Unboxed Values and Polymorphic Typing Revisited

Peter J. Thiemann
Wilhelm-Schickard-Institut, Universität Tübingen
Sand 13, D-72076 Tübingen, Germany

Abstract

Polymorphic languages require that values passed to polymorphic functions all have a representation of the same size. Any value whose natural representation does not fit this size must be boxed, *i.e.* represented by a pointer to a heap-allocated record. Major performance gains can be achieved by handling values in their natural, unboxed representation whenever possible. We show that not only monomorphic functions, but also many polymorphic functions can handle unboxed values if the function calling convention of the underlying implementation satisfies a mild assumption.

A representation type system is defined which describes boxing requirements. A type reconstruction algorithm is given which translates an untyped program into an explicitly typed program where all changes of representation are made explicit. Furthermore, we define an abstract machine which employs the required calling convention and is an adequate operational model for the representation type system.

Keywords boxing, polymorphic typing, unification

1 Introduction

A compiler for a polymorphic functional language like ML faces a dilemma: parametric polymorphism dictates that every value must be uniformly represented in a single word. In consequence, if the natural representation of a value does not fit into one word, it must be represented by a pointer to a heap-allocated record. Examples are floating-point numbers, long integers, tuples, etc. This indirect representation is called a *boxed representation*. However, it is often more efficient to keep a value in its natural, unboxed representation. There are two reasons for this. First, the primitive operations usually operate on the unboxed representation and, second, repeated boxing and unboxing operations take time and waste memory. For example, a floating-point number is most naturally represented in a floating-point register, since arithmetic operations (on modern architectures) require their arguments in registers.

Hence, it is advantageous to handle values in their unboxed representation wherever possible and to put values "in the box" only if they are passed to a polymorphic function. The work of Leroy [11] offers experimental data which shows that the gain of so doing can be substantial. He wraps polymorphic functions into coercions (functions which change the representation) such that types which instantiate generic type variables assume boxed values only. With his algorithm, monomorphic functions operate on unboxed data, while polymorphic functions require some amount of boxing in their arguments. We show that the boxing requirements for the arguments of polymorphic functions can be alleviated if the underlying execution model eliminates tail calls. Even more boxing constraints can be eliminated by assuming a specific calling convention. CPS-based compilers fulfill the latter assumption if they use a certain CPS transformation.

In our presentation, an order-sorted representation type language makes boxing explicit on the type level. Generic type variables come in two flavors, boxed or unrestricted, and can only be instantiated accordingly. A boxed type variable must be instantiated by boxed types, while unrestricted type variables can be instantiated by any type. Boxing constraints are imposed using a predicate on types which expresses that a value of that type is to be moved by the compiled code. Doing so imposes less boxing constraints than previous approaches.

A type reconstruction algorithm for representation types is presented for the functional core of the strict language ML. It translates its input into explicitly typed ML (Core-XML) with coercions that transform the representation of values.

Coercions on recursive datatypes such as lists are particularly costly and have to be avoided. In our approach lists can have unboxed elements in many circumstances.

The paper proceeds in Section 2 by exhibiting some examples where unboxed values are advantageous and where previous algorithms impose too many boxing constraints. Section 3 formalizes our notion of representation types and representation coercions and exhibits the corresponding type system. An algorithm which performs type reconstruction and inserts appropriate coercions is presented in Section 4. An operational semantics for which our representation types are adequate is presented in Section 6. Section 7 discusses constructed datatypes. Practical results are reported in Section 8. In Section 9 we discuss possible extensions, Section 10 reviews some related work, and Section 11 concludes.

2 Examples

In order to be able to distinguish boxed and unboxed values at the type level we talk about *representation types*, which may be boxed or unboxed. Informally, a (un-) boxed type assumes (un-) boxed values, only. See Sec. 3.2 for the formal definition.

As an example consider the polymorphic identity function `id` of type $\forall\beta\,.\,\beta \to \beta$, the function `abs: real` $\to$ `real`, and a number `x: real`. If `x` lives in a floating-point register and the function `abs` uses floating-point registers for parameter passing, the expression

`abs (id x)`

is not acceptable. Before passing `x` to `id` the value must be boxed. Consequently, the result of `id x` is a boxed `real` which must be unboxed before `abs` can be applied. Thus, whenever generic type variables are instantiated, the corresponding values must be boxed, as in the example where β is instantiated to (boxed) `real`. Wrapping `id` in suitable coercions yields:

$$\texttt{abs}\ (\texttt{unbox}_{\texttt{real}}\ (\texttt{id}\ (\texttt{box}_{\texttt{real}}\ \texttt{x})))$$

This is equivalent to what Leroy's algorithm does and it also forms the basic assumption of other work in this area, e.g. [16, 17, 8].

Without making any assumptions on the execution model it is indeed necessary to insist on binding generic type variables to boxed representation types only. However, there is room for improvement.

The K combinator (a projection function) provides a simple example for possible improvement:

$$\lambda x.\lambda y.x : \forall\alpha\,.\,\forall\beta\,.\,\alpha \to \beta \to \alpha$$

Even though the variable y has type β the actual argument need not be boxed since it is ignored. The code for K need not touch the value for y at all. It works correctly as long as it has access to x and can return its value. This "Don't-Care-Polymorphism" can be exploited without any assumptions. It applies to other projection functions as well.

The next observation is that some functions of type $\forall\beta \ldots \to \beta$ need not insist on β being boxed in the presence of proper tail recursion. Consider the function `apply`:

$$\lambda f.\lambda x.f\ x : \forall\alpha\,.\,\forall\beta\,.\,(\alpha \to \beta) \to \alpha \to \beta$$

The variable x has type α and is passed as an argument to f. Thus, x is required to be bound only to a boxed value. But the result of $f\ x$ of type β is only passed on as result. If the underlying implementation properly eliminates tail calls, the code for the above function ends with an indirect jump to f's code. Hence the result of type β is immediately passed from f to the caller of `apply`, both of which presumably know how to handle the specific instance of β. Functions like the combinators B and C fall into this class as well. This kind of optimization is possible under the following assumption.

Assumption 1 (Tail Call Elimination) *All function calls in tail position are implemented by the equivalent of a jump instruction.*

Tail call elimination can be captured by means of a transformation to continuation-passing style (CPS [5]). In continuation-passing style, functions never return, instead they pass their result to the continuation which is explicitly passed as an additional argument. Then a function can return an unboxed value whenever the continuation is able to deal with it. Tail calls are just function calls which reuse the continuation of their caller.

In a CPS transformation there is a choice on which argument comes first. In Fischer's original transformation [5] the continuation argument comes first, in the one apparently used by Appel [1] in the SML/NJ compiler the continuation argument comes last.

Let's consider with an example what happens if a compiler employs a CPS transformation which places the continuation argument first. The B combinator is

$$\lambda f.\lambda g.\lambda x.f\ (g\ x)$$

with ML-type $\forall\alpha\,.\,\forall\beta\,.\,\forall\gamma\,.\,(\beta\rightarrow\gamma)\rightarrow(\alpha\rightarrow\beta)\rightarrow(\alpha\rightarrow\gamma)$. From the previous example we know that γ need not be boxed, since it is only passed to the continuation. The CPS version[1] of B tells us that β may be unboxed, too:

$$\ldots\lambda k.\lambda x.g\ (f\ k)\ x.$$

Hence, β may also be unboxed since g passes its result to the continuation $(f\ k)$ which knows how to handle a β.

A direct-style implementation is also able to take advantage of the situation just mentioned. It must just obey the following additional assumption.

Assumption 2 (Parameter Passing) *All functions obey the same convention to pass parameters and to return results.*

Why doesn't it work if the continuation argument comes last? The answer is found by examining the result of applying such a CPS transformation to B:

$$\lambda k.g\ x\ (\lambda v.f\ v\ k)$$

Here the result of $g\ x$ must be bound to v since the continuation k must be passed after v. Hence, v cannot be passed implicitly.

3 Type System

3.1 Source Language

The source language is Core-ML, an implicitly typed lambda calculus, given by the grammar:

$$e ::= k \mid x \mid \lambda x.e \mid (e_1\ e_2) \mid \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$$

Here, k is taken from a set of constants and x from an infinite set of variables. The semantics of the language is applicative order (call-by-value) reduction augmented with δ-rules for functional constants.

The standard type language defines types τ and type schemes σ by

$$\tau ::= \alpha \mid b \mid \tau_1 \rightarrow \tau_2 \qquad \sigma ::= \tau \mid \forall\alpha\,.\,\sigma$$

where α is a type variable and b a base type. Typing judgements for ML-expressions have the form $A \vdash_{\mathrm{DM}} e : \sigma$ [4] and $A \vdash^{s}_{\mathrm{DM}} : \tau$ for the syntax-directed version [2]. Their inference rules are standard and therefore omitted.

[1] Using the transformation of Sabry and Felleisen [18].

3.2 Representation Types

The language of *representation types* distinguishes boxed and unboxed types. It is conveniently modelled as the set of order-sorted terms [12] over the four element upper semi-lattice $\mathcal{B} = (\{U, B, H, \top\}, \leq)$ shown in the Hasse-diagram below.

$$\begin{tikzcd}[column sep=small]
 & \top \arrow[d, dash] & \\
 & H \arrow[dl, dash] \arrow[dr, dash] & \\
U & & B
\end{tikzcd}$$

The elements U and B denote the sorts of unboxed and boxed types, respectively, the element H denotes the sort of types whose values must be moved or otherwise processed by the code, and $\top$ denotes the sort of all representation types. The distinction between H and $\top$ allows us to describe dead arguments (Don't-Care-Polymorphism) by the sort $\top$ in contrast to the sort H which indicates a live argument. In the following let Y and Z range over the elements of $\mathcal{B}$.

Each type constructor χ with k parameters has an associated *sorting* $\Sigma(\chi) = (Y_1 \ldots Y_k, Z)$ which indicates that it accepts type parameters of sorts Y_i and yields a type of sort Z. The special type constructor $\langle\_\rangle$ maps unboxed types to boxed types, hence its sorting is (U, B). All base types have sorting (ε, U) and the sorting of $\rightarrow$ is $(\top H, U)$ since a function may ignore its argument but should be able to handle its result.

Let Γ be a set of *sort assumptions* $\alpha{:}Z$ on type variables. It constrains the types which may be substituted for α to those of sort Z. We write $\Gamma\{\alpha{:}Y\}$ to denote A augmented by the assumption $\alpha{:}Y$ where α does not occur in Γ. We use $\theta[\rho/\alpha]$ to denote θ with all free occurrences of α replaced by ρ (substitution).

The formation and sorting rules for representation types are given below.

$$\begin{array}{ll}
\text{(S-VAR)} & \Gamma\{\alpha{:}Y\} \Vdash \alpha{:}Y \\[4pt]
\text{(S-CRC)} & \dfrac{\Gamma \Vdash \rho{:}Y \qquad Y \leq Z}{\Gamma \Vdash \rho{:}Z} \\[8pt]
\text{(S-APP)} & \dfrac{\Gamma \Vdash \rho_i{:}Y_i{}^{1 \leq i \leq k} \qquad \Sigma(\chi) = (Y_1 \ldots Y_k, Z)}{\Gamma \Vdash \chi(\rho_1, \ldots, \rho_k){:}Z} \\[8pt]
\text{(S-GEN)} & \dfrac{\Gamma\{\alpha{:}Y\} \Vdash \rho{:}Z}{\Gamma \Vdash \forall\alpha{:}Y \,.\, \rho{:}Z} \\[8pt]
\text{(S-SPC)} & \dfrac{\Gamma \Vdash \forall\alpha{:}Y \,.\, \rho{:}Z \qquad \Gamma \Vdash \rho'{:}Y}{\Gamma \Vdash \rho[\rho'/\alpha]{:}Z}
\end{array}$$

The judgement $\Gamma \Vdash \theta{:}Y$ is read "under sort assumptions Γ the representation type scheme θ has sort Y". The rule S-VAR extracts a type variable and its sorting from the sort assumptions. The rule S-CRC raises the sort of a representation type in the sort semi-lattice $\mathcal{B}$. The rule S-APP handles the application of a type constructor χ of sorting $\Sigma(\chi)$ to argument types ρ_i, where the sorts of the arguments must be identical to the sorts required by $\Sigma(\chi)$. The last rules deal with generalization (S-GEN) and instantiation (S-SPC) of representation types with quantification. S-GEN introduces an all-quantified type variable of sort Y. S-SPC makes sure that instantiation of all-quantified type variables respects their sort. The type language thus defined is richer than strictly necessary here. Only representation type schemes will be used where all quantification occurs at the top level. We will use the following conventions:

ρ	representation types (no quantification)
υ	unboxed types where $\Gamma \Vdash \upsilon{:}U$ (no quantification)
θ	representation type schemes, possibly with top level quantification

In other notions of representation types it is only possible to abstract boxed type variables, that is $\forall\alpha{:}B \,.\, \rho$. In contrast, our representation type language encompasses abstraction over unboxed type variables, $\alpha{:}U$, and unconstrained type variables, $\alpha{:}H$ and $\alpha{:}\top$. Abstraction over unboxed type variables does not make sense, but it is allowed (and makes sense) to abstract unconstrained type variables in addition to boxed type variables, as we will show.

Code can be generated to process a value if its representation type is sufficiently known or boxed. A completely known (monomorphic) type can certainly be processed. The predicate P (P for *passed*) imposes the necessary constraints. It describes those types which have a "boxed frontier". Any value whose type is a base type (like integer, floating-point, character) or a boxed type can be passed around, since its representation is completely determined. Values of function type can be passed, as well. The following rules define P:

$$\begin{array}{lll}
\text{(P-BASE)} & P(\Gamma, \chi) & \text{where } \Sigma(\chi) = (\varepsilon, Z) \\
\text{(P-BOXED)} & P(\Gamma, \rho) & \text{where } \Gamma \Vdash \rho{:}B \\
\text{(P-FUN)} & P(\Gamma, \rho' \to \rho) & \\
\text{(P-GEN)} & \dfrac{P(\Gamma\{\alpha{:}Z\}, \theta)}{P(\Gamma, \forall \alpha{:}Z \,.\, \theta)} &
\end{array}$$

3.3 Coercions

The presentation of coercions follows Henglein and Jørgensen [8]. A coercion $c\colon \rho \rightsquigarrow \rho'$ transforms values of representation type ρ to values of representation type ρ'. Such a c exists iff the underlying standard types are equal, $|\rho| = |\rho'|$. $|\theta|$ is the standard type scheme obtained from θ by erasing all box constructors $\langle \_ \rangle$ and all sort constraints from quantified variables. For every unboxed type v there are two primitive coercions, $\mathtt{box}_v\colon v \rightsquigarrow \langle v \rangle$ and $\mathtt{unbox}_v\colon \langle v \rangle \rightsquigarrow v$. There is an identity coercion ι_ρ for every type and coercions c and c' can be composed to $c; c'$ which means "apply c first, then c'". For every type constructor there is a coercion induced from coercions on its arguments.

The formation rules for coercion terms are given below. In an abuse of notation we will not distinguish between a coercion term and the coercion it denotes whenever this does not cause confusion.

$$\iota_\rho\colon \rho \rightsquigarrow \rho \qquad \mathtt{box}_v\colon v \rightsquigarrow \langle v \rangle \qquad \mathtt{unbox}_v\colon \langle v \rangle \rightsquigarrow v$$

$$\frac{c_1\colon \rho_1 \rightsquigarrow \rho_1' \qquad c_2\colon \rho_2 \rightsquigarrow \rho_2'}{c_1 \to c_2\colon \rho_1' \to \rho_2 \rightsquigarrow \rho_1 \to \rho_2'}$$

$$\frac{c'\colon \rho \rightsquigarrow \rho' \qquad c''\colon \rho' \rightsquigarrow \rho''}{c'; c''\colon \rho \rightsquigarrow \rho''}$$

The following equality rules hold for coercions.

$$\begin{gathered}
(c_1; c_2); c_3 = c_1; (c_2; c_3) \\
c; \iota = \iota; c = c \\
(c_1 \to c_2); (c_1' \to c_2') = (c_1'; c_1) \to (c_2; c_2') \\
\iota_\rho \to \iota_{\rho'} = \iota_{\rho \to \rho'} \\
\mathtt{box}_v; \mathtt{unbox}_v = \iota_v \\
\mathtt{unbox}_v; \mathtt{box}_v = \iota_{\langle v \rangle}
\end{gathered}$$

Whenever we say $c = c'$ in the following we mean that it is a consequence of the above equality rules closed under congruence.

For every coercion $c : v \rightsquigarrow v'$ on unboxed types there is a coercion $\langle c \rangle = \mathtt{unbox}_v; c; \mathtt{box}_v : \langle v \rangle \rightsquigarrow \langle v' \rangle$ which changes the representation "in the box". The inversion operator transforms a coercion $c\colon \rho \rightsquigarrow \rho'$ into $c^{-1}\colon \rho' \rightsquigarrow \rho$ such that $c; c^{-1} = \iota$.

3.4 Core-XML

The subject language for representation type inference is Core-XML. It is an explicitly typed second-order lambda calculus extended with coercion applications.

$$\begin{aligned}
e ::= {} & x \mid \lambda x : \rho.e \mid (e\ e') \mid \mathtt{let}\ x : \theta = e\ \mathtt{in}\ e' \mid \\
& \Lambda \alpha{:}Z \,.\, e \mid e\{\rho\} \mid \langle c \rangle e
\end{aligned}$$

An expression is either a variable, a typed abstraction, an application, a `let`-expression, a sorted type abstraction, a type application, or a coercion application. The intended semantics is call-by-value reduction. No computation is performed for type abstraction and type application.

3.5 Inference Rules for Representation Typing

Let $FV(\theta)$ yield the set of type variables which occur free in θ. A type assumption A is a set with elements $x:\theta$ and $FV(A)$ is the union of all $FV(\theta)$ where $x:\theta \in A$.

The inference rules below define the judgement $\Gamma, A \vdash e : \theta$ which states that under sort assumptions Γ (on type variables) and type assumptions A (on program variables) the Core-XML expression e has representation type θ.

$$\text{(R-VAR)}\quad \frac{P(\Gamma,\theta)}{\Gamma, A\{x:\theta\} \vdash x:\theta}$$

$$\text{(R-ABS)}\quad \frac{\Gamma, A\{x:\rho'\} \vdash e:\rho}{\Gamma, A \vdash \lambda x:\rho'.e : \rho' \to \rho}$$

$$\text{(R-APP)}\quad \frac{\Gamma, A \vdash e:\rho' \to \rho \qquad \Gamma, A \vdash e':\rho'}{\Gamma, A \vdash e\ e' : \rho}$$

$$\text{(R-GEN)}\quad \frac{\Gamma\{\alpha{:}Z\}, A \vdash e:\theta}{\Gamma, A \vdash \Lambda\alpha{:}Z\,.\,e : \forall\alpha{:}Z\,.\,\theta}\quad {\scriptstyle \alpha\in FV(\theta)\backslash FV(A)}$$

$$\text{(R-SPC)}\quad \frac{\Gamma, A \vdash e:\forall\alpha{:}Z\,.\,\theta \qquad \Gamma \Vdash \rho{:}Z}{\Gamma, A \vdash e\{\rho\} : \theta[\rho/\alpha]}$$

$$\text{(R-LET)}\quad \frac{\Gamma, A \vdash e:\theta \qquad \Gamma, A\{x:\theta\} \vdash e':\rho'}{\Gamma, A \vdash \texttt{let}\ x:\theta = e\ \texttt{in}\ e' : \rho'}$$

$$\text{(R-CRC)}\quad \frac{\Gamma, A \vdash e:\rho \qquad c:\rho \leadsto \rho'}{\Gamma, A \vdash \langle c\rangle e : \rho'}$$

Notice that writing $\rho' \to \rho$ in rules R-ABS and R-APP implies that the type is well-formed according to the underlying sorting Σ. The formal statement $\Gamma \Vdash \rho' \to \rho{:}Z$ has been omitted for readability.

The variable rule R-VAR enforces the type of a variable to obey P since the value bound to the variable is actually used. This is the way to state the liveness of a variable.

The rule R-ABS for abstraction looks just like the standard rule. It does not force the argument type ρ' to obey P. If x does not occur in the body expression e the type $\alpha{:}\top$ can be assigned to x.

The rule for function application R-APP does not put any requirements on the argument. Therefore $\vdash$ presupposes the parameter passing assumption.

The rules for generalization and instantiation of types (R-GEN, R-SPC) must enforce the sort constraints. The `let`-rule (R-LET) is just the standard rule.

The rule (R-CRC) types the application $\langle c\rangle e$ of a coercion $c:\rho \leadsto \rho'$ to an expression e of type ρ.

Examples We review the typing of Core-XML versions of the introductory examples. For $K = \lambda x.\lambda y.x$ we derive:

$$\begin{aligned}\emptyset,\emptyset \vdash\ & \Lambda\alpha{:}B\,.\,\Lambda\beta{:}\top\,.\,\lambda x:\alpha.\lambda y:\beta.x : \\ & \forall\alpha{:}B\,.\,\forall\beta{:}\top\,.\,\alpha \to \beta \to \alpha\end{aligned}$$

The first argument of K must be boxed and the second argument is unconstrained. In fact, its sort $\top$ indicates that it is unused.

$$\begin{aligned}\emptyset,\emptyset \vdash\ & \Lambda\alpha{:}B\,.\,\Lambda\beta{:}H\,.\,\lambda f:\alpha \to \beta.\lambda x:\alpha.f\ x : \\ & \forall\alpha{:}B\,.\,\forall\beta{:}H\,.\,(\alpha \to \beta) \to \alpha \to \beta\end{aligned}$$

This is the expected typing for $\texttt{apply} = \lambda f.\lambda x.f\ x$ the function argument f must accept a boxed argument and can return anything, the argument x for the function must be boxed accordingly, and the result might be unboxed.

$$\begin{aligned}\emptyset,\emptyset \vdash & \Lambda\alpha{:}B\,.\,\Lambda\beta{:}H\,.\,\Lambda\gamma{:}H\,.\,\lambda f:\beta \to \gamma.\lambda g:\alpha \to \beta.\lambda x:\alpha.f\ (g\ x) : \\ & \forall\alpha{:}B\,.\,\forall\beta{:}H\,.\,\forall\gamma{:}H\,.\,(\beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma.\end{aligned}$$

This result is expected for the B combinator with the parameter passing assumption: everything but the first argument passed to g (of type α) may be unboxed.

The last example shows that there are cases where our type inference system imposes unnecessary constraints. Consider the identity function $\lambda x.x$. Its implementation can be simply a return instruction under

$$\text{T-VAR} \quad \frac{P(\Gamma, \overline{\forall\alpha_m{:}Z_m} \,.\, \rho) \qquad \Gamma \Vdash \rho_i{:}Z_i}{\Gamma, A\{x : \overline{\forall\alpha_m{:}Z_m} \,.\, \rho\} \vdash x \rightsquigarrow x\{\overline{\rho_m}\} : \rho[\overline{\rho_m/\alpha_m}]}$$

$$\text{T-ABS} \quad \frac{\Gamma, A\{x : \rho'\} \vdash e \rightsquigarrow \tilde{e} : \rho}{\Gamma, A \vdash \lambda x.e \rightsquigarrow \lambda x : \rho'.\tilde{e} : \rho' \to \rho}$$

$$\text{T-APP} \quad \frac{\Gamma, A \vdash e_1 \rightsquigarrow \tilde{e}_1 : \rho_1 \qquad \Gamma, A \vdash e_2 \rightsquigarrow \tilde{e}_2 : \rho_2 \qquad c : \rho_1 \rightsquigarrow \rho_2 \to \rho}{\Gamma, A \vdash e_1\ e_2 \rightsquigarrow (\langle c\rangle \tilde{e}_1)\ \tilde{e}_2 : \rho}$$

$$\text{T-LET} \quad \frac{\Gamma, A \vdash e_1 \rightsquigarrow \tilde{e}_1 : \rho_1 \qquad \theta = \overline{\forall\alpha_m{:}\Gamma(\alpha_m)} \,.\, \rho_1 \qquad \Gamma, A\{x : \theta\} \vdash e_2 \rightsquigarrow \tilde{e}_2 : \rho_2}{\Gamma, A \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \rightsquigarrow \texttt{let}\ x : \theta = \overline{\Lambda\alpha_m{:}\Gamma(\alpha_m)} \,.\, \tilde{e}_1\ \texttt{in}\ \tilde{e}_2}\ \{\overline{\alpha_m}\} = FV(\rho) \setminus FV(A)$$

Figure 1: Translation rules from Core-ML to Core-XML

the parameter passing assumption. However, the inference rule R-VAR requires the type of x to fulfill P, therefore x must be a boxed value (but see Sec. 9 for improvement).

$$\begin{array}{l} (\text{R-VAR}) \\ (\text{R-ABS}) \\ (\text{R-GEN}) \end{array} \quad \cfrac{\cfrac{\cfrac{P(\{\alpha{:}B\}, \alpha)}{\{\alpha{:}B\}, \{x : \alpha\} \vdash x : \alpha}}{\{\alpha{:}B\}, \emptyset \vdash \lambda x : \alpha.x : \alpha \to \alpha}}{\emptyset, \emptyset \vdash \Lambda\alpha{:}B \,.\, \lambda x : \alpha.x : \forall\alpha{:}B \,.\, \alpha \to \alpha}$$

Variation 1 In order to make the type system exactly reflect Leroy's boxing requirements it is only necessary to insist in passability for any function argument. The modified system defines judgements $\Gamma, A \vdash_1 e : \rho$ and has all the inference rules from above except that the rule R-ABS is replaced by R-ABS' below.

$$(\text{R-ABS'}) \quad \frac{\Gamma, A\{x : \rho'\} \vdash_1 e : \rho \qquad P(\Gamma, \rho')}{\Gamma, A \vdash_1 \lambda x : \rho'.e : \rho' \to \rho}$$

Variation 2 If the implementation only guarantees tail call elimination then the code for the function application must be able to pass the argument value to the function. That means the argument type must obey predicate P. Only the rule R-APP need be changed (wrt. $\vdash$) to implement the judgement $\Gamma, A \vdash_2 e : \rho$:

$$(\text{R-APP'}) \quad \frac{\Gamma, A \vdash_2 e : \rho' \to \rho \qquad \Gamma, A \vdash_2 e' : \rho' \qquad P(\Gamma, \rho')}{\Gamma, A \vdash_2 e\ e' : \rho}$$

3.6 Translation

Building on the representation type system we give a syntax directed translation of ML-expressions into Core-XML with explicit coercions. The translation is defined by a set of inference rules which derive judgements of the form $\Gamma, A \vdash e \rightsquigarrow \tilde{e} : \rho$. The meaning is "under sort assumption Γ and type assumption A expression e translates to the Core-XML expression $\tilde{e}$ with representation type ρ". The rules are defined in Fig. 1. In the rules and henceforth we use abbreviations like $\overline{\forall\alpha_m{:}Z_m}$ for $\forall\alpha_1{:}Z_1 \ldots \forall\alpha_m{:}Z_m$. Translations for the tail recursion assumption or without any assumptions are obtained by similar modifications as for the respective typing judgements.

The following lemmas provide the connections between the translation, Core-XML typing, and standard ML typing. They are all proved by induction on derivation trees.

Lemma 1 *If* $\Gamma, A \vdash e \rightsquigarrow \tilde{e} : \theta$ *then* $\Gamma, A \vdash \tilde{e} : \theta$ *is derivable.*

The erasure $|e|$ of a Core-XML expression is the Core-ML expression obtained by dropping all type applications, type abstractions, type annotations, and coercions.

Lemma 2 *If* $\Gamma, A \vdash e \rightsquigarrow \tilde{e} : \theta$ *then* $e = |\tilde{e}|$.

Lemma 3 *If* $\Gamma, A \vdash e : \theta$ *then* $|A| \vdash_{DM} |e| : |\theta|$.

The connection between $\vdash_{\mathrm{DM}}$ and its syntax-directed version $\vdash^{s}_{\mathrm{DM}}$ is well-known [2, Theorem 2.1]:

Theorem 1 *Let A be a standard type assumption, e a Core-ML expression, and* gen *the standard generalization.*

1. *If $A \vdash^{s}_{DM} e : \tau$ then $A \vdash_{DM} e : \tau$.*
2. *If $A \vdash_{DM} e : \sigma$ then there is a type τ such that $A \vdash^{s}_{DM} e : \tau$ and σ is a generic instance of* $\mathrm{gen}(A, \tau)$.

In order to state the connection between the standard typing of an expression and its translation, we need the following definition.

Definition 1 *A sort assumption Γ is* consistent *with type assumption A if $\Gamma \Vdash \theta{:}\top$ for each $x : \theta \in A$.*

Lemma 4 *For any representation type assumption A, expression e, and sort assumption Γ (consistent with A) where $P(\Gamma, \theta)$ holds for each $x : \theta \in A$ with x free in e it holds that if $|A| \vdash^{s}_{DM} e : \tau$ then there are some ρ and $\tilde{e}$ such that $\Gamma, A \vdash e \rightsquigarrow \tilde{e} : \rho$ can be derived where $\tau = |\rho|$ and $e = |\tilde{e}|$.*

4 Type Reconstruction

On the basis of the type system introduced above we present an algorithm which performs type reconstruction for representation types and inserts the necessary coercions into the inspected term.

4.1 Unification

The unification algorithm is a sorted unification algorithm which additionally computes the necessary coercion along with the unifying substitution. Given ρ and ρ' it solves the problem "$\exists\vartheta.\exists c.c : \vartheta\rho \rightsquigarrow \vartheta\rho'$?" by computing a most general such ϑ (or fails if it does not exist) along with the coercion c which is uniquely determined by ϑ, ρ, and ρ'. The algorithm is presented as a rewriting system on an *equation system with coercions*, *i.e.* a set E of pairs of the form $(\rho \overset{?}{\rightsquigarrow} \rho', K)$ where $\rho \overset{?}{\rightsquigarrow} \rho'$ is an equation to be solved and K is a *coercion context*. Beware that the orientation of $\rho \overset{?}{\rightsquigarrow} \rho'$ is important.

Intuitively, a coercion context is a coercion term with a hole, written as $[\,]$. If we insert a coercion term c into the hole we obtain the coercion term $K[c]$. We can also insert another context K' into the hole to obtain the new coercion context $K[K']$. The following grammar defines coercion contexts K where c is an arbitrary coercion term:

$$K ::= [\,] \mid K \to c \mid c \to K \mid K; c \mid c; K \mid \langle K \rangle \mid (K)^{-1}$$

We will also make use of coercion variables γ, γ_i in coercion terms in the following algorithm. They are used like logical variables, hence we need fresh coercion variables whenever such a variable occurs on the right hand side of a rule below.

Recall for the following definition that ρ denotes arbitrary representation types and υ unboxed types. Let π denote boxed types where $\Gamma \Vdash \pi{:}B$ and let $LS(\Gamma, \rho) = \min\{Z \mid \Gamma \Vdash \rho{:}Z\}$ be the least sort of ρ with respect to sort assumption Γ.

Definition 2 *Let Γ be a sort assumption and γ, γ_i be some coercion variables. The relation $\xrightarrow{\Gamma}$ on equation systems with coercions is defined by the rules in Fig. 2.*

The first rules (1)–(2) deal with removing type constructors. The function type constructor $\to$ is contravariant in its first argument, hence the equation for ρ_1 is reversed. The contexts are propagated such that if $c_1 : \rho_1' \rightsquigarrow \rho_1$ and $c_2 : \rho_2 \rightsquigarrow \rho_2'$ are coercions then the coercion terms $c_1 \to \gamma_2$ and $\gamma_2 \to c_2$ can be unified to the coercion $c_1 \to c_2$ of type $\rho_1 \to \rho_2 \rightsquigarrow \rho_1' \to \rho_2'$. The coercion under the boxing constructor need not be reversed.

Reversing an equation corresponds to reversing the respective hole in the context. Otherwise rule (3) is identical to the usual unification rule, extended by exchanging two variables if the sorting of the variable on the left hand side is smaller than on the right hand side of $\overset{?}{\rightsquigarrow}$.

$$
\begin{array}{rcll}
\{(\rho_1 \to \rho_2 \stackrel{\cdot}{\rightsquigarrow} \rho_1' \to \rho_2', K)\} \cup E & \xrightarrow{\Gamma} & \{(\rho_1' \stackrel{\cdot}{\rightsquigarrow} \rho_1, K[[\,] \to \gamma_2]), (\rho_2 \stackrel{\cdot}{\rightsquigarrow} \rho_2', K[\gamma_1 \to [\,]])\} \cup E & \text{where } \gamma_1, \gamma_2 \text{ are fresh} \quad (1) \\
\{(\langle v \rangle \stackrel{\cdot}{\rightsquigarrow} \langle v' \rangle, K)\} \cup E & \xrightarrow{\Gamma} & \{(v \stackrel{\cdot}{\rightsquigarrow} v', K[\langle [\,] \rangle])\} \cup E & (2) \\
\{(\rho \stackrel{\cdot}{\rightsquigarrow} \alpha, K)\} \cup E & \xrightarrow{\Gamma} & \{(\alpha \stackrel{\cdot}{\rightsquigarrow} \rho, K[([\,])^{-1}])\} \cup E \quad \text{if } \rho \notin V \vee (\rho = \alpha' \wedge \Gamma(\alpha') < \Gamma(\alpha)) & (3) \\
\{(v \stackrel{\cdot}{\rightsquigarrow} \pi, K)\} \cup E & \xrightarrow{\Gamma} & \{(\langle v \rangle \stackrel{\cdot}{\rightsquigarrow} \pi, K[\mathtt{box}; [\,]])\} \cup E & (4) \\
\{(\pi \stackrel{\cdot}{\rightsquigarrow} v, K)\} \cup E & \xrightarrow{\Gamma} & \{(\pi \stackrel{\cdot}{\rightsquigarrow} \langle v \rangle, K[[\,]; \mathtt{unbox}])\} \cup E & (5) \\
\{(\alpha \stackrel{\cdot}{\rightsquigarrow} \rho, K)\} \cup E & \xrightarrow{\Gamma} & \{(\alpha \stackrel{\cdot}{\rightsquigarrow} \rho, K)\} \cup E[\rho/\alpha] \quad \text{if } \alpha \in FV(E) \setminus FV(\rho) \wedge LS(\Gamma, \rho) \leq \Gamma(\alpha) & (6)
\end{array}
$$

Figure 2: Simplification rules for equation systems with coercions

Rules (4) and (5) deal with change of representation. In both cases one side of the equation is boxed. This is recorded in the context by prepending a `box` (appending an `unbox`) operation.

The substitution rule (6) is identical to the usual unification rule. It insists on sort decreasing substitutions. Contexts are not changed by applying a substitution.

There are no rules to remove trivial equations $\alpha \stackrel{\cdot}{\rightsquigarrow} \alpha$. They cannot be dropped since their contexts may contain vital coercion information.

Lemma 5 *Rewriting with* $\xrightarrow{\Gamma}$ *is sound and complete.*

Proof: By analysis of the simplification rules.

Lemma 6 $\xrightarrow{\Gamma}$ *is terminating.*

The proof uses standard techniques. Now we turn to reading off the resulting coercion.

Proposition 1 *Let* $CC(E) = \{K[\iota] \mid (\rho \stackrel{\cdot}{\rightsquigarrow} \rho', K) \in E\}$ *be a set of coercion terms and* $E \xrightarrow{\Gamma} E'$.
If $CC(E)$ *is syntactically unifiable then so is* $CC(E')$.

Proposition 2 *Let* $E \xrightarrow{\Gamma}{}^{*} E'$ *and* ϑ *a solution of* E *and* E'.
For all $(\rho \stackrel{\cdot}{\rightsquigarrow} \rho', K) \in E$ *there is an* $n > 0$ *and* $(\tilde{\rho}_i \stackrel{\cdot}{\rightsquigarrow} \tilde{\rho}_i', K_i) \in E'$ *for* $1 \leq i \leq n$ *with* $c_i : \vartheta\tilde{\rho}_i \rightsquigarrow \vartheta\tilde{\rho}_i'$ *and* $K_i = K[K_i']$ *such that* $c_0 : \vartheta\rho \rightsquigarrow \vartheta\rho'$ *where* c_0 *is the coercion denoted by the syntactic most general unifier of* $\{K_i'[c_i] \mid 1 \leq i \leq n\}$.

The unification algorithm $\mathrm{unify}^{\Gamma}(\rho, \rho')$ starts with $\{(\rho \stackrel{\cdot}{\rightsquigarrow} \rho', [\,])\}$ and applies $\xrightarrow{\Gamma}$ until it is no longer applicable. If the resulting system E' is in solved form the unification algorithm yields the pair (ϑ, c) where ϑ is the most general solution and c is the coercion denoted by the syntactic most general unifier of $CC(E')$ (a coercion term).

Notice, that order-sorted unification need not be unitary (not even finitary) in general and that the general problem is NP-complete [19]. The reason is that symbols may be overloaded, *i.e.* they may have more than one sorting. Since we do not use that feature, our unification problem can be solved in quasi-linear time (see [12]).

4.2 Reconstruction Algorithm

The requirement that a representation type ρ fulfills the predicate P in some environment Γ is not the same as requiring ρ to be a boxed type. The predicate P is also fulfilled if ρ is a base type or a function type. Only the generalization of a type variable which must fulfill P forces it to become boxed.

The algorithm keeps track of P by attaching a tag to each type variable. The tag is either $\{P\}$ or $\emptyset$ denoting the requirement to fulfill P or no requirement at all. In the generalization step the tags on the abstracted type variables are translated into boxing requirements: $\emptyset$ maps to $\top$ and $\{P\}$ maps to B. The unification rule (6) is enhanced to propagate the tag according to P.

In the variable case (T-VAR) the type reconstruction algorithm assigns the tag $\{P\}$ to the top of the type introduced. To enforce the tail recursion assumption, the argument to a function is assigned the tag $\{P\}$.

The latter assignment is achieved by passing the required tag as another parameter similar to an inherited attribute.

The main differences to Milner's algorithm $\mathcal{W}$ [13] are the different unification algorithm, the propagation of tags, the modified generalization mechanism, and the transformed expression as an additional result. The type reconstruction and transformation function is called *TRT*. The different versions $TRT^{\emptyset}$, and $TRT^{\{P\}}$ require their result to satisfy the respective predicate. As an example, we demonstrate the case for function application. The full algorithm appears in a forthcoming technical report.

$$
\begin{array}{lll}
\multicolumn{3}{l}{TRT^X(\Gamma, A, e\ e')} \\
= \texttt{let} & (\vartheta, \rho, \tilde{e}) & = TRT^{\emptyset}(\Gamma, A, e) \\
 & (\vartheta', \rho', \tilde{e}') & = TRT^{\{P\}}(\Gamma, \vartheta(A), e') \\
 & \alpha & = \text{fresh type variable of sort } \top \\
 & (\zeta, c) & = \text{unify}^{\Gamma}(\vartheta'(\rho), \rho' \to \alpha^X) \\
\texttt{in} & \multicolumn{2}{l}{(\zeta \circ \vartheta' \circ \vartheta, \zeta(\alpha), ((\langle c\rangle \zeta(\tilde{e}))\ \zeta(\vartheta'(\tilde{e}'))))}
\end{array}
$$

The application $\zeta(\tilde{e})$ of a substitution ζ to a Core-XML expression $\tilde{e}$ denotes the core-XML expression which is obtained from $\tilde{e}'$ by applying ζ to all type annotations and to the types of all coercions.

Syntactical correctness is stated as follows:

Proposition 3 *If* $TRT^X(\Gamma, A, e) = (\vartheta, \rho, \tilde{e})$ *then* $\Gamma, \vartheta(A) \vdash e \rightsquigarrow \tilde{e} : \rho$ *and* $P(\Gamma, \rho)$ *if* $P \in X$.

The corresponding completeness statement is as follows.

Proposition 4 *Suppose* $\Gamma, A \vdash e \rightsquigarrow \tilde{e} : \rho$ *holds. Then* $TRT^X(\Gamma, A, e)$ *succeeds and yields* $(\vartheta, \rho', \tilde{e}')$ *such that there is a substitution* δ *where* $\rho = \delta(\rho')$ *and* $\tilde{e} = \delta(\tilde{e}')$.

5 Simplification of Coercions

In the translated term coercions only occur in subterms of the form $((\langle c\rangle e_1)\ e_2)$ due to rule T-APP. Furthermore c is the by-product of unifying some $\vartheta'(\rho)$ with $\rho' \to \alpha$ where α has sort $\top$ and does not occur in $\vartheta'(\rho)$ and ρ'. Inspection of the unification rules shows that either $c = c_1 \to \iota$ (since there is no constraint on the result whatsoever, viz. α of sort $\top$ in the algorithm *TRT*) or $c = \texttt{unbox}_{\rho' \to \rho''}; c_1 \to \iota$.

Only the primitive coercions `box` and `unbox` are directly implementable on a machine. Functional coercions first have to be expressed in terms of the primitive ones by application of the following rules [8]:

$$\langle c_1; c_2\rangle e = \langle c_2\rangle\langle c_1\rangle e \tag{7}$$

$$((\langle c_2 \to c_1\rangle e_1)\ e_2) = \langle c_1\rangle(e_1\ (\langle c_2\rangle e_2)) \tag{8}$$

$$\langle \iota\rangle e = e \tag{9}$$

Functional coercion can be particularly costly since sometimes an η-redex must be introduced before equation (8) can be applied.

$$\langle c_1 \to c_2\rangle e = \lambda x : \rho.(\langle c_2\rangle e)(\langle c_1\rangle x) \tag{10}$$

If e is a lambda abstraction an η-redex need not be introduced. The following rule applies instead:

$$\langle c_1 \to c_2\rangle \lambda x : \rho_1'.e = \lambda x : \rho_1.\texttt{let}\ x : \rho_1' = \langle c_1\rangle x\ \texttt{in}\ \langle c_2\rangle e \tag{11}$$

It should be noted that in fact the above rules are shorthands for a set of typed rules. They only apply to terms which are well-typed in some environments Γ and A.

Simplification applies the above rules to a Core-XML expression until no rule applies any more. Preference is given to (8) before (11) and, at last, (10). The resulting expression contains only primitive coercions.

6 An Execution Model

The implementation of Core-XML on the EV-machine, which is defined below, satisfies the parameter passing assumption and is therefore an adequate operational model for representation typing. It is similar to the CAM [3]. A machine state is a triple $C\langle E, V\rangle$ of code C, an environment E, and a value V. V is used for passing the parameter to a function and to hold the return value of a function. The environment E is a heterogeneous list of values where a value is either a primitive value (an integer, a floating-point number, etc.) or a closure which is represented as a pair of code and an environment. The code C is a list of instructions described below.

In the transition rules for the instructions we use $(\_,\_)$ to construct closures, $:$ for list (environment) construction, and $E\downarrow i$ to access the environment (the i^{th} element of list E, starting from 0).

$$
\begin{array}{rlcl}
\mathtt{VAR}_\theta(i): & C\langle \qquad E, V\rangle & \mapsto & C\langle \qquad E,\ E\downarrow i\rangle \\
& & & \text{if } E\downarrow i : \theta \\
\mathtt{CLS}(C'): & C\langle \qquad E, V\rangle & \mapsto & C\langle \qquad E, (C', E)\rangle \\
\mathtt{APP}: & C\langle (C', E') : E, V\rangle & \mapsto & C'\langle (C, E) : E',\ V\rangle \\
\mathtt{ENTER}_\theta: & C\langle \qquad E, V\rangle & \mapsto & C\langle V : E,\ V\rangle \\
& & & \text{if } V : \theta \\
\mathtt{LEAVE}: & C\langle X : E, V\rangle & \mapsto & C\langle \qquad E,\ V\rangle \\
\mathtt{RET}: & C\langle (C', E') : E, V\rangle & \mapsto & C'\langle \qquad E',\ V\rangle
\end{array}
$$

The $\mathtt{VAR}_\rho$ instruction transfers the ith entry from the environment to the value register V. The representation type of the moved value must match θ. The $\mathtt{CLS}(C')$ instruction constructs a closure consisting of the code C' and the current environment. The $\mathtt{APP}$ instruction performs function application: the closure on top of the environment list is applied to the value V. The continuation closure, consisting of the remaining code and the old environment, is put on top of the new environment which is install in place of the old environment. The $\mathtt{ENTER}_\rho$ instruction saves the current value V on top of the environment. The representation type of the moved value must match θ. The $\mathtt{LEAVE}$ instruction pops the top of the environment. The $\mathtt{RET}$ instruction calls the continuation closure on top of the environment.

The translation $\mathcal{T}$ of Core-XML-expressions into EV-code is straightforward. It makes use of a symbol table σ mapping variables to environment positions and a number n indicating the current depth of the environment.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathcal{T} : \text{Core-XML} \to (\text{Var} \to \mathbf{N}) \to \mathbf{N} \to \text{Code}} \\
\mathcal{T}[\![x : \rho]\!]\sigma n & = & \mathtt{VAR}_\rho(n - \sigma(x)) \\
\mathcal{T}[\![\lambda x : \rho.e]\!]\sigma n & = & \mathtt{CLS}(\mathcal{B}^\rho_x[\![e]\!]\sigma(n+1); \mathtt{RET}) \\
\mathcal{T}[\![e_1\ e_2]\!]\sigma n & = & \mathcal{T}[\![e_1]\!]\sigma n; \mathtt{ENTER}_{\alpha\to\beta}; \mathcal{T}[\![e_2]\!]\sigma(n+1); \mathtt{APP} \\
\mathcal{T}[\![\mathtt{let}\ x : \theta = e_1\ \mathtt{in}\ e_2]\!]\sigma n & = & \mathcal{T}[\![e_1]\!]\sigma n; \mathcal{B}^\theta_x[\![e_2]\!]\sigma n
\end{array}
$$

No code is generated for type abstractions and type application, they do not give rise to computations.

The auxiliary $\mathcal{B}^\rho_x[\![e]\!]\sigma n$ translates the body e of a binding. It generates code to bind x to the current value in V, evaluate e, and remove the binding again. The arguments σ and n retain their meaning.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathcal{T} : \text{Core-XML} \to (\text{Var} \to \mathbf{N}) \to \mathbf{N} \to \text{Code}} \\
\mathcal{B}^\rho_x[\![e]\!]\sigma n & = & \mathtt{ENTER}_\rho; \mathcal{T}[\![e]\!]\sigma[x \mapsto n+1](n+1); \mathtt{LEAVE}
\end{array}
$$

The following additional instructions handle boxing and unboxing. To define the instructions let V be a value of representation type υ (which is unboxed at the top level) and V' be its (top level) boxed counterpart. For example, if υ is `int` then V could be the direct representation of a number and V' could be a pointer to the direct representation of the same number.

$$
\begin{array}{rlcl}
\mathtt{BOX}_\upsilon: & C\langle E, V\rangle & \mapsto & C\langle E, V'\rangle \\
\mathtt{UNBOX}_\upsilon: & C\langle E, V'\rangle & \mapsto & C\langle E, V\rangle
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{T}[\![\langle \mathrm{box}_\upsilon\rangle\ e]\!]\sigma n & = & \mathcal{T}[\![e]\!]\sigma n; \mathtt{BOX}_\upsilon \\
\mathcal{T}[\![\langle \mathrm{unbox}_\upsilon\rangle\ e]\!]\sigma n & = & \mathcal{T}[\![e]\!]\sigma n; \mathtt{UNBOX}_\upsilon
\end{array}
$$

The following theorem shows the semantic correctness of representation typing under the parameter passing assumption with respect to the EV-machine.

Theorem 2 *Let A be a type assumption, E an environment, $\sigma \in Var \to \boldsymbol{N}$, and $n \in \boldsymbol{N}$ such that $n = |E| > |\operatorname{dom}(\sigma)|$, $\operatorname{dom}(\sigma) = \operatorname{dom}(A)$, and for all $x \in \operatorname{dom}(\sigma)$ it holds that $E \downarrow (n - \sigma(x)) : A(x)$.*
If $\Gamma, A \vdash e \rightsquigarrow \tilde{e} : \rho$ then $\mathcal{T}[\![\tilde{e}]\!]\sigma n \langle E, x\rangle \mapsto^* \langle E, V\rangle$ and $V : \rho$.

7 Constructed Datatypes

To extend representation typing to other type constructors we must supply a sorting for it, a rule for the passability predicate P, inference rules for the constructors and selectors, a coercion formation rule, and conversion rules for coercions involving the new constructors.

7.1 Products

The sorting of the product constructor $\Sigma(\times) = (HH, U)$: the natural representation of a pair is unboxed and it accepts any components. A pair is passable if and only if both its components are passable. Predicate P extends to pairs as follows:

$$\text{(P-PROD)} \quad \frac{P(\Gamma, \rho_1) \qquad P(\Gamma, \rho_2)}{P(\Gamma, \rho_1 \times \rho_2)}$$

The additional representation type inference rules are such that the rules for introduction and elimination (R-PROD, R-PROJ) of product types require that the values must obey P since the code must be able to move them.

$$\text{(R-PROD)} \quad \frac{\Gamma, A \vdash e_1 : \rho_1 \qquad \Gamma, A \vdash e_2 : \rho_2 \qquad P(\Gamma, \rho_1 \times \rho_2)}{\Gamma, A \vdash (e_1, e_2) : \rho_1 \times \rho_2}$$

$$\text{(R-PROJ)} \quad \frac{\Gamma, A \vdash e : \rho_1 \times \rho_2 \qquad P(\Gamma, \rho_1 \times \rho_2)}{\Gamma, A \vdash \pi_i(e) : \rho_i} \quad i = 1, 2$$

Coercion formation for pairs is done as follows.

$$\frac{c_1 : \rho_1 \rightsquigarrow \rho_1' \qquad c_2 : \rho_2 \rightsquigarrow \rho_2'}{c_1 \times c_2 : \rho_1 \times \rho_2 \rightsquigarrow \rho_1' \times \rho_2'}$$

The following conversion rules hold for coercions on pairs.

$$\begin{aligned} (c_1 \times c_2); (c_1' \times c_2') &= (c_1; c_1') \times (c_2; c_2') \\ \iota \times \iota &= \iota \\ (c_1 \times c_2)^{-1} &= c_1^{-1} \times c_2^{-1} \end{aligned}$$

Alternatively, the passability predicate in the typing rules (R-PROD, R-PROJ) can be dropped if the pair constructor and the projections are supplied as overloaded polymorphic functions: $\mathtt{pair} : \forall \alpha{:}H \,.\, \forall \beta{:}H \,.\, \alpha \to \beta \to \alpha \times \beta$, $\pi_i : \forall \alpha_1{:}H \,.\, \forall \alpha_2{:}H \,.\, \alpha_1 \times \alpha_2 \to \alpha_i$. Under the parameter passing assumption no constraints are placed on the component types of the product. To implement `pair` we observe that inline code can be generated as soon as there is a type application $\mathtt{pair}\ \{\rho_1, \rho_2\}$ where both of ρ_1 and ρ_2 are passable. If there are type applications $\mathtt{pair}\ \{\rho_1, \rho_2\}$ where one or both of ρ_1 and ρ_2 are not passable we introduce additional parameters which abstract those data constructors and selectors for which inline code cannot be generated. The additional parameters are then propagated backwards through the program. As an example consider `fapp` defined by

$$\Lambda\alpha{:}B \,.\, \Lambda\beta{:}H \,.\, \Lambda\gamma{:}\top \,.\, \lambda f : \alpha \to (\beta \times \gamma).\lambda x : \alpha.\pi_1\ \{\beta, \gamma\}\ (f\ x).$$

Here the instantiation $\pi_1\ \{\beta, \gamma\}$ does not lead to a passable type. We propose to parameterize them as in:

$$\begin{aligned} &\Lambda\alpha{:}B \,.\, \Lambda\beta{:}H \,.\, \Lambda\gamma{:}\top \,. \\ &\lambda\pi_1 : (\beta \times \gamma) \to \beta.\lambda f : \alpha \to (\beta \times \gamma).\lambda x : \alpha.\pi_1\ \{\beta, \gamma\}\ (f\ x) \end{aligned}$$

Every occurrence of $\mathtt{fapp}\ \{\rho_1, \rho_2, \rho_3\}$ must be replaced by $\mathtt{fapp}\ \{\rho_1, \rho_2, \rho_3\}\ (\pi_1\ \{\rho_2, \rho_3\})$ in the remaining program. If ρ_2 and ρ_3 are passable types we are done. Otherwise $\pi_1\ \{\rho_2, \rho_3\}$ must be parameterized again and propagated further.

A similar goal could be achieved through overloading similar to type classes [21, 7, 10]. Define a type class $R_\times$ with `pair`, π_1, and π_2 as member functions. The (implicit) instances of $R_\times$ are all $\rho_1 \times \rho_2$ where ρ_1 and ρ_2 are passable types.

7.2 Algebraic Types

Algebraic datatypes are defined by domain equations of the form:

$$\chi(\alpha_1, \ldots, \alpha_k) \cong C_1\tau_1 + \cdots + C_m\tau_m$$

where $k \geq 0$, $m > 0$, the C_i are different *constructor names*, the τ_i are standard types with $FV(\tau_i) \subseteq \{\overline{\alpha_k}\}$, and $+$ denotes the smash sum of domains. The constructors serve to provide names for the summands. This corresponds to an ML `datatype` definition.

The representation type of a constructor C_i can be read off the domain equation:

$$C_i : \overline{\forall\alpha_k{:}Z_k} \,.\, \hat{\tau}_i \to \chi(\alpha_1, \ldots, \alpha_k)$$

where $\overline{Z_k}$ is maximal such that $\{\overline{\alpha_k{:}Z_k}\} \Vdash \hat{\tau}_i{:}H$ can be derived ($\hat{\tau}$ is the interpretation of a standard type as a representation type).

There are two subcases for the non-recursive case: $m = 1$, where only one summand is present, and $m > 1$, where several summands must be discriminated. In the case $m = 1$ the datatype χ is represented just like the type τ_1 and χ's sorting is the sorting of τ's top constructor. For $m > 1$ the generic representation is a heap-allocated record where the first word holds a tag value which discriminates the different summands and the rest has the same layout as the respective $\hat{\tau}_i$ [1, 15]. E.g. in a stack-based implementation the unboxed representation of such a value could put the tag value on top on the stack and the components below [14]. However, passing such a value requires conditional code since the contents of the stack depend on the tag value. That is only acceptable if the unboxed value is the subject expression of a `case`-expression. In all other cases conditional code should be avoided because of its expense, and the values should be boxed as soon as their top constructor (the value of their tag) is no longer statically known.

In the recursive case where χ appears in at least one τ_i, the recursive parts must be boxed to stop the recursion. Otherwise we would have to deal with structures of unknown and unbounded size. How much of the non-recursive parts need to be boxed depends on the approach that is adopted.

As an example we consider lists defined by

$$\texttt{list}(\alpha) \cong \texttt{nil}() + \texttt{cons}(\alpha \times \texttt{list}(\alpha)).$$

By choosing a list representation which places the recursive field (the tail) first we obtain the typing $\texttt{tl} : \forall\alpha{:}\top \,.\, \texttt{list}\ \alpha \to \texttt{list}\ \alpha$ for free. To see this, observe that the format of the (boxed) recursive field is known and that it is not necessary to know the size of the list element in order to access the tail. The unconstrained type for `tl` induces similar unconstrained types for functions like `null`, `len`, and `drop`.

Representations for the elements of a list can be chosen in several different ways. Leroy [11] proposes to box all list elements recursively. Poulsen's global representation analysis [17] requires that list elements are boxed as far as any polymorphic function which is applied to the list requires. Harper and Morrisett [6] propose to keep all list elements unboxed and to compute the access functions on the fly from type applications. This requires non-trivial computations to be performed at type applications.

We propose to keep list elements unboxed in general and to pass the required access functions as additional parameters, as demonstrated above for pairs. This way, `cons` and `hd` can have typings $\texttt{cons} : \forall\alpha{:}H \,.\, \alpha \to \texttt{list}\ \alpha \to \texttt{list}\ \alpha$ and $\texttt{hd} : \forall\alpha{:}H \,.\, \texttt{list}\ \alpha \to \alpha$ which amounts to overloading the constructor and selector operations for lists. As in the case for pairs we either have to ensure that ρ in all applications $\texttt{cons}\ \{\rho\}$ and $\texttt{hd}\ \{\rho\}$ is passable or we have to parameterize all occurrences $\texttt{hd}\ \{\rho\}$ and $\texttt{cons}\{\rho\}$ where ρ is not passable.

8 Practical Results

We have implemented (in SML) representation type inference and the translation into Core-XML as presented in this paper. In order to evaluate the quality of the translation we have build an interpreter for Core-XML, which keeps track of the executed boxing operations. We have obtained results on a set of benchmarks, which has been used by Leroy, Poulsen, Henglein, and Jørgensen. Program "msort" is a monomorphic insertion sort, "psort" is a polymorphic insertion sort, "mswap" is `map swap` on a list of pairs, "leroy" is the "pathological example" of Leroy, "poulsen" is a pathological example for Poulsen's algorithm, "sieve" computes prime number, and "horner" evaluates a polynomial by constructing a list of functions.

We have done experiments in two modes indicated by suffixes "L" and "P". Mode "L" behaves like Leroy's algorithm: generic type variables are only instantiated with boxed types. To obtain a fair comparison we have allowed for unboxed list elements and have simplified the coercions afterwards (a la [8]). Mode "P" implements the parameter passing assumption with unboxed lists and simplification. The table gives the number of box and unbox operations executed in one run of the program. Suffix "LP" means that the results in both modes are identical.

program	int		$\to$		$\times$	
	b	u	b	u	b	u
msort/LP	0	0	0	0	0	0
psort/LP	17	152	0	0	0	0
mswap/L	10	0	0	0	10	5
mswap/P	10	0	0	0	0	0
leroy/LP	257	256	12	12	0	0
poulsen/LP	3	3	0	0	0	0
sieve/LP	99	436	0	0	0	0
horner/P	11	10	0	0	0	0
horner/L	31	30	0	0	0	0

9 Extensions

Built-in Functions Built-in polymorphic functions are easy to add. For typing purposes they can be treated as polymorphic constants if the parameter passing assumption is employed. For example the fixpoint operator `fix` can be given the type $\forall\alpha{:}\top\,.\,(\alpha \to \alpha) \to \alpha$, and the conditional is $\mathtt{cond} : \forall\alpha{:}\top\,.\,\mathtt{bool} \to \alpha \to \alpha \to \alpha$.

A Better Type for the Identity As already remarked in Sec. 3.5 the type for the identity function is $\forall\alpha{:}B\,.\,\alpha \to \alpha$. This is not as satisfactory as it could be since the code for it could be just a return instruction. Here, we analyze how this can be remedied.

First, the machine needs a new instruction $\mathtt{APP}(i)$ which makes it possible to apply a closure directly out of the environment without first transferring it to V.

$$\mathtt{APP}(i) : C\langle E, V\rangle \mapsto C'\langle (C,E) : E', V\rangle \text{ where } (C', E') = E \downarrow i$$

Second, we improve the code generation for binders such that the bound variable is only moved to the environment when absolutely necessary. To this end, the number of free occurrences of the bound variable is counted and its context analyzed.

The definition of E below characterizes contexts which need not perform serious computations before evaluating their hole.

$$E ::= [\,] \mid (v\ E) \mid ((\lambda v.e)\ E) \mid \mathtt{let}\ v = E\ \mathtt{in}\ e$$

Using $count(x, e)$ (number of free occurrences of x in e) and contexts E we can define an optimizing version of $\mathcal{B}^{\rho}_{x}$:

$$\mathcal{B}^{\rho}_{x}[\![e]\!]\sigma n = \begin{cases} \mathcal{T}[\![e]\!]\sigma n & \text{if } count(x,e) = 0 \\ \mathcal{T}_x[\![e]\!]\sigma n & \text{if } count(x,e) = 1 \wedge e = E[x] \\ \text{as before} & \text{otherwise.} \end{cases}$$

$$\begin{array}{lcl} \mathcal{T}_x : E[x] \to (\mathrm{Var} \to \mathbf{N}) \to \mathbf{N} \to \mathrm{Code} & & \\ \mathcal{T}_x[\![x]\!]\sigma n & = & \varepsilon \\ \mathcal{T}_x[\![v\ E[x]]\!]\sigma n & = & \mathcal{T}_x[\![E[x]]\!]\sigma n; \mathtt{APP}(n - \sigma(v)) \\ \mathcal{T}_x[\![(\lambda v:\rho.e)E[x]]\!]\sigma n & = & \mathcal{T}_x[\![E[x]]\!]\sigma n; \mathcal{B}^{\rho}_{v}[\![e]\!]\sigma n \\ \mathcal{T}_x[\![\mathtt{let}\ v:\theta = E[x]\ \mathtt{in}\ e]\!]\sigma n & = & \mathcal{T}_x[\![E[x]]\!]\sigma n; \mathcal{B}^{\theta}_{v}[\![e]\!]\sigma n \end{array}$$

Now, $\mathcal{T}[\![\lambda x.x]\!]\{\}0 = \mathtt{CLS}(\mathtt{RET})$. Therefore, the type of $\lambda x.x$ can be $\forall\alpha{:}\top\,.\,\alpha \to \alpha$. The improved translation facilitates better types for `apply`, the B combinator, and others.

Monomorphic Expansion A brute-force approach to enjoy unboxed values everywhere is the full monomorphic expansion of a program. It is constructed by replicating the definitions of polymorphic values for each monomorphic instance which is used in the program. It is well-known that the size of a program may increase exponentially under monomorphic expansion, but recent experiments of Jones [10] have shown the actual increase in size may be much less.

With the typings provided by our approach, all data can remain unboxed without the cost of a full monomorphic expansion: definitions need only be replicated for all monomorphic instances of their *boxed* generic type variables. Some amount of polymorphism can remain.

Using the type system to guide such a "boxed monomorphic expansion" has an additional advantage: the expensive simplification for coercions can simply be omitted.

Modularity An advantage of Leroy's algorithm over other approaches is its modularity. External functions can be treated as soon as their (standard) types are known. Our approach is also modular. If only the standard type of a function is known, a representation type which assumes the worst (makes all the generic variables boxed and assumes lists to be recursively boxed) will work. Once the representation type of an external function is known it is treated like any other function.

More Don't-Care-Polymorphism Don't-Care-Polymorphism as we have presented it is pretty much confined to the body of single functions. In order to take advantage of it on a global scale, we could make use of the overloading which is possible in our order-sorted type language. If each type constructor has the sorting $(\top \ldots \top, \top)$ in addition to its usual sorting then types of sort $\top$ come up in a program. However, values that are used have a type of sort at most H and a value of a type of sort $\top$ is never used in a computation. Therefore every computation on the latter values is dead.

10 Related Work

Leroy discusses handling unboxed values in the presence of polymorphic typing [11]. He translates an ML core language into a language where all boxing operations are explicit. The translation is guided by the principal ML-typing derivation. A generic type variable can only be instantiated to a boxed type. Our algorithm inserts representation coercions at function applications while Leroy's algorithm places them at variables. He seems to inspect the result of the type reconstruction algorithm only after it is completed while we insert coercions during the reconstruction process.

Henglein and Jørgensen define a notion of formally optimal boxing based on a calculus of representation coercions [8]. They identify two notions to optimally insert coercions into an expression. However, they are not specific about how coercions are placed in an expression but they also assume that only boxed types can be substituted for generic type variables. They do not provide a translation or an operational semantics. Their representation type system is very similar to the one presented here, but they have neither an order-sorted representation type language (and hence no quantification over unrestricted type variables) nor do they consider operational assumptions.

Launchbury and Peyton Jones optimize the handling of base type values inside of a compiler for a lazy language by introducing unboxed data types [16]. They can be used to express the transformation from the safe boxed representation of a value (which is a first class value) to the unboxed representation which must be used more restrictively. They also allow for some unboxed constructed datatypes. Since they work in a non-strict language unboxing also means evaluation and thus affects strictness properties. However, they require program annotations to introduce unboxed values and do not attempt to automate the coercion to and from unboxed values. They do not allow passing unboxed values as parameters to polymorphic functions.

Recent work by Nöcker and Smetsers addresses partially strict non-recursive data types in the context of lazy functional languages [14]. A special case of such a data type is a partially strict tuple which is comparable to an unboxed tuple. They rely on programmer supplied annotations to functions and data type declarations.

Thatte defines a very general framework for typing in the presence of isomorphisms between types [20]. The isomorphisms induce a non-trivial equational theory on the type language. Our type reconstruction algorithm may be seen as an application of his framework adapted to an order-sorted type language.

11 Conclusion

We have shown that polymorphic functions in strict functional languages need not impose as many boxing constraints as was previously known. Only two mild assumptions on the underlying execution model are necessary: tail calls must be properly eliminated by the implementation and the convention to pass the parameter to a function must coincide with the convention to return the value of a function application.

Boxing constraints are cleanly expressed with an order-sorted type language. An automatic translation is given from an ML subset into an explicitly typed language with coercions. An adequate execution model is specified with a simple abstract machine.

It would be interesting to examine the applicability of our framework to non-strict languages.

References

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

[2] D. Clément, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ML. In *ACM Symposium on LISP and Functional Programming*, pages 13–27, 1986.

[3] G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. *Science of Programming*, 8:173–202, 1987.

[4] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. 9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[5] M. J. Fischer. Lambda-calculus schemata. *Lisp and Symbolic Computation*, 6(3/4):259–288, 1993.

[6] R. Harper and G. Morrisett. Compiling with non-parametric polymorphism. Technical Report CMU-CS-94-122, Carnegie Mellon University, Pittsburgh, PA 15213, 1994.

[7] Report on the programming language Haskell, a non-strict, purely functional language, version 1.2. *SIGPLAN Notices*, 27(5):R1–R164, May 1992.

[8] F. Henglein and J. Jørgensen. Formally optimal boxing. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, pages 213–226, Portland, OG, Jan. 1994. ACM Press.

[9] J. Hughes, editor. *Functional Programming Languages and Computer Architecture*, Cambridge, MA, 1991. Springer-Verlag. LNCS 523.

[10] M. P. Jones. Partial evaluation for dictionary-free overloading. In P. Sestoft and H. Søndergaard, editors, *Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 107–118, Orlando, Fla., June 1994. ACM.

[11] X. Leroy. Unboxed objects and polymorphic typing. In *Proc. 19th ACM Symposium on Principles of Programming Languages*, pages 177–188, Albuquerque, New Mexico, Jan. 1992.

[12] J. Meseguer, J. Goguen, and G. Smolka. Order-sorted unification. *Journal of Symbolic Computation*, 8:383–413, 1989.

[13] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17:348–375, 1978.

[14] E. Nöcker and S. Smetsers. Partially strict non-recursive data types. *Journal of Functional Programming*, 3(2):191–216, Apr. 1993.

[15] S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[16] S. L. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In Hughes [9], pages 636–666. LNCS 523.

[17] E. R. Poulsen. Representation analysis for efficient implementation of polymorphism. Master's thesis, Department of Computer Science (DIKU), University of Copenhagen, Apr. 1993.

[18] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3/4):289–360, 1993.

[19] M. Schmidt-Schauß. A many-sorted calculus with polymorphic functions based on resolution and paramodulation. In *Proc. 9th International Joint Conference on Artificial Intelligence*, pages 1162–1168, 1985.

[20] S. R. Thatte. Coercive type isomorphism. In Hughes [9], pages 29–49. LNCS 523.

[21] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Proc. 16th ACM Symposium on Principles of Programming Languages*, pages 60–76, 1989.