \forall t_1 : \Omega, \ldots, t_n : \Omega.|\tau|_S$$

The standard translation on terms is defined by the judgement $\Gamma \vdash_\Delta e : \tau \Rightarrow_S |e|$ where $\Gamma$ and $\Delta$ are a Mini-ML type assignment and list of type variables respectively, $e$ is a Mini-ML term, $\tau$ is a Mini-ML type, and $|e|$ is the resulting $\lambda^{ML}$ term. Conceptually, we take the derivation of $\Gamma \vdash_\Delta e : \tau$ and use that to build a corresponding derivation of the translation $|e|$. The judgement is defined formally in Appendix B. Here, we give the two most interesting rules that translate identifiers and let-expressions:

$$\frac{\Gamma(x) = \forall t_1, \ldots, t_n.\tau \qquad \Delta \vdash \tau_1 \ \mathsf{type} \quad \cdots \quad \Delta \vdash \tau_n \ \mathsf{type}}{\Gamma \vdash_\Delta x : [\tau_1/t_1, \ldots, \tau_n/t_n]\tau \Rightarrow_S x[|\tau_1|_S] \ldots [|\tau_n|_S]}$$

$$\frac{\Gamma \vdash_{\Delta, t_1, \ldots, t_n} v : \tau' \Rightarrow_S |v| \qquad \Gamma, x \mapsto \forall t_1, \ldots, t_n.\tau' \vdash_\Delta e : \tau \Rightarrow_S |e|}{\Gamma \vdash_\Delta \mathsf{let}\ x = v\ \mathsf{in}\ e : \tau \Rightarrow_S (\lambda\, x : \forall t_1 : \Omega, \ldots, t_n : \Omega.|\tau'|_S.|e|)(\Lambda\, t_1 : \Omega, \ldots, t_n : \Omega\, .|v|) \qquad (t_i \notin \Delta)}$$

The first rule makes type application explicit while the second rule makes type abstraction explicit. Note that the type translation used in the first rule can produce constructors with free constructor variables. However, due to the requirement that $\Delta \vdash \tau_i \ \ \mathsf{type}$, we can show that the translation has bound these type variables in an outer context using the second rule.

Given the syntax-directed nature of the translation rules, it is apparent that once a derivation of $\Gamma \vdash_\Delta e : \tau$ is fixed, then there exists a unique $|e|$ such that $\Gamma \vdash_\Delta e : \tau \Rightarrow_S |e|$. We state the following theorem without proof, using $|\Gamma|_S$ to denote the type environment that assigns the type $|\Gamma(x)|_S$ to $x$ and $|\Delta|$ to denote the $\lambda^{ML}$ context that assigns the kind $\Omega$ to each variable in $\Delta$.

**Theorem 3** *(Standard Translation Type Preservation): If* $\Gamma \vdash_\Delta e : \tau \Rightarrow_S |e|$, *then* $|\Gamma|_S \vdash_{|\Delta|} |e| : |\tau|_S$.

We state the following theorem without proof, where we use $\theta$ to denote an "observable type" (i.e. unit or products of observable types)[1]:

**Theorem 4** *(Standard Translation Correctness): If* $\emptyset \vdash_\emptyset e : \theta \Rightarrow_S |e|$ *and* $e \longmapsto^* v$, *then there exists a* $|v|$ *such that* $\emptyset \vdash_\emptyset v : \theta \Rightarrow_S |v|$ *and* $|e| \longmapsto^* |v|$.

This theorem can be proved using Theorem 3 and an *erasure* argument, similar to the one described by Harper and Lillibridge[6].

Note that the translation is not valid for Standard ML programs since SML allows non-value expressions bound by a $\mathsf{let}$ to be assigned a generalized type, unlike our definition of Mini-ML (see Section 2.1). For instance, SML would assign a generalized type to the diverging expression bound in the following $\mathsf{let}$-expression:

$$\mathsf{let} \ \ x = (\mathsf{fix}\, y(z).y\, z)\langle\rangle \,\mathsf{in}\, \langle\rangle$$

Since our translation scheme makes type abstraction explicit, our translation would yield the expression

$$(\lambda\,(x : \forall t : \Omega.t).\langle\rangle)(\Lambda t : \Omega\,.(\mathsf{fix}\, y(z : \mathsf{unit}).y\, z)\langle\rangle)$$

that (incorrectly) terminates. Thus, we limit generalization in Mini-ML to syntactically apparent values. Fortunately, Wright has shown that this *value restriction* is reasonable in practice for languages such as SML[17]. Furthermore, this restriction solves the well known problems of generalization in the presence of other computational effects including mutable objects and first class continuations.

Given the value restriction, it is possible to eliminate all polymorphism at compile time by duplicating and/or inlining let-bound values. We do not consider this approach reasonable since it prevents effective separate compilation of polymorphic definitions.

# 3 Non-Parametric $\lambda^{ML}$

In this section, we discuss non-parametric extensions to $\lambda^{ML}$ that allow us to "match and recur" over the constructors of a monotype to determine a computation. Recall from Section

[1] The theorem still holds if we extend $\lambda^{ML}$ with another primitive "observable" type such as $\mathsf{int}$ with more than one value.

2.2 that the type system of $\lambda^{ML}$ is stratified so that polymorphic types live in a higher universe than monotypes and monotypes are conceptually generated inductively. Thus, it is possible to define well-founded induction elimination forms for $\lambda^{ML}$ monotypes. In this section, we show how one general elimination form called typerec may be added to $\lambda^{ML}$ without destroying the phase distinction. In Sections 4 and 5, we show how typerec may be used to implement overloading and canonical representations for Mini-ML. In Section 6, we use other non-parametric extensions to $\lambda^{ML}$ to support unboxed representations.

To add typerec to $\lambda^{ML}$, we augment the syntactic classes as follows:

$$\begin{array}{rcrcll} u & \in & \text{constr} & ::= & \ldots & \mid \mathsf{TypeRec}(\tau;\ u_{\mathsf{u}};\ u_{\times};\ u_{\rightarrow}) \\ e & \in & \text{term} & ::= & \ldots & \mid \mathsf{typerec}(\tau;\ e_{\mathsf{u}};\ e_{\times};\ e_{\rightarrow}) \end{array}$$

We add typerec to the terms and the corresponding TypeRec to the constructors. Intuitively, typerec corresponds to a combination of a "type-case" operator with a "type-recursion" operator, so that computations may depend upon a single type.

Before giving the changes to the dynamic semantics to support typerec, we note that computations involving typerec will depend upon the type constructor used, so we need to evaluate constructors to a canonical form. Since types in $\lambda^{ML}$ are essentially a simply typed $\lambda$-calculus extended with products, TypeRec, and a single base kind of monotypes but no fix-point construct, we note that such normal forms exist and can be found using, for instance, call-by-name, call-by-need, or call-by-value evaluation strategies.

The one-step evaluation relation for terms is augmented so that typerec evaluates its type constructor argument. It then uses the root constructor of the canonical form to select the appropriate sub-term and applies this term to the argument constructors and the result of applying the typerec to these constructors:

$$\begin{array}{rcl} E[\mathsf{typerec}(\dot{\mathsf{unit}};\ e_{\mathsf{u}};\ e_{\times};\ e_{\rightarrow})] & \longmapsto & E[e_{\mathsf{u}}] \\ E[\mathsf{typerec}(\dot{\times}\, u;\ e_{\mathsf{u}};\ e_{\times};\ e_{\rightarrow})] & \longmapsto & E[e_{\times}[\dot{\pi}_1\ u][\dot{\pi}_2\ u] \\ & & \quad (\mathsf{typerec}(\dot{\pi}_1\ u;\ e_{\mathsf{u}};\ e_{\times};\ e_{\rightarrow})) \\ & & \quad (\mathsf{typerec}(\dot{\pi}_2\ u;\ e_{\mathsf{u}};\ e_{\times};\ e_{\rightarrow}))] \\ E[\mathsf{typerec}(\dot{\rightarrow} u;\ e_{\mathsf{u}};\ e_{\times};\ e_{\rightarrow})] & \longmapsto & E[e_{\rightarrow}[\dot{\pi}_1\ u][\dot{\pi}_2\ u] \\ & & \quad (\mathsf{typerec}(\dot{\pi}_1\ u;\ e_{\mathsf{u}};\ e_{\times};\ e_{\rightarrow})) \\ & & \quad (\mathsf{typerec}(\dot{\pi}_2\ u;\ e_{\mathsf{u}};\ e_{\times};\ e_{\rightarrow}))] \end{array}$$

The static semantics are changed so that TypeRec constructors are considered equivalent to their "unrolled" counterparts, and the following term formation rule is added for typerec expressions:

$$\frac{\begin{array}{c} \Delta \vdash \tau : \Omega \ \ \mathsf{type} \qquad \Delta \vdash \forall t : \Omega.\sigma \ \ \mathsf{type} \\ \Gamma \vdash_{\Delta} e_{\mathsf{u}} : \sigma(\dot{\mathsf{unit}}) \\ \Gamma \vdash_{\Delta} e_{\rightarrow} : \forall t_1 : \Omega.\forall t_2 : \Omega.\sigma(t_1) \rightarrow \sigma(t_2) \rightarrow \sigma(\dot{\rightarrow}(t_1, t_2)) \\ \Gamma \vdash_{\Delta} e_{\times} : \forall t_1 : \Omega.\forall t_2 : \Omega.\sigma(t_1) \rightarrow \sigma(t_2) \rightarrow \sigma(\dot{\times}(t_1, t_2)) \end{array}}{\Gamma \vdash_{\Delta} \mathsf{typerec}(\tau;\ e_{\mathsf{u}};\ e_{\rightarrow};\ e_{\times}) : \sigma(\tau)}$$

Note in particular that $\tau$ is restricted to being of kind $\Omega$ (i.e. a monotype).

$$\mathsf{makestring} \;:\; \forall t : \Omega.t \rightarrow \mathsf{string}$$

$$\begin{array}{lrcl}
\mathsf{makestring} & =_{def} & \forall t : \Omega.\mathsf{typerec}(t; e_{\mathsf{u}}; e_{\mathsf{s}}; e_{\times}; e_{\rightarrow}) & \\
\textbf{where} & e_{\mathsf{u}} & = & \lambda x : \mathsf{unit}.\texttt{"<>"} \\
 & e_{\mathsf{s}} & = & \lambda x : \mathsf{string}.(\texttt{"`"} \hat{} x \hat{} \texttt{"'"}) \\
 & e_{\times} & = & \Lambda t_1 : \Omega.\Lambda t_2 : \Omega.\lambda f_1 : t_1 \rightarrow \mathsf{string}.\lambda f_2 : t_2 \rightarrow \mathsf{string}.\lambda x : t_1 \times t_2. \\
 & & & \quad (\texttt{"<"} \hat{} (f_1\,(\pi_1\,x)) \hat{} \texttt{","} \hat{} (f_2\,(\pi_2\,x)) \hat{} \texttt{">"}) \\
 & e_{\rightarrow} & = & \Lambda t_1 : \Omega.\Lambda t_2 : \Omega.\lambda f_1 : t_1 \rightarrow \mathsf{string}.\lambda f_2 : t_2 \rightarrow \mathsf{string}.\lambda x : t_1 \rightarrow t_2. \\
 & & & \quad \texttt{"*fn*"}
\end{array}$$

Figure 2: Defining makestring in $\lambda^{ML}$

# 4 Overloading

In this section we show how typerec can be used to implement various features that are generally classified as *overloading*. A prime example is Standard ML's polymorphic equality (poly-eq) that takes two (non-functional) objects of the same type and returns an indication as to whether they are equal. Another example is a print operation that prints a representation of an arbitrary value. These operations can be easily implemented by $\lambda^{ML}$ +typerec, and can thus be exported to Mini-ML. However, unlike true overloading, the operations must be defined at all types.

Overloaded operations are usually implemented by *tagging* values with constructor information. For example, Standard ML of New Jersey (SML/NJ) tags *all* values with enough information to support polymorphic equality[1]. Our approach does not tag values with constructors. Instead, we pass constructors *only* to polymorphic operations and keep the computations on constructors separate from the computations on values.

As an example, consider adding the primitive type string to $\lambda^{ML}$ along with string literals (e.g. `"foo"`) and an infix concatenation operator ( ^ ). Figure 2 shows how a "makestring" function can be coded using typerec. This function takes a type and computes a function that, when given a value of that type, returns a string representation of the value. The interesting case is the sub-expression $e_{\times}$, that, when applied to two types $\tau_1$ and $\tau_2$, and given functions for converting values of these types to strings, produces a function that takes a product of these two types and converts it to a string. For example, the expression:

$$\mathsf{makestring}[\mathsf{string} \times \mathsf{unit}]\ \langle \texttt{"foo"}, \langle\rangle \rangle$$

evaluates to the string `"<'foo',<>>"`. Note that it is trivial to export the makestring function to Mini-ML as a constant with type $\forall t.t \rightarrow \mathsf{string}$.

As is obvious from the definition of makestring, typerec expressions quickly become unwieldy, so we adopt a pattern matching, recursive style in subsequent definitions. As another example of using typerec to provide an overloaded operation, consider computing the size of a value in machine-words of memory. This is used in languages such as C to allocate memory, copy objects, etc. The sizeof operation can easily be expressed as an overloaded

function using typerec:

$$\begin{aligned}
\mathsf{sizeof}[\dot{\mathsf{unit}}] &= 0 \\
\mathsf{sizeof}[\dot{\mathsf{int}}] &= 1 \\
\mathsf{sizeof}[\dot{\times}(\tau_1, \tau_2)] &= \mathsf{sizeof}[\tau_1] + \mathsf{sizeof}[\tau_2] \\
\mathsf{sizeof}[\dot{\rightarrow}(\tau_1, \tau_2)] &= 2
\end{aligned}$$

Our approach is similar to the "method passing" approach suggested by Wadler and Blott as an implementation technique for Haskell overloading[15]. In the method passing approach, dictionaries of methods (i.e. functions) are passed as "hidden" arguments to overloaded functions. The overloaded functions simply select and invoke the appropriate method from the dictionary. However, Augustsson notes that method passing is difficult to compile efficiently due to the use of higher-order functions[3]. In contrast, our approach passes type constructors which are simply data structures as far as a compiler is concerned. Consequently, our approach may not suffer from the same optimization difficulties as method passing. We further highlight the differences between method passing and our approach in the context of compiling polymorphism in Section 6.3.

# 5 Polymorphism and Canonical Representations

Consider adding an operation view to Mini-ML along with the following typing rule:

$$\frac{\Gamma \vdash_\Delta e : \tau \qquad \tau \equiv \tau'}{\Gamma \vdash_\Delta \mathsf{view}\ e\ \mathsf{as}\ \tau' : \tau'}$$

The rule allows us to *view* $e$ as if it has type $\tau'$ as long as we can prove that $e$ has type $\tau$ and $\tau'$ is *equivalent* to $\tau$. Figure 3 gives one definition of equivalence that is generated by considering tuple types equivalent up to associativity. This situation arises in languages such as C where structs (records) are represented in a "flattened", canonical form. Viewing a struct is more efficient than constructing a copy of the struct with the desired associativity. Furthermore, viewing preserves sharing of the mutable components while copying does not, because we can view *through* refs.

In an abstract sense, view allows us to *observe* that two types are represented by the same canonical form. However, maintaining a canonical form such as C's flattened structs is difficult in the presence of polymorphism. As each value is created, we must insure that the value is in the canonical form. But we cannot determine "the" canonical form if the object is polymorphic.

For example, consider the equivalence relation defined in Figure 3 and suppose we choose right-associated tuples as the canonical representation of tuple types. We cannot determine whether the type $t \times \mathsf{unit}$ is in canonical form or not, because $t$ could be instantiated to unit, in which case the tuple is canonical, or $t$ could be instantiated to $\mathsf{unit} \times \mathsf{unit}$, in which case the type is not canonical. In this section, we show how this canonical form can be maintained by using non-parametric primitive operations in conjunction with typerec. We do not use a "truly flattened" representation of tuples as do C implementations, because $\lambda^{ML}$ only supports binary tuples. However, in Section 7 we discuss adding $n$-tuples to $\lambda^{ML}$ to support such a representation.

$$\frac{}{\tau \equiv \tau} \qquad \frac{\tau_1 \equiv \tau_2 \qquad \tau_2 \equiv \tau_3}{\tau_1 \equiv \tau_3} \qquad \frac{\tau_1 \equiv \tau_1' \qquad \tau_2 \equiv \tau_2'}{\tau_1 \times \tau_2 \equiv \tau_1' \times \tau_2'}$$

$$\frac{\tau_1 \equiv \tau_1' \qquad \tau_2 \equiv \tau_2'}{\tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2'} \qquad \frac{\tau \equiv \tau'}{\tau\ \mathsf{ref} \equiv \tau'\ \mathsf{ref}} \qquad \frac{}{\tau_1 \times (\tau_2 \times \tau_3) \equiv (\tau_1 \times \tau_2) \times \tau_3}$$

Figure 3: Associativity Equivalence of Tuples

We start by giving a translation on types, $|\tau|_\flat$:

$$\begin{array}{rcl} |t|_\flat & = & t \\ |\mathsf{unit}|_\flat & = & \dot{\mathsf{unit}} \\ |\tau\ \mathsf{ref}|_\flat & = & \dot{\mathsf{ref}}(|\tau|_\flat) \\ |\tau_1 \to \tau_2|_\flat & = & \dot{\to}(|\tau_1|_\flat, |\tau_2|_\flat) \\ |\tau_1 \times \tau_2|_\flat & = & \sigma^\flat\ |\tau_1|_\flat\ |\tau_2|_\flat \end{array}$$

where $\sigma^\flat$ is defined using the type constructor $\mathsf{TypeRec}$ to "flatten" tuple types into a right-associated form:

$$\begin{array}{rcl} \sigma^\flat =_{def} \dot{\lambda}\, t_1 : \Omega . \dot{\lambda}\, t_2 : \Omega . \mathsf{TypeRec}(t_1\ ;\ u_u\ ;\ u_\times\ ;\ u_\to) & & \\ u_\mathsf{u} & = & \dot{\times}(\dot{\mathsf{unit}}, t_2) \\ \textbf{where} \quad u_\times & = & \dot{\lambda}\, t_a, t_b, t_a', t_b' : \Omega . \dot{\times}(t_a, t_b') \\ u_\to & = & \dot{\lambda}\, t_a, t_b, t_a', t_b' : \Omega . \dot{\times}(\dot{\to}(t_a, t_b), t_2) \end{array}$$

Recall that the $\mathsf{TypeRec}$ constructor is equivalent to its unrolled counterpart and the unrolling is guaranteed to terminate because the type argument is constrained to be a monotype (kind $\Omega$).

The translation on terms must be modified so that tuples are created in their canonical form and projections extract the appropriate components according to the source types. In the translation rules below, we have encapsulated these operations into three non-parametric functions: $\mathsf{pair}^\flat$, $\mathsf{proj}_1^\flat$ and $\mathsf{proj}_2^\flat$.

$$\frac{\Gamma \vdash_\Delta e_1 : \tau_1 \Rightarrow_\flat |e_1| \qquad \Gamma \vdash_\Delta e_2 : \tau_2 \Rightarrow_\flat |e_2|}{\Gamma \vdash_\Delta \langle e_1, e_2 \rangle : \tau_1 \times \tau_2 \Rightarrow_\flat \mathsf{pair}^\flat[|\tau_1|_\flat][|\tau_2|_\flat]\,|e_1|\,|e_2|}$$

$$\frac{\Gamma \vdash_\Delta e : \tau_1 \times \tau_2 \Rightarrow_\flat |e|}{\Gamma \vdash_\Delta \pi_i\, e : \tau_i \Rightarrow_\flat \mathsf{proj}_i^\flat[|\tau_1|_\flat][|\tau_2|_\flat]\,|e|}$$

The definitions of $\mathsf{pair}^\flat$ and $\mathsf{proj}_i^\flat$ are:

$$\mathsf{pair}^\flat : \forall t_1, t_2 : \Omega . t_1 \to t_2 \to (\sigma^\flat\ t_1\ t_2)$$

$$\begin{array}{rcl} \mathsf{pair}^\flat[\dot{\mathsf{unit}}][\tau_2] & = & \lambda x : \dot{\mathsf{unit}} . \lambda y : \tau_2 . \langle x, y \rangle \\ \mathsf{pair}^\flat[\dot{\to}(\tau_a, \tau_b)][\tau_2] & = & \lambda x : \dot{\to}(\tau_a, \tau_b), \lambda y : \tau_2 . \langle x, y \rangle \\ \mathsf{pair}^\flat[\dot{\times}(\tau_a, \tau_b)][\tau_2] & = & \lambda x : \dot{\times}(\tau_a, \tau_b), \lambda y : \tau_2 . \langle \pi_1\, x, \mathsf{pair}^\flat[\tau_b][\tau_2]\,(\pi_2\, x)\ y \rangle \end{array}$$

$$\mathsf{proj}_1^\flat : \forall t_1, t_2 : \Omega.(\sigma^\flat\ t_1\ t_2) \to t_1$$

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

$$\mathsf{proj}_2^\flat : \forall t_1, t_2 : \Omega.(\sigma^\flat\ t_1\ t_2) \to t_2$$

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

We have used the pattern-matching style definitions for readability, but these can be coded using typerec.

The interesting case in each of the definitions is the product case. For the $\mathsf{pair}^\flat$ function,the product case causes the pair $x$ to be flattened onto $y$ by projecting off the first component of $x$ and pairing it with the flattening of the rest of $x$ with $y$. For the $\mathsf{proj}_1^\flat$ function, the product case projects the $\tau_a$ component using $\pi_1$ and the $\tau_b$ component using $\mathsf{proj}_1^\flat[\tau_b][\tau_2]$ and tuples them to produce the result. For the $\mathsf{proj}_2^\flat$ function, the product case removes the $\tau_a$ component using $\pi_2$ and removes the $\tau_b$ component using $\mathsf{proj}_2^\flat[\tau_b][\tau_2]$.

Whenever $\mathsf{pair}^\flat$ or $\mathsf{proj}_i^\flat$ are used at monomorphic types, the type-applications and typerec occurrences can be eliminated, resulting in a series of primitive pairing and/or projection operations.

# 6 Compiling Polymorphism

In monomorphic languages such as C, types are used to describe the size and shape of a data structure's representation. Operations such as pairing and projection are compiled to primitive operations that differ according to the types. For example, assuming floats are two words and ints are one word, a pairing operation on floats ($\mathsf{pair}_{\mathsf{float,float}}$) will allocate twice as much memory as a pairing operation on ints ($\mathsf{pair}_{\mathsf{int,int}}$). Since all types can be determined statically in a monomorphic language, the compiler can choose the appropriate primitive operation (e.g. $\mathsf{pair}_{\mathsf{int,int}}$ versus $\mathsf{pair}_{\mathsf{float,float}}$) at compile time.

In a polymorphic language, however, it becomes difficult to choose "the" primitive operation at compile time, because the type might vary at runtime through polymorphic instantiation. This issue is entirely glossed over by the standard translation of Section 2.3. In this section, we discuss three techniques for solving this problem. The first technique, used by the SML/NJ 0.93 system, abandons the idea that types describe the shape of objects and converts objects to a universal representation so that a single primitive operation suffices for all cases and can always be selected at compile time. However, this approach introduces indirection in the data structures which can be expensive in both space and time. The second approach, described by Leroy and used in the Gallium compiler, compiles monomorphic code using the natural representation and, roughly speaking, polymorphic code using

the universal representation. Coercions are introduced to convert values to and from the universal representation as necessary, yet the coercions and all primitive operations are selected at compile time. However, Leroy's approach does not extend directly to mutable objects and does not work well with large objects. The third approach uses *non-parametric* primitive operations to avoid *ever* having to introduce indirection in data structures at the cost of selecting some primitive operations at run time. Furthermore, the approach extends directly to mutable objects. It is particularly illuminating to see all three approaches in our type-directed translation framework.

### 6.1 The Boxing Approach

In SML/NJ (versions 0.93 and earlier)[2], all objects are represented in a *universal* fashion so that primitive operations may be selected at compile time. This is accomplished by *boxing* values — that is, placing objects larger than size one in memory and using a pointer to the object as its representation.

We can cast the boxing strategy into our type-directed framework as follows. We add to the set of $\lambda^{ML}$ monotypes, $\dot{\mathsf{box}}(\tau)$, representing boxed values and we assume two new families of operations, $\mathsf{box}_\tau$ and $\mathsf{unbox}_\tau$ that convert objects to and from the boxed representation at the appropriate type. We further assume that primitive operations such as pairing and projection work uniformly on boxed types. So, for example, $\dot{\mathsf{pair}}_{\dot{\mathsf{box}}(\mathsf{int}),\dot{\mathsf{box}}(\mathsf{int})} = \dot{\mathsf{pair}}_{\dot{\mathsf{box}}(\mathsf{float}),\dot{\mathsf{box}}(\mathsf{float})}$ and we shall write $\dot{\mathsf{pair}}_{\dot{\mathsf{box}},\dot{\mathsf{box}}}$ for the primitive pairing operation on boxed values, regardless of the type.

The type translation for the boxing strategy is defined as $|\tau|_B$ using an auxiliary definition, $||\tau||_B$:

$$\begin{aligned}
||t||_B &= t \\
||\mathsf{unit}||_B &= \dot{\mathsf{unit}} \\
||\tau_1 \times \tau_2||_B &= \dot{\times}(|\tau_1|_B, |\tau_2|_B) \\
||\tau_1 \to \tau_2||_B &= \dot{\to}(|\tau_1|_B, |\tau_2|_B)
\end{aligned}$$

$$|\tau|_B = \dot{\mathsf{box}}\ ||\tau||_B$$

$$|\forall t_1, \ldots, t_n.\tau|_B = \forall t_1 : \Omega, \ldots, t_n : \Omega.|\tau|_B$$

The full term translation is given in Appendix C, but we note here that introduction rules $\mathsf{box}$ their results while elimination rules $\mathsf{unbox}$ their arguments. For example, the pair and projection rules are:

$$\frac{\Gamma \vdash_\Delta e_1 : \tau_1 \Rightarrow_B |e_1| \qquad \Gamma \vdash_\Delta e_2 : \tau_2 \Rightarrow_B |e_2|}{\Gamma \vdash_\Delta \langle e_1, e_2 \rangle : \tau_1 \times \tau_2 \Rightarrow_B \mathsf{box}_{\mathsf{box}\times\mathsf{box}}(\dot{\mathsf{pair}}_{\mathsf{box},\mathsf{box}}(|e_1|, |e_2|))}$$

$$\frac{\Gamma \vdash_\Delta e : \tau_1 \times \tau_2 \Rightarrow_B |e|}{\Gamma \vdash_\Delta \pi_i\, e : \tau_i \Rightarrow_B \dot{\mathsf{proj}}_{i,\mathsf{box}\times\mathsf{box}}(\mathsf{unbox}_{\mathsf{box}}(|e|))} \quad (i = 1,2)$$

The identifier rule must use the *auxiliary* type translation, $||\tau||_B$, in instantiation since we

have already required that $t$ be translated to $\dot{\mathsf{box}}(t)$:

$$\frac{\Gamma(x) = \forall t_1, \ldots, t_n.\tau \qquad \Delta \vdash \tau_1 \ \mathsf{type} \quad \ldots \quad \Delta \vdash \tau_n \ \mathsf{type}}{\Gamma \vdash_\Delta x : [\tau_1/t_1, \ldots, \tau_n/t_n]\tau \Rightarrow_B x[||\tau_1||_B] \ldots [||\tau_n||_B]}$$

It is straightforward to show that the boxing translation like the standard translation preserves its type translation. The key lemma one must prove is that $||\rho||_B(|\tau|_B) = |\rho(\tau)|_B$, where $\rho$ is the substitution:

$$\rho = \{t_1 \mapsto \tau_1, \ldots, t_n \mapsto \tau_n\}$$

and $||\rho||_B$ is the substitution that maps $t$ to $||\rho(t)||_B$. Once type correctness of the translation is established, it is easy to see that all primitive operations have been effectively selected at compile time by the translation, since all values are boxed and there is only one primitive operation for boxed values, regardless of the type.

### 6.2 Leroy's Approach

The boxing compilation strategy produces excessive indirection which can cost substantially in both space and time. Leroy proposed an alternative strategy that only boxes values that are given the type $t$ (a type variable.)[7]. Consequently, only polymorphic code pays the price of indirection.

We can cast Leroy's compilation strategy into our type-directed translation framework as follows. We define a type translation from Mini-ML to $\lambda^{ML}$ (extended with $\dot{\mathsf{box}}$) that only boxes polymorphic objects:

$$\begin{aligned} |t|_L &= \dot{\mathsf{box}}\ t \\ |\mathsf{unit}|_L &= \dot{\mathsf{unit}} \\ |\tau_1 \times \tau_2|_L &= \dot{\times}(|\tau_1|_L, |\tau_2|_L) \\ |\tau_1 \rightarrow \tau_2|_L &= \dot{\rightarrow}(|\tau_1|_L, |\tau_2|_L) \end{aligned}$$

$$|\forall t_1, \ldots, t_n.\tau|_L = \forall t_1 : \Omega, \ldots, t_n : \Omega.|\tau|_L$$

The term translation is similar to the standard translation of Section 2.3. In particular, no boxing is introduced by the value creation rules. However, a type mismatch arises in the identifier translation rule, because a polymorphic object is compiled as if $t$ is boxed, but the *use* expects it to be unboxed. That is, if $\rho$ is the substitution used in the instantiation rule to map type variables to types, then $|\rho|_L(|\tau|_L) \neq |\rho(\tau)|_L$. Leroy suggests applying a *coercion*, $S_\rho$ to the polymorphic object at its uses to convert the object to the appropriate type, based on $\rho$. Thus, the translation rule for identifiers, making $\rho$ explicit becomes:

$$\frac{\Gamma(x) = \forall t_1, \ldots, t_n.\tau \qquad \rho = \{t_1 \mapsto \tau_1, \ldots, t_n \mapsto \tau_n\} \qquad \Delta \vdash \rho(t_1) \ \mathsf{type} \quad \ldots \quad \Delta \vdash \rho(t_n) \ \mathsf{type}}{\Gamma \vdash_\Delta x : [\rho(t_1)/t_1, \ldots, \rho(t_n)/t_n]\tau \Rightarrow_L S_\rho(x;\tau)[|\rho(t_1)|_L] \ldots [|\rho(t_n)|_L]}$$

The definition of $S_\rho$ is given below along with the dual coercion, $G_\rho$, that is necessary for coercing functions. $S$ stands for "specialization" and $G$ for "generalization". Technically,

we should use the type-specific primops such as $\mathsf{pair}_{\tau_1,\tau_2}$ and $\mathsf{proj}_{i,\tau_1,\tau_2}$ in the definitions instead of the generic primops such as $\langle\cdot,\cdot\rangle$ and $\pi_i$.

$$
\begin{array}{rcll}
S_\rho(e;t) & = & \mathsf{unbox}_{|\rho(t)|_L}(e) & (t \in Dom(\rho)) \\
S_\rho(e;t) & = & e & (t \notin Dom(\rho)) \\
S_\rho(e;\mathsf{unit}) & = & e & \\
S_\rho(e;\tau_1 \times \tau_2) & = & \mathsf{let}\ \ x = e\ \mathsf{in}\ \langle S_\rho(\pi_1\, x;\tau_1), S_\rho(\pi_2\, x;\tau_2)\rangle & \\
S_\rho(e;\tau_1 \to \tau_2) & = & \mathsf{let}\ \ f = e\ \mathsf{in}\ \lambda\, x : |\tau_1|_L . S_\rho(f\ (G_\rho(x;\tau_2));\tau_1) &
\end{array}
$$

$$
\begin{array}{rcll}
G_\rho(e;t) & = & \mathsf{box}_{|\rho|_L(|t|_L)}(e) & (t \in Dom(\rho)) \\
G_\rho(e;t) & = & e & (t \notin Dom(\rho)) \\
G_\rho(e;\mathsf{unit}) & = & e & \\
G_\rho(e;\tau_1 \times \tau_2) & = & \mathsf{let}\ \ x = e\ \mathsf{in}\ \langle G_\rho(\pi_1\, x;\tau_1), G_\rho(\pi_2\, x;\tau_2)\rangle & \\
G_\rho(e;\tau_1 \to \tau_2) & = & \mathsf{let}\ \ f = e\ \mathsf{in}\ \lambda\, x : |\tau_1|_L . G_\rho(f\ (S_\rho(x;\tau_2));\tau_1) &
\end{array}
$$

In order to prove translation type preservation, we need to show that $S_\rho$ is a function of type $|\rho|_L(|\tau|_L) \to |\rho(\tau)|_L$. Similarly, we need to show that $G_\rho$ has type $|\rho(\tau)|_L \to |\rho|_L(|\tau|_L)$.

As with the boxing translation, Leroy's coercion translation has effectively selected the primitive operations at compile time. Furthermore, there appears to be less boxing/unboxing of values. In particular, boxing/unboxing only occurs if polymorphism is used.

### 6.3 Our Approach

Unfortunately, Leroy's approach does not directly extend to mutable data structures such as refs or arrays, and is impractical for "large" objects such as vectors and lists. The problem is that $S_\rho$ and $G_\rho$ essentially *copy* an object, changing some of the components to/from the boxed state. Copying is expensive for large objects and is incorrect for mutable objects. If we add ref to Mini-ML, then we would like to leave the contents of ref cells unboxed in the translation to $\lambda^{ML}$ but we cannot extend the $S_\rho$ and $G_\rho$ conversions directly to support this. A first attempt at extending $S_\rho$ to refs is:

$$S_\rho(e;\tau\ \mathsf{ref}) \quad = \quad \mathsf{ref}(S_\rho(!e;\tau))$$

but this does not preserve sharing of refs. A second attempt at extending $S_\rho$ is:

$$
\begin{array}{rcl}
S_\rho(e;\tau\ \mathsf{ref}) & = & \mathsf{let}\ x\ =\ e \\
 & & \mathsf{in} \\
 & & \qquad x := S_\rho(!x;\tau) \\
 & & \mathsf{end}
\end{array}
$$

but this is unsound since the contents of $x$ must simultaneously have type $|\rho(\tau)|_L$ and $|\rho|_L(|\tau|_L)$, which are not necessarily the same.

Leroy suggests two ways to fix this shortcoming. The first technique treats a ref cell (or large object) as if its contents were polymorphic and thus the contents are boxed. In fact, as Leroy points out, the contents must be *recursively* boxed. To see the problem, note that a ref containing a tuple of must be "viewable" as not only a $t\,\mathsf{ref}$, but also a $(t_1 \times t_2)\,\mathsf{ref}$. A consequence of this is that the $S$ and $G$ coercions must be changed at type variables to recursively unbox/box values. Obviously, this approach can be quite expensive. As an

example, a polymorphic array instantiated to be a float array must box all of the floats. If the array is instantiated to be an array of complex numbers (i.e., pairs of floating point numbers), the tuples and their components must be boxed.

The second technique, used by the Gallium compiler, associates *methods* (i.e., functions) with a ref/array to read and write components of the ref/array. In this fashion, Gallium is able to keep the components of the mutable object unboxed since the $S$ and $G$ conversions can be applied to the methods instead of the data. The method passing approach was also proposed by Wadler and Blott as an implementation technique for Haskell overloading[15].

An alternative approach to method passing that we propose, is to pass type information to the read and write operations on arrays and let them compute the appropriate primitive, monomorphic read/write operation, possibly at runtime. This amounts to making read and write *non-parametric* polymorphic operations. Furthermore, we note that other data structures, such as tuples, may always be unboxed by making pairing and projection non-parametric polymorphic operations. In fact, boxing can be eliminated *entirely* from data structures[2] simply by translating all polymorphic primitive operations to non-parametric polymorphic functions that compute the appropriate monomorphic operation based on the type. Of course, if the operations are applied to monomorphic types, then the type computations may be eliminated at compile time. Thus, like the method passing approach, only polymorphic code pays the price of computing a primitive operation at compile time.

Here we sketch briefly and at a high-level how the non-parametric approach is formalized in our framework, so that boxing may be eliminated from tuples. The same techniques can be applied to function application (to unbox arguments and place them in registers), refs and arrays, and other data structures. We assume that the following non-parametric operators are added to $\lambda^{ML}$, as we added typerec:

$$\begin{array}{rcl} \mathsf{pair} & : & \forall t_1, t_2 : \Omega.t_1 \to t_2 \to (t_1 \times t_2) \\ \mathsf{proj}_1 & : & \forall t_1, t_2 : \Omega.(t_1 \times t_2) \to t_1 \\ \mathsf{proj}_2 & : & \forall t_1, t_2 : \Omega.(t_1 \times t_2) \to t_2 \end{array}$$

These operations are essentially functions that compute the appropriate primitive operation according to their type arguments, so we add the following corresponding reduction rules:

$$\begin{array}{rcl} E\,[\mathsf{pair}[\tau_1][\tau_2]] & \longmapsto & \lambda\, x_1 : \tau_1.\lambda\, x_2 : \tau_2.\mathsf{pair}_{\tau_1,\tau_2}\, x_1\, x_2 \\ E\,[\mathsf{proj}_1[\tau_1][\tau_2]] & \longmapsto & \lambda\, x : (\tau_1 \times \tau_2).\mathsf{proj}_{1,\tau_1,\tau_2}\, x \\ E\,[\mathsf{proj}_2[\tau_1][\tau_2]] & \longmapsto & \lambda\, x : (\tau_1 \times \tau_2).\mathsf{proj}_{2,\tau_1,\tau_2}\, x \end{array}$$

It is reasonable to assume such operations because these functions are defined by a compiler for a monomorphic language.

The translation is straightforward: we simply map $\langle e_1 : \tau_1, e_2 : \tau_2 \rangle$ to $\mathsf{pair}[|\tau_1|][|\tau_2|]\,|e_1|\,|e_2|$ and similarly for projection. However, we can specialize the translation in order to guarantee that the appropriate primitive operations are chosen by the translation for monomorphic code. This is done by introducing two translation rules for each operation: one for the monomorphic case that performs the above reductions at compile time and one for the

[2]It is worth noting that boxing cannot be directly eliminated from closures based on types, because the type does not tell us the size of the code nor the types of the free variables.

general case where the reduction is delayed, possibly until run time.

$$\frac{\Gamma \vdash_\Delta e_1 : \tau_1 \Rightarrow_O |e_1| \qquad \Gamma \vdash_\Delta e_2 : \tau_2 \Rightarrow_O |e_2| \qquad FTV(\tau_1) = FTV(\tau_2) = \emptyset}{\Gamma \vdash_\Delta \langle e_1,e_2\rangle : \tau_1 \times \tau_2 \Rightarrow_O \mathsf{pair}_{\tau_1,\tau_2}\ |e_1|\ |e_2|}$$

$$\frac{\Gamma \vdash_\Delta e_1 : \tau_1 \Rightarrow_O |e_1| \qquad \Gamma \vdash_\Delta e_2 : \tau_2 \Rightarrow_O |e_2| \qquad FTV(\tau_1) \neq \emptyset \qquad FTV(\tau_2) \neq \emptyset}{\Gamma \vdash_\Delta \langle e_1,e_2\rangle : \tau_1 \times \tau_2 \Rightarrow_O \mathsf{pair}[\tau_1][\tau_2]\,|e_1|\ |e_2|}$$

$$\frac{\Gamma \vdash_\Delta e : \tau_1 \times \tau_2 \Rightarrow_O |e| \qquad FTV(\tau_1) = FTV(\tau_2) = \emptyset}{\Gamma \vdash_\Delta \pi_i\, e : \tau_i \Rightarrow_O \mathsf{proj}_{i,\tau_1,\tau_2}\ |e|}$$

$$\frac{\Gamma \vdash_\Delta e : \tau_1 \times \tau_2 \Rightarrow_O |e| \qquad FTV(\tau_1) \neq \emptyset \qquad FTV(\tau_2) \neq \emptyset}{\Gamma \vdash_\Delta \pi_i\, e : \tau_i \Rightarrow_O \mathsf{proj}_i[\tau_1][\tau_2]\,|e|}$$

It is worth noting that as soon as a type becomes closed, these reductions can be made. Consequently, if the non-parametric approach is used to implement polymorphism at the module level, these type reductions can occur when a functor is applied to a structure argument.

The boxing approach and Leroy's approach have one advantage over message-passing and our approach: no restriction is needed on polymorphic generalization (see Section 2.3), so these techniques work directly for Standard ML. But, as mentioned earlier, Wright has found that the value restriction is not a problem in practice. Furthermore, the non-parametric approach has an important advantage over boxing and Leroy's technique for compiling polymorphism: Since boxing need never be introduced (nor prohibited) in the representation of data structures, interfacing to an external entity such as a C procedure, or the operating system, or a device becomes much simpler and more efficient since the representation demanded by the external entity can be met directly. This is true, of course, only if *tags* for other purposes such as garbage collection or polymorphic equality can be eliminated from the representation. Fortunately, as we have shown earlier, overloaded operations such as polymorphic equality can be implemented as a non-parametric operation, so values do not need to be tagged for this purpose. Furthermore, Tolmach has recently shown that by translating ML into a second-order language like $\lambda^{ML}$, copying garbage collection can also be implemented without the need to tag objects[14].

Of course, method passing can be used to eliminate boxing as well, since method passing and our approach appear to be duals of each other. However, with method passing, a method for each primitive operation (including pairing, reading/writing from a ref, etc.) must be *eagerly* constructed when a polymorphic function is instantiated because we cannot tell how the object we pass to the function will be used. Augustsson reports that computing all of these methods is a source of inefficiency in most implementations of Haskell type classes[3]. In contrast, for a non-parametric operation, "methods" are constructed *lazily* at the point

where they are used while a single type is constructed eagerly. Which approach leads to superior performance is application and implementation dependent. Fortunately, the two approaches are not incompatible: if a primitive operation is found to be frequently used, for instance computing the size of an object, then the method passing approach can be used to implement the operation. Other operations can be implemented using non-parametricity, with the expectation that they will be invoked at most once.

# 7 Conclusions and Future Work

We have demonstrated that by translating ML-like languages into a second-order, explicitly typed language extended with the ability to define *non-parametric* operations, we can address and solve a variety of hard language implementation problems, including overloading and unboxed representations in the presence of polymorphism. Our approach is to define a *type-directed* translation from the source language to the target language. The target languages we have used are based on $\lambda^{ML}$, a *stratified* language in which monotypes can be thought of as being inductively defined. Consequently, we are able to extend $\lambda^{ML}$ with induction elimination forms (e.g. typerec) to support the definition of non-parametric operations, yet type checking for the target language remains decidable.

The problems we have concentrated on are essentially *representation* issues: We are concerned with providing the programmer and/or the implementor of a polymorphic language with maximal control over representations of data structures. In particular, we have shown how unnecessary indirection (boxing) may be eliminated even in the presence of polymorphism and we have shown that it is not necessary to tag objects to support language features such as overloading. Others, such as Tolmach and Ohori, have backed this claim by demonstrating that even more language features, notably copying garbage collection and polymorphic field selection, can be implemented using this same general technique.

We still have many difficult open problems to address: First, we expect that $\lambda^{ML}$ may be extended to work with n-tuples instead of just binary tuples, but the details have not been worked out. The basic idea is to add $\Omega$ list to the kinds and consider "n-tuple" as a constructor of kind $\Omega$ list $\rightarrow \Omega$. Then, an induction elimination form (fold) on these lists may be added as a term. N-tuples are necessary to implement truly flattened representations of tuples efficiently. Second, we see no way to provide an induction elimination form for general recursive types without losing decidability of type checking. However, inductively defined recursive types, such as lists and trees, can be effectively de-structured by a finite unrolling, so we expect that we can add a simple, but effective facility for such types. Third, there are obvious connections between views and subsumption and we hope to explore this in order to support an efficient implementation of, for instance, record subsumption in the style of Ohori. Finally, we are in the process of building a prototype compiler that translates SML/NJ source to an intermediate language based on the concepts of $\lambda^{ML}$ and a back-end that supports unboxed, tagless representations. We hope to use this prototype empirically to explore tradeoffs in representation techniques such as method passing versus non-parametric operations.

## 8 Acknowledgements

We would like to thank Andrzej Filinski and Matthias Felleisen for their helpful discussions regarding this work and for their proofreading of earlier drafts of this document.

## A $\lambda^{ML}$ (with typerec) Static Semantics

### A.1 Constructor Formation

$$\frac{}{\Delta \vdash t : k} \quad \Delta(t) = k \qquad \frac{}{\Delta \vdash \dot{\mathsf{unit}} : \Omega} \qquad \frac{}{\Delta \vdash \dot{\rightarrow} : \Omega \times \Omega \rightarrow \Omega}$$

$$\frac{}{\Delta \vdash \dot{\times} : \Omega \times \Omega \rightarrow \Omega} \qquad \frac{\Delta \vdash u_1 : k_1 \qquad \Delta \vdash u_2 : k_2}{\Delta \vdash (u_1, u_2) : k_1 \times k_2}$$

$$\frac{\Delta \vdash u : k_1 \times k_2}{\Delta \vdash \dot{\pi}_i \; u : k_i} \quad (i = 1, 2)$$

$$\frac{\Delta, t : k_1 \vdash u : k_2}{\Delta \vdash \dot{\lambda}\, t : k_1 . u : k_1 \rightarrow k_2} \qquad \frac{\Delta \vdash u_1 : k_1 \rightarrow k_2 \qquad \Delta \vdash u_2 : k_1}{\Delta \vdash u_1 \; u_2 : k_2}$$

$$\frac{\begin{array}{c}\Delta \vdash u : \Omega \qquad \Delta \vdash u_{\mathsf{u}} : \Omega \\ \Delta \vdash u_{\times} : \Omega \rightarrow \Omega \rightarrow \Omega \\ \Delta \vdash u_{\rightarrow} : \Omega \rightarrow \Omega \rightarrow \Omega\end{array}}{\Delta \vdash \mathsf{TypeRec}(u \,;\, u_{\mathsf{u}} \,;\, u_{\times} \,;\, u_{\rightarrow}) : \Omega}$$

### A.2 Type Formation

$$\frac{}{\Delta \vdash \mathsf{unit} \;\; \mathsf{type}} \qquad \frac{\Delta \vdash u : \Omega}{\Delta \vdash T(u) \;\; \mathsf{type}} \qquad \frac{\Delta \vdash \sigma_1 \;\; \mathsf{type} \qquad \Delta \vdash \sigma_2 \;\; \mathsf{type}}{\Delta \vdash \sigma_1 \times \sigma_2 \;\; \mathsf{type}}$$

$$\frac{\Delta \vdash \sigma_1 \;\; \mathsf{type} \qquad \Delta \vdash \sigma_2 \;\; \mathsf{type}}{\Delta \vdash \sigma_1 \rightarrow \sigma_2 \;\; \mathsf{type}} \qquad \frac{\Delta, t : k \vdash \sigma \;\; \mathsf{type}}{\Delta \vdash \forall t : k . \sigma \;\; \mathsf{type}}$$

### A.3 Term Formation

$$\frac{\Delta \vdash \sigma \;\; \mathsf{type} \qquad \Gamma(x) = \sigma}{\Gamma \vdash_{\Delta} x : \sigma} \qquad \frac{}{\Gamma \vdash_{\Delta} \langle\rangle : \mathsf{unit}}$$

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

$$\frac{\Gamma \vdash_{\Delta, t:k} e : \sigma}{\Gamma \vdash_\Delta \Lambda\, t : k\, .e : \forall t : k.\sigma} \quad (t \notin Dom(\Delta))$$

$$\frac{\Gamma, x_1 : (\sigma_1 \to \sigma_2), x_2 : \sigma_1 \vdash_\Delta e : \sigma_2}{\Gamma \vdash_\Delta \mathsf{fix}\, x_1(x_2 : \sigma_1).e : \sigma_1 \to \sigma_2} \quad (x_1, x_2 \notin Dom(\Gamma))$$

$$\frac{\Gamma \vdash_\Delta e : \forall t : k.\sigma}{\Gamma \vdash_\Delta e[u] : [u/t]\sigma} \qquad \frac{\Gamma \vdash_\Delta e_1 : \sigma_1 \to \sigma_2 \qquad \Gamma \vdash_\Delta e_2 : \sigma_1}{\Gamma \vdash_\Delta e_1\, e_2 : \sigma_2}$$

$$\frac{\Gamma \vdash_\Delta e : \sigma_1 \qquad \Gamma \vdash_\Delta \sigma_1 \equiv \sigma_2 \;\; \mathsf{type}}{\Gamma \vdash_\Delta e : \sigma_2}$$

$$\frac{\begin{array}{c}\Delta \vdash \tau : \Omega \;\; \mathsf{type} \qquad \Delta \vdash \forall t : \Omega.\sigma \;\; \mathsf{type} \\ \Gamma \vdash_\Delta e_{\mathsf{u}} : \sigma(\dot{\mathsf{unit}}) \\ \Gamma \vdash_\Delta e_{\to} : \forall t_1 : \Omega.\forall t_2 : \Omega.\sigma(t_1) \to \sigma(t_2) \to \sigma(\dot{\to}(t_1, t_2)) \\ \Gamma \vdash_\Delta e_{\times} : \forall t_1 : \Omega.\forall t_2 : \Omega.\sigma(t_1) \to \sigma(t_2) \to \sigma(\dot{\times}(t_1, t_2))\end{array}}{\Gamma \vdash_\Delta \mathsf{typerec}(\tau;\; e_{\mathsf{u}};\; e_{\to};\; e_{\times}) : \sigma(\tau)}$$

## A.4 Constructor Equivalence

$$\frac{\Delta \vdash u_1 : k_1 \qquad \Delta \vdash u_2 : k_2}{\Delta \vdash \dot{\pi}_i\, (u_1, u_2) \equiv u_i : k_i} \quad (i = 1, 2)$$

$$\frac{\Delta \vdash u : k_1 \times k_2}{\Delta \vdash (\dot{\pi}_1\; u, \dot{\pi}_2\; u) \equiv u : k_1 \times k_2}$$

$$\frac{\Delta \vdash u_1 : k_1 \qquad \Delta, t : k_1 \vdash u_2 : k_2}{\Delta \vdash (\dot{\lambda}\, t : k_1.u_2)\; u_1 \equiv [u_1/t]u_2 : k_2}$$

$$\frac{\Delta \vdash u : k_1 \to k_2}{\Delta \vdash \dot{\lambda}\, t : k_1.(u\; t) \equiv u : k_1 \to k_2} \quad (t \notin Dom(\Delta))$$

$$\frac{\Delta \vdash \mathsf{TypeRec}(\dot{\mathsf{unit}};\ u_{\mathsf{u}};\ u_{\times};\ u_{\rightarrow}) : \Omega}{\Delta \vdash \mathsf{TypeRec}(\dot{\mathsf{unit}};\ u_{\mathsf{u}};\ u_{\times};\ u_{\rightarrow}) \equiv u_{\mathsf{u}} : \Omega}$$

$$\frac{\Delta \vdash \mathsf{TypeRec}(\dot{\times}(u_1, u_2);\ u_{\mathsf{u}};\ u_{\times};\ u_{\rightarrow}) : \Omega}{\Delta \vdash \mathsf{TypeRec}(\dot{\times}(u_1, u_2);\ u_{\mathsf{u}};\ u_{\times};\ u_{\rightarrow}) \equiv u_{\times}(\mathsf{TypeRec}(u_1;\ u_{\mathsf{u}};\ u_{\times};\ u_{\rightarrow}))(\mathsf{TypeRec}(u_2;\ u_{\mathsf{u}};\ u_{\times};\ u_{\rightarrow}))}$$

$$\frac{\Delta \vdash \mathsf{TypeRec}(\dot{\rightarrow}(u_1, u_2);\ u_{\mathsf{u}};\ u_{\times};\ u_{\rightarrow}) : \Omega}{\Delta \vdash \mathsf{TypeRec}(\dot{\rightarrow}(u_1, u_2);\ u_{\mathsf{u}};\ u_{\times};\ u_{\rightarrow}) \equiv u_{\rightarrow}(\mathsf{TypeRec}(u_1;\ u_{\mathsf{u}};\ u_{\times};\ u_{\rightarrow}))(\mathsf{TypeRec}(u_2;\ u_{\mathsf{u}};\ u_{\times};\ u_{\rightarrow}))}$$

### A.5 Type Equivalence

$$\frac{}{\Delta \vdash T(\dot{\mathsf{unit}}) \equiv \mathsf{unit}\ \ \mathsf{type}} \qquad \frac{\Delta \vdash \tau_1 \equiv \tau_2 : \Omega}{\Delta \vdash T(\tau_1) \equiv T(\tau_2)\ \ \mathsf{type}}$$

$$\frac{}{\Delta \vdash T(\dot{\times}(\tau_1, \tau_2)) \equiv T(\tau_1) \times T(\tau_2)\ \ \mathsf{type}}$$

$$\frac{}{\Delta \vdash T(\dot{\rightarrow}(\tau_1, \tau_2)) \equiv T(\tau_1) \rightarrow T(\tau_2)\ \ \mathsf{type}}$$

## B The Standard Translation

$$\begin{array}{rcl} |t| & = & t \\ |\mathsf{unit}| & = & \dot{\mathsf{unit}} \\ |\tau_1 \times \tau_2| & = & \dot{\times}(|\tau_1|, |\tau_2|) \\ |\tau_1 \rightarrow \tau_2| & = & \dot{\rightarrow}(|\tau_1|, |\tau_2|) \end{array}$$

$$|\forall t_1, \ldots, t_n.\tau| \quad = \quad \forall t_1 : \Omega, \ldots, t_n : \Omega.|\tau|$$

$$\frac{\Gamma(x) = \forall t_1, \ldots, t_n.\tau \qquad \Delta \vdash \tau_1\ \ \mathsf{type} \qquad \ldots \qquad \Delta \vdash \tau_n\ \ \mathsf{type}}{\Gamma \vdash_{\Delta} x : [\tau_1/t_1, \ldots, \tau_n/t_n]\tau \Rightarrow_S x[|\tau_1|_S] \ldots [|\tau_n|_S]}$$

$$\frac{}{\Gamma \vdash_{\Delta} \langle\rangle : \mathsf{unit} \Rightarrow \langle\rangle}$$

$$\frac{\Gamma \vdash_\Delta e_1 : \tau_1 \Rightarrow |e_1| \qquad \Gamma \vdash_\Delta e_2 : \tau_2 \Rightarrow |e_2|}{\Gamma \vdash_\Delta \langle e_1, e_2 \rangle : \tau_1 \times \tau_2 \Rightarrow \langle |e_1|, |e_2| \rangle}$$

$$\frac{\Gamma \vdash_\Delta e : \tau_1 \times \tau_2 \Rightarrow |e|}{\Gamma \vdash_\Delta \pi_i\, e : \tau_i \Rightarrow \pi_i\, |e|} \quad (i = 1, 2)$$

$$\frac{\Gamma, x_1 \mapsto \forall.(\tau_1 \to \tau_2), x_2 \mapsto \forall.\tau_1 \vdash_\Delta e : \tau_2 \Rightarrow |e|}{\Gamma \vdash_\Delta \mathsf{fix}\, x_1(x_2).e : \tau_1 \to \tau_2 \Rightarrow \mathsf{fix}\, x_1(x_2 : |\tau_1|).|e|} \quad (x_1, x_2 \notin \Gamma)$$

$$\frac{\begin{array}{c}\Gamma \vdash_\Delta e_1 : \tau_1 \to \tau_2 \Rightarrow |e_1| \\ \Gamma \vdash_\Delta e_2 : \tau_1 \Rightarrow |e_2|\end{array}}{\Gamma \vdash_\Delta e_1\, e_2 : \tau_2 \Rightarrow |e_1|\, |e_2|}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\Delta, t_1, \ldots, t_n} v : \tau' \Rightarrow |v| \\ \Gamma[x \mapsto \forall t_1, \ldots, t_n.\tau'] \vdash_\Delta e : \tau \Rightarrow |e|\end{array}}{\begin{array}{c}\Gamma \vdash_\Delta \mathsf{let}\ \ x = v\, \mathsf{in}\, e : \tau \Rightarrow \\ (\lambda\, x : \forall t_1 : \Omega, \ldots, t_n : \Omega.|\tau'|.|e|)(\Lambda\, t_1 : \Omega, \ldots, t_n : \Omega\, .|v|) \qquad (t_i \notin \Delta)\end{array}}$$

# C The Boxing Translation

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

$$|\tau|_B = \dot{\mathsf{box}}\, ||\tau||_B$$

$$|\forall t_1, \ldots, t_n.\tau|_B = \forall t_1 : \Omega, \ldots, t_n : \Omega.|\tau|_B$$

$$\frac{\begin{array}{c}\Gamma(x) = \forall t_1, \ldots, t_n.\tau \\ \Delta \vdash \tau_1\ \ \mathsf{type} \qquad \ldots \qquad \Delta \vdash \tau_n\ \ \mathsf{type}\end{array}}{\Gamma \vdash_\Delta x : [\tau_1/t_1, \ldots, \tau_n/t_n]\tau \Rightarrow_B x[||\tau_1||_B] \ldots [||\tau_n||_B]}$$

$$\frac{}{\Gamma \vdash_\Delta \langle\rangle : \mathsf{unit} \Rightarrow_B \mathsf{box}_{\mathsf{unit}}\, \langle\rangle}$$

$$\frac{\Gamma \vdash_\Delta e_1 : \tau_1 \Rightarrow_B |e_1| \qquad \Gamma \vdash_\Delta e_2 : \tau_2 \Rightarrow_B |e_2|}{\Gamma \vdash_\Delta \langle e_1, e_2 \rangle : \tau_1 \times \tau_2 \Rightarrow_B \mathsf{box}_{\mathsf{box}\times\mathsf{box}}(\mathsf{pair}_{\mathsf{box},\mathsf{box}}(|e_1|, |e_2|))}$$

$$\frac{\Gamma \vdash_\Delta e : \tau_1 \times \tau_2 \Rightarrow_B |e|}{\Gamma \vdash_\Delta \pi_i\, e : \tau_i \Rightarrow_B \mathsf{proj}_{i,\mathsf{box}\times\mathsf{box}}(\mathsf{unbox}_{\mathsf{box}}(|e|))} \quad (i = 1,2)$$

$$\frac{\Gamma, x_1 \mapsto \forall.(\tau_1 \to \tau_2), x_2 \mapsto \forall.\tau_1 \vdash_\Delta e : \tau_2 \Rightarrow_B |e|}{\Gamma \vdash_\Delta \mathsf{fix}\, x_1(x_2).e : \tau_1 \to \tau_2 \Rightarrow_B \mathsf{box}_{\mathsf{box}\to\mathsf{box}}\mathsf{fix}\, x_1(x_2 : |\tau_1|_B).|e|} \quad (x_1, x_2 \notin \Gamma)$$

$$\frac{\begin{array}{c}\Gamma \vdash_\Delta e_1 : \tau_1 \to \tau_2 \Rightarrow_B |e_1| \\ \Gamma \vdash_\Delta e_2 : \tau_1 \Rightarrow_B |e_2|\end{array}}{\Gamma \vdash_\Delta e_1\, e_2 : \tau_2 \Rightarrow_B (\mathsf{unbox}_{\mathsf{box}\to\mathsf{box}}\, |e_1|)\, |e_2|}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\Delta, t_1, \ldots, t_n} v : \tau' \Rightarrow_B |v| \\ \Gamma[x \mapsto \forall t_1, \ldots, t_n.\tau'] \vdash_\Delta e : \tau \Rightarrow_B |e|\end{array}}{\begin{array}{c}\Gamma \vdash_\Delta \mathsf{let}\ \ x = v\, \mathsf{in}\, e : \tau \Rightarrow_B \\ (\lambda\, x : \forall t_1 : \Omega, \ldots, t_n : \Omega.|\tau'|_B.|e|)(\Lambda\, t_1 : \Omega, \ldots, t_n : \Omega\, .|v|)\end{array} \qquad (t_i \notin \Delta)}$$

## References


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

[2] A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In M. Wirsing, editor, *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13, New York, Aug. 1991. Springer-Verlag. Volume 528 of *Lecture Notes in Computer Science*.

[3] L. Augustsson. Implementing Haskell overloading. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 65–73, Copenhagen, Denmark, June 1993.

[4] D. Clément, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ML. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 13–27, 1986.

[5] M. Felleisen, D. Friedman, E. Kohlbecker, and B. Duba. Reasoning with continuations. In *First Symposium on Logic in Computer Science*, pages 131–141. IEEE, June 1986.

[6] R. Harper and M. Lillibridge. Explicit polymorphism and CPS conversion. In *Conference Record of the 20th Annual ACM Symposium on Principles of Programming Languages*, Jan. 1993.

[7] X. Leroy. Unboxed objects and polymorphic typing. In *Proceedings of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 177–188, Jan. 1992.

[8] P. Mendler. *Recursive Definition in Type Theory*. PhD thesis, Cornell University, 1987.

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

[10] J. C. Mitchell and R. Harper. The essence of ML. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, pages 28–46, Jan. 1988.

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

[12] A. Ohori. A compilation method for ML-style polymorphic record calculi. In *Proceedings of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 154–165, Jan. 1992.

[13] C. Strachey. Fundamental concepts in programming languages. In *Lecture Notes for International Summer School in Computer Programming, Copenhagen*. Aug. 1967.

[14] A. Tolmach. Tag-free garbage collection using explicit type parameters. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, June 1994. To appear.

[15] P. Wadler and S. Blott. How to make ad hoc polymorphism less ad hoc. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60–76, Austin, Texas, 1989.

[16] A. Wright and M. Felleisen. A syntactic approach to type soundness. Technical Report TR91-160, Department of Computer Science, Rice University, Apr. 1991.

[17] A. K. Wright. Polymorphism for imperative languages without imperative types. Technical Report TR93-200, Department of Computer Science, Rice University, Feb. 1993.