Flexible Representation Analysis*

Zhong Shao

Dept. of Computer Science
Yale University
New Haven, CT 06520-8285
shao-zhong@cs.yale.edu

Abstract

Statically typed languages with Hindley-Milner polymorphism have long been compiled using inefficient and fully boxed data representations. Recently, several new compilation methods have been proposed to support more efficient and unboxed multi-word representations. Unfortunately, none of these techniques is fully satisfactory. For example, Leroy's coercion-based approach does not handle recursive data types and mutable types well. The type-passing approach (proposed by Harper and Morrisett) handles all data objects, but it involves extensive runtime type analysis and code manipulations.

This paper presents a new *flexible representation analysis* technique that combines the best of both approaches. Our new scheme supports unboxed representations for recursive and mutable types, yet it only requires little runtime type analysis. In fact, we show that there is a continuum of possibilities between the coercion-based approach and the type-passing approach. By varying the amount of boxing and the type information passed at runtime, a compiler can freely explore any point in the continuum—choosing from a wide range of representation strategies based on practical concerns. Finally, our new scheme also easily extends to handle type abstractions across ML-like higher-order modules.

1 Introduction

Statically typed languages with Hindley-Milner polymorphism [8] have long been compiled using inefficient and *fully boxed* data representations. Under these implementations, all program variables, function closures, function parameters, and record fields are uniformly represented in exactly one word. If the natural representation of a value does not fit into one word, the value is boxed (e.g., allocated on the heap) and the pointer to this boxed object is used instead. For example, in the following ML code,

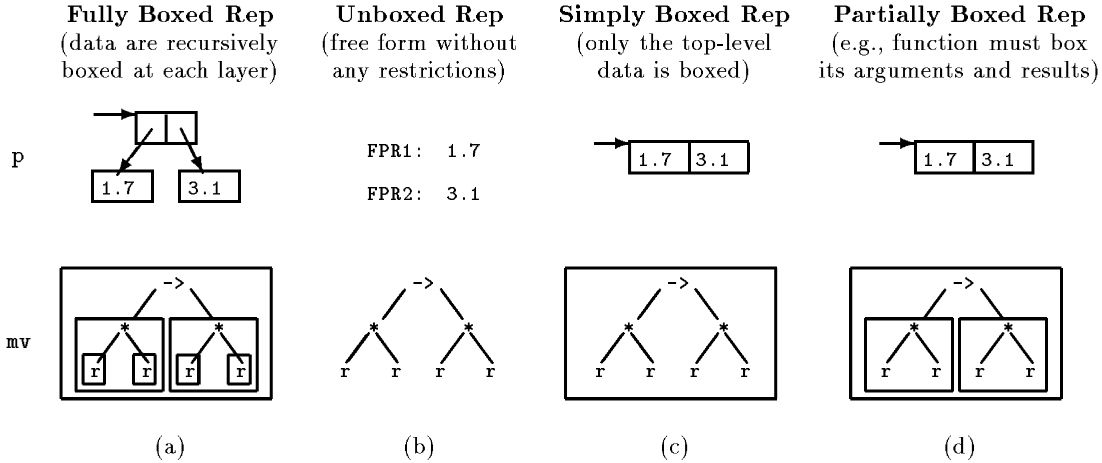
```
fun quad (f,x) =
  let val z = f(f(f(f(x))))
   in (z::z::nil)
  end
val p = (1.7, 3.1)
fun mv (x,y) = (x + 3.1, y + 2.7)
```

function quad is a polymorphic function with type $\forall\alpha.(((\alpha \rightarrow \alpha) * \alpha) \rightarrow \alpha\ \texttt{list})$; all of the four calls to f inside quad must use the most conservative calling conventions—passing x as a single-word object. Value p is a pair of floats, but since it might be passed to polymorphic functions and treated as objects of type $\alpha * \alpha$, its entire data structure is fully boxed at each layer (see Figure 1a). Similarly, though function mv has monomorphic type $(\texttt{real} * \texttt{real}) \rightarrow (\texttt{real} * \texttt{real})$, it cannot use any special calling conventions; not only it is represented as a boxed closure, but its arguments and return results must be assumed as fully boxed as well (see Figure 1a). Uniform full boxing does implement polymorphism correctly, but it has two major drawbacks: first, because of the boxing, monomorphic code runs much slower than those written in C or assembly languages; second, since all data objects must be fully boxed, the interoperability with low-level C or assembly code is difficult and inefficient.

Xavier Leroy [19] recently presented a *representation analysis* technique that does not always require variables be boxed in one word. In his scheme, monomorphic objects such as p and mv can use efficient *unboxed* representations (see Figure 1b): value p can stay in two floating-point (FP) registers and function mv can freely pass the arguments and return the results in two FP registers. When monomorphic objects are passed to polymorphic contexts (as in quad(mv, p)), they are coerced back into fully boxed representations. Leroy's technique, unfortunately, does not handle recursive datatypes and mutable types well. Coercions on large data structures are impractical because the cost of the copying often outweighs the benefits of unboxed representations [19, 14]. More seriously, mutable data structures such as arrays cannot be copied or coerced; if we make a copy of the value to box the components, then updates to the copy will not be reflected in the original array and vice versa. As a result, values such as lists and arrays must still use fully boxed representations, even when they are not inside polymorphic contexts.

Harper and Morrisett [14, 24] later solved this problem on recursive and mutable types using a *type-passing* approach. Under their scheme, data objects (including lists and arrays) are kept "unboxed" all the time, even inside polymorphic contexts. Polymorphism is not implemented through coercions, but by using runtime type analysis and code manipulations. For example, function quad is implemented with

*To appear in the Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP'97). This research was sponsored in part by the Defense Advanced Research Projects Agency ITO under the title "Building Evolutionary Software through Modular Executable Specifications and Incremental Derivations," DARPA Order No. D961, issued under Contract No. F30602-96-2-0232, and in part by an NSF CAREER Award CCR-9501624, and NSF Grant CCR-9633390. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.



We use boxed type trees to illustrate boxing for complicated type structures (e.g., `mv`'s). Each box refers to one boxing layer. ML type `real` is abbreviated as the symbol `r`.

Figure 1: Comparison of Various Data Representations.

an extra runtime type parameter α; all primitive operations inside `quad` (e.g., function call `f(x)`, list *cons* `z::nil`) must analyze the type (α) at runtime in order to select and dispatch to the appropriate code to manipulate unboxed objects. The type-passing approach is still not satisfactory, for the following reasons:

- Although monomorphic code can fully take advantages of the unboxed representations, polymorphic code becomes much slower than the full-boxing approach. In fact, all primitive operations inside polymorphic contexts are no longer *primitive*: simple function calls (or returns) and record operations (e.g., creation, selection, list *cons*) now become either indirect procedure calls or large `typerec` switches [24]. Compiler optimizations such as type specialization may eliminate part of these overheads, but they can lead to code explosions or excessively long compile time.

- A more severe problem, also called *the vararg problem* (see [24, page 216 and 175] for details), is the implementation of function definitions and function calls with arguments of unknown types (e.g., the application of `f` inside `quad`). The challenge is to simulate the advanced calling conventions based on the runtime type information (since the actual `f` may use any calling conventions). For machines with k argument registers (including FP registers), this would require 2^k cases (e.g., entrant code, coercions) to deal with all the possible calling conventions [24]. Actually, modern compilers often use even more elaborate calling conventions [5, 17]—making the above simulation virtually impossible. Of course, one could always resort to *runtime code generation* [18, 4, 9] or simply use a very restricted set of calling conventions [24], but then, either the cost is too expensive or the interoperability suffers.

This paper presents a new *flexible representation analysis* technique that combines the best of both the coercion-based and the type-passing approaches. Our new scheme supports unboxed representations for recursive and mutable types, yet it only requires little runtime type analysis. Our idea is simple: we avoid the heavy-weight runtime type manipulations by boxing all polymorphic values; however, instead of doing full boxing as in the coercion-based approach [19], we use the *simply boxed* representation (see Figure 1c) or other *partially boxed* representations (see Figure 1d). Intuitively, a simply boxed object just boxes the top layer of the data structure so that the entire object can be referenced as a single-word pointer. Simple boxing is generally much cheaper than full boxing, and most of the time, it is just an identity function because the natural representations of many "unboxed" objects (e.g., lists, closures, records, arrays) are already simply boxed. Simple boxing solves the problem of recursive and mutable types because any simply boxed object can be easily unboxed (at the top layer) before being *cons*-ed onto lists or put inside arrays.

Simple boxing is trickier to implement than full boxing because the coercion may also rely on runtime type information. For example, coercing an object of type $\beta * \gamma$ into type α would involve first unboxing the β and γ fields, and then pairing them up based on the actual types β and γ have at runtime.

More interestingly, there is a continuum of freedom between the coercion-based and the type-passing approaches: by varying the amount of boxing and the type information passed at runtime, a compiler can freely explore any point in the continuum, choosing from a wide range of representation strategies as the *canonical boxed form*. This flexibility is particularly nice because one can trade boxing with runtime type manipulations on the per-tycon (i.e., type constructor) basis. For example, using the partially boxed representation shown in Figure 1d, we can completely eliminate the vararg problem in the type-passing approach (see later sections for more details).

The main contributions of this paper are:

- Among all the known coercion-based approaches [19, 26, 27, 15, 33, 36], our scheme is the first to successfully solve the open problem on recursive and mutable types.

- Unlike the type-passing approaches [25, 14, 24, 34], our

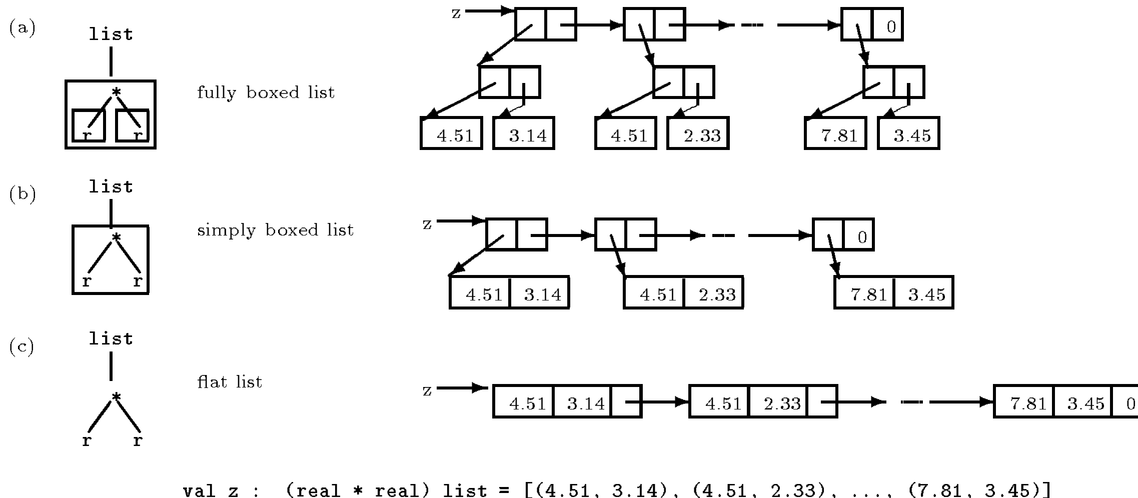


Figure 2: Comparison of Various List Representations

scheme requires little runtime type analysis, even for the heavily polymorphic programs.

- Our scheme is very flexible because a compiler can trade boxing with runtime type analysis on the per-tycon basis. In Section 3, we present a formal framework and a set of axioms that precisely characterize this trade-off.
- By choosing partially boxed representations (e.g., Figure 1d) as the canonical boxed form, our scheme can completely eliminate the nasty vararg problem.
- We extend Leroy's representation analysis to a predicative variant of the polymorphic λ-calculus F_ω [10, 28]. We show that our technique works for both the ML-style polymorphism and the F_ω-like higher-order polymorphism.
- We show how easily our scheme can be extended to handle type abstractions across ML-like higher-order modules [22, 21, 11, 20].
- We show that with a simple twist based on the parametricity property, most runtime type manipulations in our scheme can be eliminated.
- We have implemented our scheme (with partially boxed representations) in an experimental version of the SML/NJ compiler [3, 33]. Preliminary measurements show that code involving recursive and mutable types gets significant speedup while normal polymorphic code remains almost as efficient as before.

In the rest of this paper, we first give an informal presentation of the main idea. We then formalize the presentation and give several major theorems about our flexible approach. We also show how to extend our scheme to handle the entire ML language [22]. Finally, we discuss the experimental results, the related work, and then conclude.

2 Informal Development

In this section, we present a series of examples in which we illustrate both previous approaches and our flexible approach to the implementation of polymorphism.

2.1 Canonical boxed form

The key to the implementation of polymorphism lies on how to define the *canonical boxed form*: given a polymorphic object of type α (a single type variable), what should its representation be like ? Of course, the representation depends on the actual instantiation of α. So assume α is instantiated into type τ, how do we represent such a value of type τ ?

Canonical boxed form used in all the coercion-based approaches is indeed always boxed. That is, it can be handled as a single-word object. This dramatically simplifies the implementation of polymorphism, because all objects of type α can be manipulated in the same way regardless of what α really is. For example, primitive operations such as function applications and record operations can all be inlined at compile time: a value of type $\alpha * \beta$ can be built by simply creating a two-word record, each containing the corresponding boxed value, and so on. In type-passing approaches, the "canonical boxed form" is not always boxed, so it may not fit in one word; this makes all polymorphic primitive operations dependent on runtime types.

Once the canonical boxed form is decided, two primitive coercion functions can be defined: $\mathtt{wrap}[\tau]$ converts a value of type τ from its natural unboxed form into the corresponding canonical boxed form—simulating a value of type α; $\mathtt{unwrap}[\tau]$ does the reverse.

Coercions for more complicated types can be inductively defined based on these two primitive coercions (see Section 3.5 or Leroy [19] for more details). For example, to implement the function application `quad(mv, p)` defined earlier, `mv` is coerced from $(\mathtt{real} * \mathtt{real}) \rightarrow (\mathtt{real} * \mathtt{real})$ to $\alpha \rightarrow \alpha$ as follows (assuming a call-by-value semantics):

$$mv' = \lambda y : \alpha.\mathtt{wrap}[\mathtt{r*r}](\mathtt{mv}(\mathtt{unwrap}[\mathtt{r*r}](y)));$$

then `p` is coerced into $p' = \mathtt{wrap}[\mathtt{r*r}](\mathtt{p})$, and the function call to $\mathtt{quad}(mv', p')$ is performed.

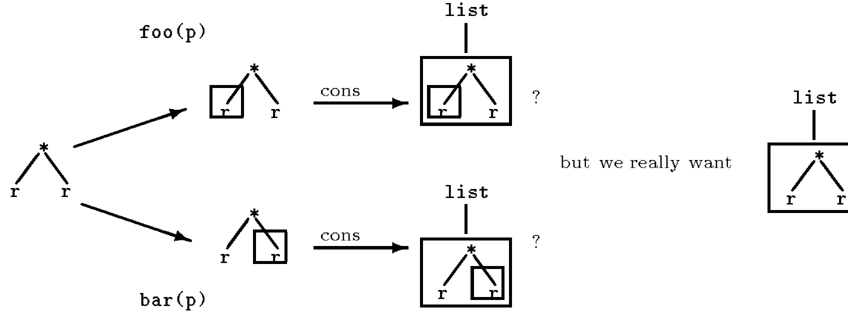


Figure 3: Coercion Must Commute with Type Instantiations

Coercion-based approaches would have worked nicely if all data objects were *coercible*. Unfortunately, many commonly used objects are *incoercible*. Coercions on large data structures are impractical because the cost of the copying often outweighs the benefits of unboxed representations [19]. More seriously, mutable data structures such as arrays cannot be copied or coerced; if we make a copy of the value to box the components, then updates to the copy will not be reflected in the original array and vice versa. Therefore, all coercions on recursive and mutable objects must be treated as identity functions.

In the earlier example, the result of quad(mv',p') should be coerced from α `list` to (`real`$*$`real`) `list`. But since this is too expensive, the list is not coerced. This forces all lists of type (`real` $*$ `real`) `list` to use same representations as those for α `list`, $(\beta * \gamma)$ `list`, or ($\beta *$`real`) `list`, etc., since they can all possibly be instantiated into (`real`$*$`real`) `list`.

For this reason, all previous coercion-based approaches use full boxing as the canonical boxed form. Unlike other monomorphic values, monomorphic lists such as (`real`$*$ `real`) `list` and arrays must always use the fully boxed representations as shown in Figure 2a.

2.2 Simply boxed representations

Our main idea is to use the simply boxed or partially boxed representations as the canonical boxed form. Simple boxing just boxes the top layer of a data structure. For example, simple boxing of the value `p` would just build a flat vector of floats (see Figure 1c). Under simple boxing, application of quad(mv',p') would return a simply boxed list as shown in Figure 2b.

It is easy to simply box monomorphic values such as `real` $*$ `real`, but what about polymorphic values such as $\beta * \gamma$, or $\beta *$`real`, or `real`$* \gamma$? More generally, how to define `wrap`[τ] and `unwrap`[τ] if τ contains type variables ?

One could attempt to box $\beta *$`real` by treating β as a single word. This does not go very far. Consider the following ML code:

```
fun foo(x, y) = (x, y+3.0)::nil
fun bar(x, y) = (x+3.0, y)::nil
```

Here, `foo` has type $\forall\beta.(\beta *$ `real`$) \rightarrow (\beta *$ `real`$)$ `list` and `bar` has type $\forall\gamma.($`real` $* \gamma) \rightarrow ($`real` $* \gamma)$ `list`. The list *cons* operation `::` has type $\forall\alpha.(\alpha * \alpha$ `list`$) \rightarrow \alpha$ `list`, so the pairs inside `foo` and `bar` will be coerced by either `wrap`[$\beta *$ `real`] or `wrap`[`real` $* \gamma$] before being *cons*-ed onto the empty list. Figure 3 shows the corresponding coercions inserted when we apply `foo` and `bar` to `p`.

Clearly, neither produces the simply boxed list. The correct definition of `wrap`[$\beta *$ `real`] (or `wrap`[`real` $* \gamma$]) is, of course, to first *uncover* (unbox) the β or γ field and then build the exact same vector as in Figure 1c. The "uncover" operation does not have to be done recursively because by invariant, β is already in the simply boxed form. Uncovering does require the use of runtime type information (the actual type of β); this is realized in the same way as in the type-passing approach [14].

One important insight we get here is that a boxing scheme can serve as a *valid* canonical boxed form only if it commutes with the type instantiation relations. A type variable α can be instantiated into either $\beta *$`real` or `real`$* \gamma$, then further into `real` $*$ `real`; whichever path it takes, the two resulting boxed forms (and their corresponding coercions) must be equivalent (see Section 3.5 for the details).

In most compilers, data structures like lists often use the simply boxed form as their default monomorphic representations. This fits extremely well with simple boxing because most common list functions (e.g., `cons`, `map`, `fold`, `length`, `append`) remain as efficient as they could be. For example, in the following ML code:

```
fun map f l = let fun m (a::r) = (f a) :: (m r)
                    | m nil = nil
               in m l
              end

val z = [(4.5, 3.1), (4.3, 2.0), ...]
val nz  = map mv z
```

Here `map` has type $\forall\alpha\beta.(\alpha \rightarrow \beta) \rightarrow (\alpha$ `list` $\rightarrow \beta$ `list`$)$, so function `mv` is coerced into mv' (defined before in Section 2.1). List `z` is incoercible, so it is not coerced. When the internal loop `m` traverses the list, the only invariant it requires is that every element be a single-word pointer. Simple boxing does satisfy this invariant, so no extra coercions or runtime type analysis are necessary.

There are cases where simple boxing requires more coercions than full boxing. These situations are often less common, and when they occur, the coercions are always kept at the minimum. For example, when applying the following function `unzip`, which has type $\forall\alpha\beta.(\alpha * \beta)$ `list` $\rightarrow$ α `list` $* \beta$ `list`, to the simply boxed list z in Figure 2b,

```
fun unzip l =
  let fun h((a,b)::r, u, w) = h(r, a::u, b::w)
        | h(nil, u, w) = (rev u, rev w)
   in h(l,nil,nil)
  end
```

the only coercion necessary occurs when applying the :: projection to `(a,b)::r` where each list element (a flat float vector) is coerced into a standard boxed record of type $\alpha * \beta$. (through $\mathtt{unwrap}[\alpha * \beta]$).

Simple boxing also supports flat lists as shown in Figure 2c and flat arrays easily because any simply boxed object can be uncovered at the top layer only [1] and then be *cons*-ed onto lists or put inside arrays. Of course, all polymorphic list and array primitives will now become partially dependent on the runtime types.

2.3 Partially boxed representations

One problem with simple boxing is that their primitive coercions, $\mathtt{wrap}[\tau]$ and $\mathtt{unwrap}[\tau]$, may also run into the nasty vararg problem encountered in the type-passing approach. Fortunately, by using the partially boxed representations (such as the one in Figure 1d) as the canonical boxed form, we can completely avoid this problem. Partial boxing here is similar to simple boxing, except it also maintains the invariant that *all function arguments and results are also partially boxed into a single word.*

Under partial boxing, coercions on unknown function types never depend on runtime types. For example, under simple boxing, $\mathtt{wrap}[\alpha \rightarrow \beta]$ would need to uncover α and β; this is not necessary under partial boxing because by invariant, α and β should already be in the partially boxed form.

The effect of using partial boxing on the rest of the code is very minor. Data structures such as `(real*real) list` are represented exactly same as those under simple boxing. Flat lists or flat arrays can still be built through the simple uncovering. All monomorphic functions such as $(\mathtt{real} * \mathtt{real}) \rightarrow \mathtt{real}$ can still use the special calling conventions (passing arguments in FP registers, etc); they will be partially boxed only if they are put inside data structures such as lists and arrays.

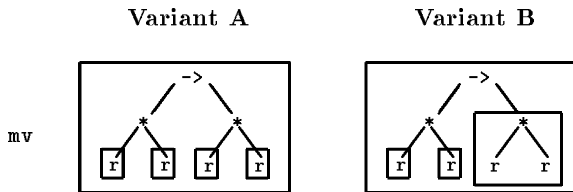


Figure 4: Two Variants of Partial Boxing

2.4 Other valid boxing schemes

Simple boxing and partial boxing are not the only possible boxing schemes. In fact, one of our main contributions is to give the precise conditions (in Section 3) about what kind of boxing schemes can serve as the valid canonical boxed form. There is actually a continuum of possibilities between full boxing and no boxing. For example, Figure 4 gives two interesting variants of the partial boxing scheme, each with different trade-offs on the vararg problem:

[1] Fully boxed objects are recursively boxed at each layer, so they cannot be easily uncovered.

- In variant A, instead of boxing the entire arguments or results into a single word, we box each individual argument and return result only. This means that multi-argument (or multi-return-result) functions can pass each argument (or result) in designated *general-purpose registers*. The vararg problem is not completely eliminated because coercing a boxed value into a function of type $\alpha \rightarrow \beta$ requires examining the runtime type of α and β.
- Variant B is same as variant A except we let the entire return results be boxed as in the standard partial boxing approach. Since most functions have a single return result (always true in C and assembly) this might be a good compromise. Of course, we could limit the number of arguments also (e.g., allow 5 boxed args maximum); doing so would further simplify the implementation of the vararg coercions.

This kind of flexibility is very useful when we compile languages with a richer set of type constructors. One can imagine to have several different record or function tycons: all would work in any contexts, but some perform better on the monomorphic code and others do better on the polymorphic ones. The flexible framework also gives us finer control on the boxing degrees, making it easier to interoperate efficiently with lower-level C and assembly code.

3 Formalization

In this section, we present a formal framework to explain our flexible representation analysis techniques. Instead of performing the analysis directly on the ML source language (SRC), we use a predicative variant of the polymorphic λ-calculus F_ω [10, 28, 14] as the intermediate language (IL). Representation analysis is then expressed as a type-directed program transformation that automatically inserts coercions and translates IL programs into the target implementation calculus (TGT—also known as λ_i^{ML} [14]). The benefit of doing it this way is to show that our analysis works not only on the ML-like polymorphism [8] but on the more general higher-order polymorphism as well.

The rest of this section is organized as follows: we first give the syntax and semantics of the three languages: SRC, IL, and TGT; we show how to embed the SRC language into IL, and then presents the IL-to-TGT translation algorithm that does the representation analysis; we prove the type correctness and the semantic correctness of our translation, and then give a set of axioms that characterize the *valid* canonical boxing schemes; finally, we formally define simple boxing and partial boxing, and show why they all satisfy these axioms.

3.1 Source language: SRC

We use a variant of Mini-ML [7] as our source language (SRC). Its syntax is defined by the following grammar:

$$
\begin{array}{llll}
(\textit{monotypes}) & \tau & ::= & t \mid \mathtt{int} \mid \mathtt{real} \mid \tau_1 * \tau_2 \\
 & & \mid & \tau_1 \rightarrow \tau_2 \mid \tau\ \mathtt{pack} \\
(\textit{polytypes}) & \sigma & ::= & \tau \mid \forall t.\sigma \\
(\textit{terms}) & e & ::= & x \mid i \mid f \mid \langle e_1, e_2 \rangle \mid \pi_1 e \mid \pi_2 e \\
 & & \mid & e_1 e_2 \mid \lambda x.e \mid \mathtt{pk}\ e \mid \mathtt{upk}\ e \\
 & & \mid & \mathtt{let}\ x = v\ \mathtt{in}\ e \\
(\textit{values}) & v & ::= & x \mid i \mid f \mid \langle v_1, v_2 \rangle \mid \lambda x.e \mid \mathtt{pk}\ v
\end{array}
$$

Here, monotypes τ are either type variables (t), `int`, `real`, binary product types, arrow types, or `pack` types. Polytypes (i.e., type schemes) σ are either monotypes or prenex quantified types. Terms e consist of identifiers (x), integer constants (i), float constants (f), pairs, first and second projections, abstractions, applications, let expressions, and packing and unpacking expressions. Values (v) are a subset of terms and include identifiers, constants, pair of values, abstractions, and packed values.

The static and dynamic semantics for SRC are all standard and same as those for Mini-ML (see [7, 14, 24]). The type inference rule (given later as part of the translation from SRC to IL in Figure 6) is in the form of $\triangle;\Gamma \vdash e : \tau$ where $\triangle$ is a set of free type variables, and Γ is a type environment mapping identifiers to polytypes. We also restrict the let-bound expressions to values [37] so that type abstractions can be made explicit in the translation. The natural (call-by-value) dynamic semantics for SRC can be defined as $e \hookrightarrow_s v$ where e is a closed expression and v is a closed value.

To show how our techniques apply to arbitrary recursive or mutable types, we introduce a special type constructor named `pack` in SRC. The typing rules and the operational semantics for packing and unpacking are defined as follows:

$$\frac{\triangle;\Gamma \vdash e : \tau}{\triangle;\Gamma \vdash \mathtt{pk}\ e : \tau\ \mathtt{pack}} \qquad \frac{\triangle;\Gamma \vdash e : \tau\ \mathtt{pack}}{\triangle;\Gamma \vdash \mathtt{upk}\ e : \tau}$$

$$\frac{e \hookrightarrow_s v}{\mathtt{pk}\ e \hookrightarrow_s \mathtt{pk}\ v} \qquad \frac{e \hookrightarrow_s \mathtt{pk}\ v}{\mathtt{upk}\ e \hookrightarrow_s v}$$

We divide all type constructors into two categories: *coercible tycons* are those type constructors whose values can be readily coerced at runtime; *incoercible tycons* are those whose values cannot be coerced because either it is too expensive or it violates the semantics. For example, `int`, `real`, binary product, and arrow tycons are usually coercible tycons; all recursive and mutable tycons (e.g., lists, arrays) are incoercible tycons. To simplify the presentation, we use `pack` to serve as a representative for incoercible tycons; however, all techniques described here easily carry to other incoercible tycons.

3.2 Intermediate language: IL

We use a predicative variant [12, 14] of the polymorphic λ-calculus F_ω [10, 28] as our intermediate language. The four syntactic classes for IL, kinds (κ), constructors (μ), types (σ), and terms (e), are defined as follows:

$$\begin{array}{llll}
(kinds) & \kappa & ::= & \Omega \mid \kappa_1 \rightarrow \kappa_2 \\
(con's) & \mu & ::= & t \mid \mathtt{Int} \mid \mathtt{Real} \mid \mathtt{Pack}(\mu) \mid \rightarrow (\mu_1, \mu_2) \\
 & & \mid & \times(\mu_1, \mu_2) \mid \lambda t :: \kappa.\mu \mid \mu_1[\mu_2] \\
(types) & \sigma & ::= & T(\mu) \mid \sigma_1 \rightarrow \sigma_2 \mid \sigma_1 \times \sigma_2 \mid \forall t :: \kappa.\sigma \\
(terms) & e & ::= & x \mid i \mid f \mid \mathtt{Pk} \mid \mathtt{Upk} \mid \langle e_1, e_2 \rangle \mid \pi_1 e \mid \pi_2 e \\
 & & \mid & \lambda x : \sigma.e \mid @e_1 e_2 \mid \Lambda t :: \kappa.e \mid e[\mu] \\
(values) & v & ::= & i \mid f \mid \mathtt{Pk} \mid \mathtt{Upk} \mid \mathtt{PK}_\mu(v) \mid \langle v_1, v_2 \rangle \\
 & & \mid & \lambda x : \sigma.e \mid \Lambda t :: \kappa.e
\end{array}$$

Here, kinds classify constructors, and types classify terms. Constructors of kind Ω name monotypes. The monotypes are generated from variables, `Int`, `Real` through the constructors $\rightarrow$, $\times$, and `Pack`; here, `Pack` is the counterpart of the `pack` tycon in SRC. The application and abstraction constructors correspond to the function kind $\kappa_1 \rightarrow \kappa_2$. Types in IL include the monotypes, and are closed under products, function spaces, and polymorphic quantification. Like Harper and Morrisett [14], we use $T(\mu)$ to denote the type corresponding to the constructor μ. The terms are an explicitly typed λ-calculus with explicit constructor abstraction and application forms. The values defined here are closed and used by the operational semantics; to account for the packed values, we use $\mathtt{PK}_\mu(v)$ to denote the result of applying term $\mathtt{Pk}[\mu]$ to value v. This calculus is predicative because term expressions can only be applied to constructors (μ) but not arbitrary polymorphic types (σ).

The static semantics of IL, given in Figure 5, consists of a collection of rules for constructor formation, constructor equivalence, type formation, type equivalence, and term formation. The term formation rules are in the form of $\triangle;\Gamma \vdash e : \sigma$ where $\triangle$ is a kind environment mapping type variables to kinds, and Γ is the type environment. Apart from the standard language constructs [14], the packing primitives have the following types:

$$\begin{array}{c}
\mathtt{Pk} : \forall t :: \Omega.\, T(t) \rightarrow T(\mathtt{Pack}(t)) \\
\mathtt{Upk} : \forall t :: \Omega.\, T(\mathtt{Pack}(t)) \rightarrow T(t)
\end{array}$$

The natural (call-by-value) dynamic semantics for IL, also given in Figure 5, is defined as a set of axioms in the form of $e \hookrightarrow_i v$ where e is a closed term and v is a closed value. Because IL is very much like Harper and Mitchell's λ^{ML} [12], we can show in the similar way that type-checking for IL is decidable, and furthermore, its typing rules are consistent with the operation semantics.

3.3 Translation from SRC to IL

In Figure 6, we give a type-directed embedding of SRC into IL. This is very similar to the translation from Core-ML to XML, given by Harper and Mitchell [12]. The translation is defined as a relation $\triangle;\Gamma \vdash e_s : \tau \Rightarrow e_i$ that carries the meaning that $\triangle;\Gamma \vdash e_s : \tau$ is a derivable typing in SRC and that the translation of the SRC term e_s determined by that typing derivation is the IL term e_i. Here, the (*var*) rule turns the SRC implicit instantiation of type variables into the IL explicit type application; the (*pk/upk*) rule shows the pack/unpack terms in SRC are implemented by polymorphic primitives `Pk` and `Upk` in IL; the (*let*) rule converts the SRC `let` expressions into normal IL function applications; the rest of the rules are all straight-forward.

The translations from SRC monotypes to IL constructors (written as τ^m) and from SRC polytypes into IL types (written as σ^p) are defined as follows:

$$\begin{array}{rcl}
t^m & = & t \\
\mathtt{int}^m & = & \mathtt{Int} \\
\mathtt{real}^m & = & \mathtt{Real} \\
(\tau_1 * \tau_2)^m & = & \times(\tau_1^m, \tau_2^m) \\
(\tau_1 \rightarrow \tau_2)^m & = & \rightarrow (\tau_1^m, \tau_2^m) \\
(\tau\ \mathtt{pack})^m & = & \mathtt{Pack}(\tau^m) \\
\\
\tau^p & = & T(\tau^m) \\
(\forall t.\sigma)^p & = & \forall t :: \Omega.\sigma^p
\end{array}$$

We write $\triangle^p$ for the kind assignment mapping t to the kind Ω for each $t \in \triangle$, and Γ^p for the type environment mapping x to $(\Gamma(x))^p$ for each $x \in Dom(\Gamma)$. The following standard type preservation theorem can be proved by structural inductions on the translation rules:

Theorem 3.1 *If* $\triangle;\Gamma \vdash e_s : \tau \Rightarrow e_i$, *then* $\triangle^p;\Gamma^p \vdash e_i : \tau^p$.

Constructor Formation and Constructor Equivalence:

(*tvar/tcon*)
$$\frac{}{\triangle \uplus \{t :: \kappa\} \triangleright t :: \kappa} \qquad \frac{}{\triangle \triangleright \mathtt{Int} :: \Omega} \qquad \frac{}{\triangle \triangleright \mathtt{Real} :: \Omega}$$

(*pk/fn/pair*)
$$\frac{\triangle \triangleright \mu :: \Omega}{\triangle \triangleright \mathtt{Pack}(\mu) :: \Omega} \qquad \frac{\triangle \triangleright \mu_1 :: \Omega \quad \triangle \triangleright \mu_2 :: \Omega}{\triangle \triangleright \times(\mu_1, \mu_2) :: \Omega} \qquad \frac{\triangle \triangleright \mu_1 :: \Omega \quad \triangle \triangleright \mu_2 :: \Omega}{\triangle \triangleright \rightarrow(\mu_1, \mu_2) :: \Omega}$$

(*cfn/capp*)
$$\frac{\triangle \uplus \{t :: \kappa_1\} \triangleright \mu :: \kappa_2}{\triangle \triangleright \Lambda t :: \kappa_1 . \mu :: \kappa_1 \rightarrow \kappa_2} \qquad \frac{\triangle \triangleright \mu_1 :: \kappa' \rightarrow \kappa \quad \triangle \triangleright \mu_2 :: \kappa'}{\triangle \triangleright \mu_1[\mu_2] :: \kappa}$$

(*cequiv*)
$$\frac{\triangle \uplus \{t :: \kappa'\} \triangleright \mu_1 :: \kappa \quad \triangle \triangleright \mu_2 :: \kappa'}{\triangle \triangleright (\Lambda t :: \kappa' . \mu_1)[\mu_2] \equiv [\mu_2/t]\mu_1 :: \kappa}$$

Type Formation and Type Equivalence:

(*tform*)
$$\frac{\triangle \triangleright \mu :: \Omega}{\triangle \triangleright T(\mu)} \qquad \frac{\triangle \triangleright \sigma_1 \quad \triangle \triangleright \sigma_2}{\triangle \triangleright \sigma_1 \times \sigma_2} \qquad \frac{\triangle \triangleright \sigma_1 \quad \triangle \triangleright \sigma_2}{\triangle \triangleright \sigma_1 \rightarrow \sigma_2} \qquad \frac{\triangle \uplus \{t :: \kappa\} \triangleright \sigma}{\triangle \triangleright \forall t :: \kappa . \sigma}$$

(*tequiv*)
$$\frac{\triangle \triangleright \mu_1 :: \Omega \quad \triangle \triangleright \mu_2 :: \Omega}{\triangle \triangleright T(\times(\mu_1, \mu_2)) \equiv T(\mu_1) \times T(\mu_2)} \qquad \frac{\triangle \triangleright \mu_1 :: \Omega \quad \triangle \triangleright \mu_2 :: \Omega}{\triangle \triangleright T(\rightarrow(\mu_1, \mu_2)) \equiv T(\mu_1) \rightarrow T(\mu_2)}$$

Term Formation:

(*const/var*)
$$\frac{}{\triangle; \Gamma \vdash i : \mathtt{Int}} \qquad \frac{}{\triangle; \Gamma \vdash f : \mathtt{Real}} \qquad \frac{}{\triangle; \Gamma \vdash x : \Gamma(x)}$$

(*pk/upk*)
$$\frac{}{\triangle; \Gamma \vdash \mathtt{Pk} : \forall t :: \Omega . T(t) \rightarrow T(\mathtt{Pack}(t))} \qquad \frac{}{\triangle; \Gamma \vdash \mathtt{Upk} : \forall t :: \Omega . T(\mathtt{Pack}(t)) \rightarrow T(t)}$$

(*pair/ith*)
$$\frac{\triangle; \Gamma \vdash e_1 : \sigma_1 \quad \triangle; \Gamma \vdash e_2 : \sigma_2}{\triangle; \Gamma \vdash \langle e_1, e_2 \rangle : \sigma_1 \times \sigma_2} \qquad \frac{\triangle; \Gamma \vdash e : \sigma_1 \times \sigma_2}{\triangle; \Gamma \vdash \pi_i e : \sigma_i} \quad (i = 1, 2)$$

(*fn/app*)
$$\frac{\triangle; \Gamma \uplus \{x : \sigma_1\} \vdash e : \sigma_2}{\triangle; \Gamma \vdash \lambda x : \sigma_1 . e : \sigma_1 \rightarrow \sigma_2} \qquad \frac{\triangle; \Gamma \vdash e_1 : \sigma' \rightarrow \sigma \quad \triangle; \Gamma \vdash e_2 : \sigma'}{\triangle; \Gamma \vdash @e_1 e_2 : \sigma}$$

(*tfn/tapp*)
$$\frac{\triangle \uplus \{t : \kappa\}; \Gamma \vdash e : \sigma}{\triangle; \Gamma \vdash \Lambda t :: \kappa . e : \forall t :: \kappa . \sigma} \qquad \frac{\triangle \triangleright \mu :: \kappa \quad \triangle; \Gamma \vdash e : \forall t :: \kappa . \sigma}{\triangle; \Gamma \vdash e[\mu] : [\mu/t]\sigma}$$

Natural Dynamic Semantics:

$$\frac{}{v \hookrightarrow_i v} \qquad \frac{e_1 \hookrightarrow_i \lambda x : \sigma' . e \quad e_2 \hookrightarrow_i v' \quad [v'/x]e \hookrightarrow_i v}{@e_1 e_2 \hookrightarrow_i v} \qquad \frac{e_1 \hookrightarrow_i \Lambda t :: \kappa . e' \quad [\mu/t]e' \hookrightarrow_i v}{e[\mu] \hookrightarrow_i v}$$

$$\frac{e \hookrightarrow_i v}{@(\mathtt{Pk}[\mu])(e) \hookrightarrow_i \mathtt{PK}_\mu(v)} \qquad \frac{e \hookrightarrow_i \mathtt{PK}_\mu(v)}{@(\mathtt{Upk}[\mu])(e) \hookrightarrow_i v} \qquad \frac{e_1 \hookrightarrow_i v_1 \quad e_2 \hookrightarrow_i v_2}{\langle e_1, e_2 \rangle \hookrightarrow_i \langle v_1, v_2 \rangle} \qquad \frac{e \hookrightarrow_i \langle v_1, v_2 \rangle}{\pi_j e \hookrightarrow_i v_j} \quad (j = 1, 2)$$

Figure 5: The Static and Dynamic Semantics for IL

$$(const) \quad \frac{}{\triangle;\Gamma \vdash i : \texttt{int} \Rightarrow i} \qquad \frac{}{\triangle;\Gamma \vdash f : \texttt{real} \Rightarrow f}$$

$$(var) \quad \frac{\Gamma(x) = \forall t_1 \ldots t_n.\tau \quad \rho = \{t_i \mapsto \tau_i \mid i = 1, \ldots, n\} \quad FTV(\rho(\tau)) \subseteq \triangle}{\triangle;\Gamma \vdash x : \rho(\tau) \Rightarrow x[\tau_1^m]\cdots[\tau_n^m]}$$

$$(pk/upk) \quad \frac{\triangle;\Gamma \vdash e : \tau \Rightarrow e'}{\triangle;\Gamma \vdash \texttt{pk}\ e : \tau\ \texttt{pack} \Rightarrow @(\texttt{Pk}[\tau^m])(e')} \qquad \frac{\triangle;\Gamma \vdash e : \tau\ \texttt{pack} \Rightarrow e'}{\triangle;\Gamma \vdash \texttt{upk}\ e : \tau \Rightarrow @(\texttt{Upk}[\tau^m])(e')}$$

$$(pair/ith) \quad \frac{\triangle;\Gamma \vdash e_1 : \tau_1 \Rightarrow e_1' \quad \triangle;\Gamma \vdash e_2 : \tau_2 \Rightarrow e_2'}{\triangle;\Gamma \vdash \langle e_1, e_2\rangle : \tau_1 * \tau_2 \Rightarrow \langle e_1', e_2'\rangle} \qquad \frac{\triangle;\Gamma \vdash e : \tau_1 * \tau_2 \Rightarrow e'}{\triangle;\Gamma \vdash \pi_i e : \tau_i \Rightarrow \pi_i e'} \quad (i = 1, 2)$$

$$(fn/app) \quad \frac{\triangle;\Gamma \uplus \{x : \tau_1\} \vdash e : \tau_2 \Rightarrow e'}{\triangle;\Gamma \vdash \lambda x.e : \tau_1 \rightarrow \tau_2 \Rightarrow \lambda x : \tau_1^p.e'} \qquad \frac{\triangle;\Gamma \vdash e_1 : \tau' \rightarrow \tau \Rightarrow e_1' \quad \triangle;\Gamma \vdash e_2 : \tau' \Rightarrow e_2'}{\triangle;\Gamma \vdash e_1 e_2 : \tau \Rightarrow @e_1' e_2'}$$

$$(let) \quad \frac{\triangle \uplus \{t_1, \ldots, t_n\};\Gamma \vdash v : \tau_0 \Rightarrow e_1' \quad \sigma_0 = \forall t_1, \ldots, t_n.\tau_0 \quad \triangle;\Gamma \uplus \{x : \sigma_0\} \vdash e : \tau \Rightarrow e_2'}{\triangle;\Gamma \vdash \texttt{let}\ x = v\ \texttt{in}\ e : \tau \Rightarrow @(\lambda x : \sigma_0^p.e_2')(\Lambda t_1 :: \Omega, \ldots, \Lambda t_n :: \Omega.e_1')}$$

Figure 6: Translation from SRC to IL

A semantic correctness theorem about this embedding can also be stated and proved using the standard logical-relations technique [24, 19].

3.4 Target language: TGT

The target language (TGT) for our representation analysis is very similar to Harper and Morrisett's λ_i^{ML} [14]. In fact, because our IL is much like λ^{ML}, TGT is essentially the previously defined IL plus Harper and Morrisett's *typerec* forms [14]. The four syntactic classes for TGT, kinds (κ), constructors (μ), types (σ), and terms (e), are defined as follows:

(*kinds*)	κ	::=	... *same as in IL* ...
(*con's*)	μ	::=	... *IL constructors* ... $\mid \texttt{Boxed}(\mu)$
		$\mid$	$\texttt{Typerec}\ \mu\ \texttt{of}\ (\mu_i \mid \mu_r \mid \mu_\rightarrow \mid \mu_\times \mid \mu_p \mid \mu_b)$
(*types*)	σ	::=	... *same as in IL* ...
(*terms*)	e	::=	... *IL terms* ... $\mid \texttt{box} \mid \texttt{unbox}$
		$\mid$	$\texttt{typerec}\ \mu\ \texttt{of}\ [t.\sigma](e_i \mid e_r \mid e_\rightarrow \mid e_\times \mid e_p \mid e_b)$
(*values*)	v	::=	... *IL values* ... $\mid \texttt{box} \mid \texttt{unbox} \mid \texttt{BX}_\mu(v)$

Here, the `Typerec` and `typerec` forms are the keys for *intensional type analysis* [14]. They provide the ability to define new constructors and terms by structural induction on monotypes. Also, `Boxed` is a new special primitive boxing constructor; `box` and `unbox` are two new term-level boxing primitives with the following types:

$$\texttt{box} : \forall t :: \Omega.\, T(t) \rightarrow T(\texttt{Boxed}(t))$$
$$\texttt{unbox} : \forall t :: \Omega.\, T(\texttt{Boxed}(t)) \rightarrow T(t)$$

Intuitively, $\texttt{Boxed}[\mu]$ denotes the boxed version of a given constructor (monotype) μ. We intentionally treat these as primitives because how they are actually implemented does not affect the correctness of our representation analysis.

As in IL, the values defined here are closed and used in the operational semantics; we use $\texttt{BX}_\mu(v)$ to denote the boxed value of applying term $\texttt{box}[\mu]$ to value v. The static semantics for TGT is almost identical to those for IL in Figure 5 (the TR version of this paper [30] contains the details); the additional rules for `Typerec` and `typerec` are same as those in Harper and Morrisett [14]. The operational semantics for TGT, written as $e \hookrightarrow_t v$, is similar to those for IL also. Harper and Morrisett [14, 24] have shown that type-checking for λ_i^{ML} is decidable, and furthermore, its typing rules are consistent with the operation semantics. Similar results hold for our target language TGT.

3.5 Translation from IL to TGT

Now that we have translated SRC into IL and given the syntax and semantics of TGT, representation analysis can be expressed as a type-directed transformation from IL programs to TGT programs. This translation is defined as a relation $\triangle;\Gamma \vdash e_i : \tau \Rightarrow e_t$ that carries the meaning that $\triangle;\Gamma \vdash e_i : \tau$ is a derivable typing in IL and that the translation of the IL term e_i determined by that typing derivation is the TGT term e_t.

We begin by introducing a special constructor $\texttt{Wrap} :: \Omega \rightarrow \Omega$ and a pair of primitive operations:

$$\texttt{wrap} : \forall t :: \Omega.\, T(t) \rightarrow T(\texttt{Wrap}[t])$$
$$\texttt{unwrap} : \forall t :: \Omega.\, T(\texttt{Wrap}[t]) \rightarrow T(t)$$

Intuitively, $\texttt{Wrap}[\mu]$ denotes the canonical boxed form of a given constructor (monotype) μ; `wrap` and `unwrap` are exactly those primitive coercion operations mentioned in Section 2.1. We intentionally delay their definitions in order to show that the representation analysis framework described here does not depend on any particular wrapping schemes. Both the constructor and the primitives will be defined later in Section 3.6, using `Typerec`, `typerec`, and the basic boxing primitives `box` and `unbox`.

The translation from IL constructors to TGT constructors, written as μ^u, is defined as follows:

$$(base)\quad \frac{}{\triangle;\Gamma \vdash x : \Gamma(x) \Rightarrow x} \qquad \frac{}{\triangle;\Gamma \vdash c : _ \Rightarrow c} \quad (c = i, f, \texttt{Pk}, \texttt{Upk})$$

$$(pa/pi)\quad \frac{\triangle;\Gamma \vdash e_1 : \sigma_1 \Rightarrow e_1' \quad \triangle;\Gamma \vdash e_2 : \sigma_2 \Rightarrow e_2'}{\triangle;\Gamma \vdash \langle e_1, e_2\rangle : \sigma_1 \times \sigma_2 \Rightarrow \langle e_1', e_2'\rangle} \qquad \frac{\triangle;\Gamma \vdash e : \sigma_1 \times \sigma_2 \Rightarrow e'}{\triangle;\Gamma \vdash \pi_i e : \sigma_i \Rightarrow \pi_i e'} \quad (i = 1, 2)$$

$$(fn/ap)\quad \frac{\triangle;\Gamma \uplus \{x : \sigma_1\} \vdash e : \sigma_2 \Rightarrow e'}{\triangle;\Gamma \vdash \lambda x : \sigma_1.e : \sigma_1 \to \sigma_2 \Rightarrow \lambda x : \sigma_1^u.e'} \qquad \frac{\triangle;\Gamma \vdash e_1 : \sigma' \to \sigma \Rightarrow e_1' \quad \triangle;\Gamma \vdash e_2 : \sigma' \Rightarrow e_2'}{\triangle;\Gamma \vdash @e_1 e_2 : \sigma \Rightarrow @e_1' e_2'}$$

$$(tfn)\quad \frac{\triangle \uplus \{t :: \kappa\};\Gamma \vdash e : \sigma \Rightarrow e'}{\triangle;\Gamma \vdash \Lambda t :: \kappa.e : \forall t :: \kappa.\sigma \Rightarrow \Lambda t : \kappa.e'}$$

$$(tapp)\quad \frac{\triangle;\Gamma \vdash e : \forall t :: \kappa.\sigma \Rightarrow e' \quad e_c' = \mathcal{C}_C([\mu^w/t]\sigma^u, [\mu^u/t]\sigma^u)}{\triangle;\Gamma \vdash e[\mu] : [\mu/t]\sigma \Rightarrow @e_c'(e'[\mu^w])}$$

Figure 7: Translation from IL to TGT

$$\begin{array}{rcll}
\mathcal{C}_C(\times(\mu_1', \mu_2'), \times(\mu_1, \mu_2)) & = & \mathcal{C}_T(T(\mu_1') \times T(\mu_2'), T(\mu_1) \times T(\mu_2)) & \\
\mathcal{C}_C(\to(\mu_1', \mu_2'), \to(\mu_1, \mu_2)) & = & \mathcal{C}_T(T(\mu_1') \to T(\mu_2'), T(\mu_1) \to T(\mu_2)) & \\
\mathcal{C}_C(\mu', \mu) & = & \texttt{identity}[\mu] & \text{if } \mu \equiv \mu' \text{ and } \triangle \rhd \mu :: \Omega \\
\mathcal{C}_C(\texttt{Wrap}[\mu], \mu) & = & \texttt{unwrap}[\mu] & \\
\mathcal{C}_C(\mu, \texttt{Wrap}[\mu]) & = & \texttt{wrap}[\mu] & \\
 & & & \\
\mathcal{C}_T(T(\mu'), T(\mu)) & = & \mathcal{C}_C(\mu', \mu) & \\
\mathcal{C}_T(\forall t :: \kappa.\sigma', \forall t :: \kappa.\sigma) & = & \lambda f : \forall t :: \kappa.\sigma'.(\Lambda t :: \kappa.(@(\mathcal{C}_T(\sigma', \sigma))(f[t]))) & \\
\mathcal{C}_T(\sigma_1' \times \sigma_2', \sigma_1 \times \sigma_2) & = & \lambda x : \sigma_1' \times \sigma_2'.\ \langle @(\mathcal{C}_T(\sigma_1', \sigma_1))(\pi_1 x), @(\mathcal{C}_T(\sigma_2', \sigma_2))(\pi_2 x)\rangle & \\
\mathcal{C}_T(\sigma_1' \to \sigma_2', \sigma_1 \to \sigma_2) & = & \lambda f : \sigma_1' \to \sigma_2'.\lambda x : \sigma_1.\ (@(\mathcal{C}_T(\sigma_2', \sigma_2))(@(f)(@(\mathcal{C}_T(\sigma_1, \sigma_1'))(x)))) &
\end{array}$$

Figure 8: The Coercion Generator

$$\begin{array}{rcl}
t^u & = & t \\
\texttt{Int}^u & = & \texttt{Int} \\
\texttt{Real}^u & = & \texttt{Real} \\
(\times(\mu_1, \mu_2))^u & = & \times(\mu_1^u, \mu_2^u) \\
(\to(\mu_1, \mu_2))^u & = & \to(\mu_1^u, \mu_2^u) \\
(\texttt{Pack}(\mu))^u & = & \texttt{Pack}(\texttt{Wrap}[\mu]) \\
(\Lambda t :: \kappa.\mu)^u & = & \Lambda t :: \kappa.\mu^u \\
(\mu_1[\mu_2])^u & = & \mu_1^u[\mu_2^u]
\end{array}$$

Notice this translation is almost a trivial identity mapping except that *incoercible* tycons such as `Pack` are treated specially: the element type μ is translated into a "wrapped" version of itself. Incoercible data structures (in TGT) will not change its representations when switching between the monomorphic and polymorphic contexts.

The translation from IL types to TGT types, written as σ^u, is just a simple extension of μ^u:

$$\begin{array}{rcl}
T(\mu)^u & = & T(\mu^u) \\
(\forall t :: \kappa.\sigma)^u & = & \forall t :: \kappa.\sigma^u \\
(\sigma_1 \times \sigma_2)^u & = & \sigma_1^u \times \sigma_2^u \\
(\sigma_1 \to \sigma_2)^u & = & \sigma_1^u \to \sigma_2^u
\end{array}$$

Finally, to make the presentation easier, we use μ^w to denote a "wrapped" version of the above translation μ^u. For any IL constructor μ, μ^w applies the `Wrap` constructor to the result of μ^u:

$$\begin{array}{rcll}
\mu^w & = & \texttt{Wrap}[\mu^u] & \text{if } \triangle \rhd \mu :: \Omega \\
(\Lambda t :: \kappa.\mu)^w & = & \Lambda t :: \kappa.\mu^w & \\
\mu^w & = & (\Lambda t :: \kappa.\mu[t])^w & \text{if } \triangle \rhd \mu :: \kappa \to \kappa'
\end{array}$$

Notice when μ is a constructor with higher-order kinds, "wrapping" is performed recursively on the body of the constructor.

Given a substitution ρ mapping t to IL constructors, we write ρ^u and ρ^w as the new substitutions mapping t to $(\rho(t))^u$ and $(\rho(t))^w$ for each $t \in Dom(\rho)$. We also write $\triangle^u$ for the kind assignment mapping t to the same kind $\triangle(t)$ for each $t \in Dom(\triangle)$, and Γ^u for the type environment mapping x to $(\Gamma(x))^u$ for each $x \in Dom(\Gamma)$.

We define a set of axioms that characterize the valid canonical boxing schemes and give two propositions about the type translation. Both propositions can be proved by inductions on the syntactic structures of the IL constructors and types.

Definition 3.2 (valid canonical boxed form) *Given a constructor definition for* `Wrap` *in TGT, we say it is in the* valid canonical boxed form *if it commutes with substitution and the following constructor equivalence rules can be derived in TGT:*

$$\begin{array}{rcl}
\texttt{Wrap}[\mu] & \equiv & \texttt{Wrap}[\texttt{Wrap}[\mu]] \\
\texttt{Wrap}[\times(\mu_1, \mu_2)] & \equiv & \texttt{Wrap}[\times(\texttt{Wrap}[\mu_1], \texttt{Wrap}[\mu_2])] \\
\texttt{Wrap}[\to(\mu_1, \mu_2)] & \equiv & \texttt{Wrap}[\to(\texttt{Wrap}[\mu_1], \texttt{Wrap}[\mu_2])]
\end{array}$$

More generally, if SRC contains more coercible constructors such as χ with arity k ($k > 0$), and χ' is its counterpart in IL and TGT, then $\texttt{Wrap}[\chi'(\mu_1, \ldots, \mu_k)]$ *must be equivalent to* $\texttt{Wrap}[\chi'(\texttt{Wrap}[\mu_1], \ldots, \texttt{Wrap}[\mu_k])]$.

Proposition 3.3 *If* `Wrap` *is in the valid canonical boxed form, then both μ^u and σ^u commute with substitution. More*

specifically, suppose ρ is a substitution mapping t to IL constructors, μ is an IL constructor, and σ is an IL type, then $(\rho(\mu))^u \equiv \rho^u(\mu^u)$ and $(\rho(\sigma))^u \equiv \rho^u(\sigma^u)$.

Proposition 3.4 *If* `Wrap` *is in the valid canonical boxed form, then μ^w commutes with substitution. More specifically, suppose ρ is a substitution mapping t to IL constructors and μ is an IL constructor, then $(\rho(\mu))^w \equiv \rho^w(\mu^w)$.*

The term translation rules are given in Figure 7 as a series of inference rules that parallel the typing rules for IL. Most of these rules are straight-forward. The only interesting case is the (*tapp*) rule; this is the place where representation coercions are inserted. Notice we always "wrap" the argument constructor μ before doing the type application; this reflects the invariant that all TGT values with unknown type are always wrapped. The coercion term e'_c is generated *at compile time* by the coercion generators $\mathcal{C}_\mathcal{C}$ and $\mathcal{C}_\mathcal{T}$ defined in Figure 8. Here, `identity` is the polymorphic identity function. Given a pair of TGT types σ' and σ, the result of $\mathcal{C}_\mathcal{T}(\sigma', \sigma)$ is a TGT term that coerces values of type σ' into those of type σ; similarly, given a pair of TGT constructors μ' and μ, $\mathcal{C}_\mathcal{C}(\mu', \mu)$ returns a TGT term that coerces values of type $T(\mu')$ into those of type $T(\mu)$. In essence, our coercion generator plays the same role as Leroy [19]'s S and G transformations. The following proposition shows that the coercions generated above remedy the type mismatches caused by constructor wrapping during type application:

Proposition 3.5 *Suppose* `Wrap` *is in the valid canonical boxed form, ρ is a substitution mapping t to IL constructors, σ is an IL type. If $\triangle; \Gamma \vdash x : \rho^w(\sigma^u)$, then $\triangle; \Gamma \vdash @(\mathcal{C}_\mathcal{T}(\rho^w(\sigma^u), \rho^u(\sigma^u)))(x) : \rho^u(\sigma^u)$; Similarly, if $\triangle; \Gamma \vdash x : \rho^u(\sigma^u)$, then $\triangle; \Gamma \vdash @(\mathcal{C}_\mathcal{T}(\rho^u(\sigma^u), \rho^w(\sigma^u)))(x) : \rho^w(\sigma^u)$;*

Theorem 3.6 (type preservation) *If* `Wrap` *is in the valid canonical boxed form and $\triangle; \Gamma \vdash e_i : \sigma \Rightarrow e_t$, then $\triangle^u; \Gamma^u \vdash e_t : \sigma^u$.*

The type preservation theorem can be proved by structural inductions on the translation rules, using Proposition 3.3–3.5. A semantic correctness theorem about the translation can also be stated and proved using the same logical-relations technique used in Morrisett [24] and Leroy [19].

3.6 Valid canonical boxing schemes

A boxing scheme is *valid* if its underlying constructor `Wrap` is in the valid canonical boxed form. Intuitively, the set of axioms in the Definition 3.2 guarantees that coercions based on the `Wrap` will commute with the type instantiation relations.

The boxing schemes described in Section 2, simple boxing, partial boxing, and full boxing, can be formally modelled by defining the constructor `Wrap` and the primitives `wrap` and `unwrap`. More sophisticated partial boxing schemes such as those described in Section 2.4 can be defined in the similar way.

Definition 3.7 (simple boxing) *The constructor* `Wrap` *for simple boxing is defined as follows (here we use the pattern-match clauses to express the* `Typerec` *form):*

$$\begin{array}{rcl}
\texttt{Wrap}[\mu] & = & \texttt{Boxed}(\texttt{Uncv}[\mu]) \\
\\
\texttt{Uncv}[\texttt{Int}] & = & \texttt{Int} \\
\texttt{Uncv}[\texttt{Real}] & = & \texttt{Real} \\
\texttt{Uncv}[\texttt{Pack}(\mu)] & = & \texttt{Pack}(\mu) \\
\texttt{Uncv}[\texttt{Boxed}(\mu)] & = & \mu \\
\texttt{Uncv}[\times(\mu_1, \mu_2)] & = & \times(\texttt{Uncv}[\mu_1], \texttt{Uncv}[\mu_2]) \\
\texttt{Uncv}[\rightarrow(\mu_1, \mu_2)] & = & \rightarrow(\texttt{Uncv}[\mu_1], \texttt{Uncv}[\mu_2])
\end{array}$$

Definition 3.8 (partial boxing) *The constructor* `Wrap` *for partial boxing is defined in the same way as simple boxing, except that the rule for* $\texttt{Uncv}[\rightarrow(\mu_1, \mu_2)]$ *is replaced by the following:*

$$\texttt{Uncv}[\rightarrow(\mu_1, \mu_2)] = \rightarrow(\texttt{Boxed}(\texttt{Uncv}[\mu_1]), \texttt{Boxed}(\texttt{Uncv}[\mu_2]))$$

Definition 3.9 (full boxing) *The constructor* `Wrap` *for full boxing is defined in the same way as partial boxing, except that the rule for* $\texttt{Uncv}[\times(\mu_1, \mu_2)]$ *is replaced by the following:*

$$\texttt{Uncv}[\times(\mu_1, \mu_2)] = \times(\texttt{Boxed}(\texttt{Uncv}[\mu_1]), \texttt{Boxed}(\texttt{Uncv}[\mu_2]))$$

Proposition 3.10 *The simple-boxing constructor* `Wrap` *is in the valid canonical boxed form.*

Proof First, $\texttt{Wrap}[\mu]$ commutes with substitutions because $\texttt{Wrap}[\mu] = \texttt{Boxed}(\texttt{Uncv}[\mu])$, and both `Boxed` and `Uncv` commute with substitutions; `Boxed` is a primitive constructor, and `Uncv` is a simple `Typerec`-form constructor (which can be shown to commute with substitutions using structural inductions, following from the constructor equivalence rules on `Typerec` defined in Harper and Morrisett [14, 24]).

Second, we prove $\texttt{Wrap}[\mu] \equiv \texttt{Wrap}[\texttt{Wrap}[\mu]]$. From the definition of `Uncv`, we have $\texttt{Uncv}[\texttt{Boxed}(\mu)] \equiv \mu$ for any μ, therefore:

$$\begin{array}{rl}
& \texttt{Wrap}[\texttt{Wrap}[\mu]] \\
\equiv & \texttt{Boxed}(\texttt{Uncv}[\texttt{Boxed}(\texttt{Uncv}[\mu])]) \\
\equiv & \texttt{Boxed}(\texttt{Uncv}[\mu]) \\
\equiv & \texttt{Wrap}[\mu].
\end{array}$$

To prove $\texttt{Wrap}[\times(\mu_1, \mu_2)] \equiv \texttt{Wrap}[\times(\texttt{Wrap}[\mu_1], \texttt{Wrap}[\mu_2])]$, we notice that

$$\texttt{Uncv}[\texttt{Wrap}[\mu]] \equiv \texttt{Uncv}[\texttt{Boxed}(\texttt{Uncv}[\mu])] \equiv \texttt{Uncv}[\mu]$$

holds for any constructor μ, thus

$$\begin{array}{rl}
& \texttt{Wrap}[\times(\texttt{Wrap}[\mu_1], \texttt{Wrap}[\mu_2])] \\
\equiv & \texttt{Boxed}(\texttt{Uncv}[\times(\texttt{Wrap}[\mu_1], \texttt{Wrap}[\mu_2])]) \\
\equiv & \texttt{Boxed}(\times(\texttt{Uncv}[\texttt{Wrap}[\mu_1]], \texttt{Uncv}[\texttt{Wrap}[\mu_2]])) \\
\equiv & \texttt{Boxed}(\times(\texttt{Uncv}[\mu_1], \texttt{Uncv}[\mu_2])) \\
\equiv & \texttt{Boxed}(\texttt{Uncv}[\times(\mu_1, \mu_2)]) \\
\equiv & \texttt{Wrap}[\times(\mu_1, \mu_2)].
\end{array}$$

Finally, $\texttt{Wrap}[\rightarrow(\mu_1, \mu_2)] \equiv \texttt{Wrap}[\rightarrow(\texttt{Wrap}[\mu_1], \texttt{Wrap}[\mu_2])]$ can be proved in the same way. **QED**.

Proposition 3.11 *The partial-boxing constructor* `Wrap` *is in the valid canonical boxed form.*

Proposition 3.12 *The full-boxing constructor* `Wrap` *is in the valid canonical boxed form.*

Proposition 3.11 and 3.12 can be proved in the same way as for Proposition 3.10. From the definition of partial boxing and full boxing, we can easily derive:

$$\begin{aligned}&\texttt{Uncv}[\rightarrow(\texttt{Boxed}(\mu_1),\texttt{Boxed}(\mu_2))]\\ \equiv\quad &\rightarrow(\texttt{Boxed}(\mu_1),\texttt{Boxed}(\mu_2))\end{aligned}$$

This means that partial boxing and full boxing do not need to coerce arguments or results of unknown function types, so they completely avoid the vararg problem.

Next, we give the definitions of the coercion primitives (`wrap` and `unwrap`) for simple boxing:

$$\begin{array}{lrcl}
(1) & \texttt{wrap}[\mu] & = & \texttt{box}[\texttt{Uncv}[\mu]]\circ\texttt{uncv}[\mu]\\
(2) & \texttt{unwrap}[\mu] & = & \texttt{cover}[\mu]\circ\texttt{unbox}[\texttt{Uncv}[\mu]]\\
\\
(3) & \texttt{uncv}[\texttt{Boxed}(\mu)] & = & \texttt{unbox}[\mu]\\
(4) & \texttt{uncv}[\times(\mu_1,\mu_2)] & = & \texttt{uncv}[\mu_1]\times\texttt{uncv}[\mu_2]\\
(5) & \texttt{uncv}[\rightarrow(\mu_1,\mu_2)] & = & \texttt{cover}[\mu_1]\rightarrow\texttt{uncv}[\mu_2]\\
(6) & \texttt{uncv}[\mu] & = & \texttt{identity}[\mu]\\
\\
(7) & \texttt{cover}[\texttt{Boxed}(\mu)] & = & \texttt{box}[\mu]\\
(8) & \texttt{cover}[\times(\mu_1,\mu_2)] & = & \texttt{cover}[\mu_1]\times\texttt{cover}[\mu_2]\\
(9) & \texttt{cover}[\rightarrow(\mu_1,\mu_2)] & = & \texttt{uncv}[\mu_1]\rightarrow\texttt{cover}[\mu_2]\\
(10) & \texttt{cover}[\mu] & = & \texttt{identity}[\mu]
\end{array}$$

Here, the pattern-match syntax is representing the the term-level `typerec` form; also, $\circ$ denotes function composition, `identity` is the polymorphic identity function, and product and function spaces are extended to functions in the usual way.

The coercion primitives for partial boxing and full boxing can be defined in the same way. For example, the definition for partial boxing can be obtained by replacing rule (5) and (9) with the following:

$$\begin{array}{ll}
(5') & \texttt{uncv}[\rightarrow(\mu_1,\mu_2)] = (\texttt{cover}[\mu_1]\circ\texttt{unbox}[\texttt{Uncv}[\mu_1]])\\
& \qquad\qquad\rightarrow(\texttt{box}[\texttt{Uncv}[\mu_2]]\circ\texttt{uncv}[\mu_2])
\end{array}$$

$$\begin{array}{ll}
(9') & \texttt{cover}[\rightarrow(\mu_1,\mu_2)] = (\texttt{box}[\texttt{Uncv}[\mu_1]]\circ\texttt{uncv}[\mu_1])\\
& \qquad\qquad\rightarrow(\texttt{cover}[\mu_2]\circ\texttt{unbox}[\texttt{Uncv}[\mu_2]])
\end{array}$$

Notice under partial boxing, the `uncv` and `cover` primitives are equivalent to the `identity` function on constructors such as $\rightarrow(\texttt{Boxed}(\mu_1),\texttt{Boxed}(\mu_2))$.

4 Extensions

In this section, we present several extensions and variations of our flexible representation analysis algorithm.

4.1 Modules and type abstractions

The algorithm and framework in Section 3 can be extended to handle the entire SML language [23] plus the MacQueen-Tofte style higher-order modules [21]. In a companion paper [32], we show that both the SML simple modules and the transparent higher-order modules can all be translated into our intermediate language IL defined in Section 3.2 (extended with product kinds). Therefore, representation analysis on the module languages is reduced to calling the same algorithm described in Section 3.5.

To handle module-level type abstractions, we treat all abstract tycons as incoercible tycons, just like what we did for lists and arrays. Coercions between concrete and abstract types are inserted at the places where abstractions are introduced. For example, in the following ML code:

```
signature SIG = sig type 'a t
                    val p : real t
                    val f : 'a t -> 'a t
                end

functor F(S : SIG) = struct val r = S.f(S.p) end
```

The parameter structure for functor F contains an abstract type constructor `t` of arity 1. Inside the body of the functor, value `S.p` needs to be coerced from `real t` into `'a t`, but since `t` is incoercible, the coercion is just identity functions. When `F` is applied to the following structure `A`:

```
structure A = struct type 'a t = 'a * 'a
                     val p = (3.0,3.0)
                     fun f (x,y) = (y,x)
              end

structure T = F(A)
```

both `p` and `f` would first be wrapped into canonical boxed form; later, after the functor application is done, the `r` field in the result must be unwrapped back to unboxed form.

4.2 Concrete vs. abstract datatypes

Consistent datatype representations across functor boundary has long been a tricky problem for ML compiler writers [2]. Flexible representation analysis offers a nice solution. Consider the following ML structure declaration:

```
structure S =
  struct
    datatype t = FCONS of (real -> real) * t
               | FNIL
    type u = real
    type s = (real -> int) * t
    val v = (fn x => x+1.0, FNIL)
    val x = FCONS v
  end
```

Here, t is a recursive but monomorphic datatype; because it is monomorphic, one would not think about inserting coercions when applying the injection function of constructor `FCONS` to value v. This is, unfortunately, incorrect; because when structure S is passed to the following functor F, the above concrete datatype t can match any similar abstract datatype definitions:

```
  functor F(A : sig type u
                    datatype t
                      = FCONS of (u -> real) * t
                      | FNIL
                end) = struct ... end
```

Here, the body of functor F will not know the actual representation of `u` until the functor application time. The solution is to consistently apply a "wrapping" (or "unwrapping") operation while injecting (or projecting) the value v into datatype t. This wrapping ensures that all values carried by the datatype (concrete or abstract) will have consistent data representations (i.e., they are all in the canonical boxed form).

The technique described above can also be used to solve the classical list representation problem [2]. This problem occurs when we apply the following functor G to structure S:

```
functor G(B : sig type s
                  datatype t = FCONS of s
                             | FNIL
              end) = struct ... end
```

Most existing compilers represent the *cons* cell as an untagged record with no indirections. Here, however, because type s is unknown, we cannot use the untagged record to represent the cons cell. The solution is to introduce a new type-dependent operator which checks at runtime to see if s is indeed a record type, if that is the case, no indirection is inserted; otherwise, an extra boxing layer is added.

4.3 Parametricity and type passing

One interesting twist about our scheme is that most of the runtime type-passing (and also coercions that depend on runtime types) can be eliminated based on a parametricity property for polymorphic functions. Consider the following example:

```
fun f(x) = let ... x::nil ...
               ... Array.update(a,i,x) ...
            in (x, x)
           end

fun g(y) = f(y, 1.0)
```

Here function `f` has type $\forall\alpha.\alpha \rightarrow (\alpha * \alpha)$; inside `f`, the argument `x` is *cons*-ed onto a list at one time and put inside an array at another time. Function `g` has type $\forall\beta.\beta \rightarrow ((\beta*\texttt{real})*(\beta*\texttt{real}))$. During the function application `f(y,1.9)`, the argument normally should be coerced by $\texttt{wrap}[\beta * \texttt{real}]$, which depends on the runtime type β.

In fact, neither `f` nor `g` really needs to know about α and β at runtime; they would not take them as extra parameters, and values of type α or β can be treated as single-word black boxes (as in the full-boxing approach). The coercion $\texttt{wrap}[\beta * \texttt{real}]$ does not need to examine and uncover β as well. This is all right because we can tell from the types of `f` and `g` (which do not contain any list and array tycons) that none of the polymorphic lists and arrays involving α (or β) will ever be exported outside function `f` (or `g`).

This observation can be made more precise as follows: given a polymorphic type $\forall\alpha_1,\ldots,\alpha_n.\tau$, only those α_is that actually occurred inside the element type of the array tycon (or other incoercible tycons) would be treated as explicit runtime type parameters, all the rest type variables can simply be considered as the single-word black boxes.

This optimization does not affect separate compilation because all of these information can be solely deduced from the type itself. Since many polymorphic functions do not involve arrays, most runtime type manipulations can be eliminated.

Finally, the parametricity property discussed here may not hold for compilers that use tag-free garbage collections [34]. The precise relationship between the two is out of the scope of the current paper.

5 Implementation

We have implemented the flexible representation analysis technique outlined above in an experimental version of the Standard ML of New Jersey compiler (v109.25m) [3, 33, 31]. To simplify the implementation, we avoid the *vararg* problem by using partial boxing as the canonical boxed form. We used the standard technique of minimal typing derivations [33, 6] to eliminate local and unnecessary polymorphisms. We did not exploit the parametricity property (discussed in Section 4.3) in the current implementation.

Bmark	**Old** total (gc)	**New** total (gc)	**Ratio**
boyer	0.97 (0.01)	0.97 (0.00)	1.00
fft	15.8 (2.96)	3.56 (0.00)	0.23
kb-comp	3.92 (0.33)	4.10 (0.36)	1.05
lexgen	5.20 (0.11)	3.80 (0.10)	0.73
life	0.63 (0.00)	0.63 (0.00)	1.00
mbrot	4.89 (0.00)	3.53 (0.00)	0.72
mlyacc	1.87 (0.14)	1.89 (0.10)	1.01
nucleic	3.18 (0.00)	2.09 (0.00)	0.66
sieve	0.46 (0.00)	0.46 (0.00)	1.00
simple	14.6 (1.59)	10.7 (0.00)	0.73
unzipr	3.03 (0.02)	6.07 (0.17)	2.00
vliw	8.87 (0.06)	9.87 (0.06)	1.11

Figure 9: Performance Measurements (Execution Time)

All recursive and mutable tycons are treated as incoercible tycons, just like `pack` in SRC. Lists are represented using the simply boxed representations shown in Figure 2b. The polymorphic array tycon is implemented specially: if the element type is real or int32, then we use the flat array, otherwise, we use the simply boxed array.

The only operations that require runtime type analysis are the coercion primitives (i.e., `wrap` and `unwrap`), the array primitives, and several *conrep* primitives (used to determine representations for concrete datatypes, with similar spirits to Appel [1, section 4.1]).

We compare the new technique with Leroy's standard coercion-based approach. Here, **Old** is the type-based compiler described in Shao and Appel [33, 29]); **New** is the new compiler that implements the flexible representation analysis described in this paper. In Figure 9, we give the measurement results (in seconds) of running the **Old** and **New** compilers on twelve ML benchmarks on a Sun Sparc20 station with 128 Mbytes memory. For each benchmark, we measured the total execution time (including GC) and also the time spent on garbage collection.

Our measurements show that several benchmarks involving recursive and mutable types gets from 27% – 77% speedups. For example, the **nucleic** benchmark involves large lists of type `(real*real*real) list`, originally it has to be fully boxed, but with our new scheme, they use more efficient simply boxed representations; the **simple** benchmark and the **fft** benchmark get speedup for the similar reasons (by using flat arrays). Benchmarks involving heavy polymorphic code (e.g., **sieve**, **mlyacc**, **vliw**, and knuth-bendix **kb-comp**) remain almost as efficient as before. We notice that most uses of polymorphic functions are to apply them directly to monomorphic data structures or to polymorphic *incoercible* objects. The 5% to 10% slowdown on **kb-comp** and **vliw** is mostly caused by the extra runtime type passing in the **New** compiler. The **unzipr** benchmark illustrates a worst-case scenerio on using partially (or simply) boxed representations; this benchmark calls the function `unzip` (defined in Section 2.2) and `zip` upon a 10000-element list of float pairs (e.g., `p` in Figure 1). Because of the extra coercions on each element, the **New** compiler runs nearly twice slower than the **Old** compiler. In the future, we plan to use more aggressive type specializations [35] and to exploit the parametricity described in Section 4.3 to to eliminate these overheads.

It is worth pointing out that the speedup we got here is likely similar to those found in the pure-type-passing compilers (e.g., TIL [34]). The strength of our flexible framework lies, however, on the fact that we can achieve the speedup without paying heavy cost on the polymorphic code. We intend to do a detailed performance comparison between SML/NJ and TIL (on heavily polymorphic code) once the TIL compiler is made publicly available.

6 Related Work and Conclusions

The main idea of our flexible representation analysis technique is to explore the use of runtime type information to support more efficient coercions and more powerful boxing schemes. Previous coercion-based approaches [19, 26, 27, 15, 33, 36] do not take advantage of runtime type information, so they can only use full boxing as its canonical boxed form. On the other hand, previous type-passing approaches [25, 14, 24, 34] do not box polymorphic objects, so they require extensive runtime type analysis and code manipulations.

Most previous work on coercion-based approaches concentrate on how to use compile-time analysis to eliminate unnecessary coercions. Both Peyton Jones [26] and Poulsen [27] extend the type system to tag monomorphic types with a *boxity* annotation, and then statically determine when to use boxed representations. Henglein and Jorgensen [15] present a term-rewriting method that translates a program with many coercions into one that contains a "formally optimal" set of coercions. Shao and Appel [33] extended Leroy's scheme to the entire SML module language and also used *minimum typing derivations* [6] to decrease the degree of polymorphism thus eliminate coercions. All these techniques still apply to our flexible approaches because the top-level S and G transformations in our scheme are almost identical to Leroy's original ones. What changed in our scheme is the interpretation of the primitive coercions `wrap`$[\tau]$ and `unwrap`$[\tau]$.

Morrison, *et al.* [25] described an implementation of Napier that passed types at runtime to determine the behavior of polymorphic operations. The type-passing approach was later formalized using λ_i^{ML} by Harper and Morrisett [14, 13, 24]; the *intensional type analysis* framework they proposed is one of the main inspirations for the present work. Very recently, Tarditi and Morrisett, *et al.* [34, 24] have implemented the type-passing approach in their TIL compiler. Their preliminary measurements showed that using unboxed representations for recursive and mutable data structures can dramatically improves the performance of most of their (monomorphic) benchmarks. Unfortunately, since their compiler specialized all the polymorphic functions (in their benchmarks), it is still unclear how their type-passing approach would perform on heavily polymorphic and heavily functorized code. [2]

We have presented a new flexible representation analysis technique for the implementation of polymorphism and abstract data types. Unlike any previous approaches, our new scheme supports unboxed representations for recursive and mutable types, yet it only requires little runtime type analysis. Our scheme is very flexible because it allows a continuum of possibilities between the coercion-based and the type-passing approaches. By varying the amount of boxing and the type information passed at runtime, a compiler can freely explore any point in the continuum, choosing from a wide range of representation strategies based on practical concerns (e.g., to avoid the vararg problem).

[2] By *heavily polymorphic*, we mean code that involves extensive use of polymorphic functions, polymorphic data structures, and parameterized modules (functors) with type abstractions. It is unrealistic to specialize all polymorphic functions and functors in realistic applications (e.g., theorem provers, ML-kit compilers, SML/NJ Compilation Manager); many C++ templates users should have similar experience. Even Mark Jones's experiment [16] only eliminates the use of type classes but not all the polymorphic functions.

Acknowledgements

We would like to thank Andrew Appel, Dave Berry, Franklin Chen, Simon Peyton Jones, and the ICFP program committee for their comments and suggestions on an early version of this paper.

References

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

[2] A. W. Appel. A critque of Standard ML. *Journal of Functional Programming*, 3(4):391–429, October 1993.

[3] 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, August 1991. Springer-Verlag.

[4] J. Auslander, M. Philipose, C. Chambers, S. Eggers, and B. Bershad. Fast, effective dynamic compilation. In *Proc. ACM SIGPLAN '96 Conf. on Prog. Lang. Design and Implementation*, pages 149–159. ACM Press, 1996.

[5] M. W. Bailey and J. W. Davidson. A formal model of procedure calling convention. In *Twenty-second Annual ACM Symp. on Principles of Prog. Languages*, pages 298–310, New York, Jan 1995. ACM Press.

[6] N. S. Bjorner. Minimal typing derivations. In *ACM SIGPLAN Workshop on ML and its Applications*, pages 120–126, June 1994.

[7] D. Clement, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ML. In *1986 ACM Conference on Lisp and Functional Programming*, New York, June 1986. ACM Press.

[8] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Ninth Annual ACM Symp. on Principles of Prog. Languages*, pages 207–212, New York, Jan 1982. ACM Press.

[9] D. R. Engler. VCODE: A retargetable, extensible, very fast dynamic code generation system. In *Proc. ACM SIGPLAN '96 Conf. on Prog. Lang. Design and Implementation*, pages 160–170. ACM Press, 1996.

[10] J. Y. Girard. *Interpretation Fonctionnelle et Elimination des Coupures dans l'Arithmetique d'Ordre Superieur.* PhD thesis, University of Paris VII, 1972.

[11] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Twenty-first Annual ACM Symp. on Principles of Prog. Languages*, pages 123–137, New York, Jan 1994. ACM Press.

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

[13] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. Technical Report CMU-CS-94-185, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, September 1994.

[14] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-second Annual ACM Symp. on Principles of Prog. Languages*, pages 130–141, New York, Jan 1995. ACM Press.

[15] F. Henglein and J. Jorgensen. Formally optimal boxing. In *Proc. 21st Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 213–226. ACM Press, 1994.

[16] M. P. Jones. Dictionary-free overloading by partial evaluation. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 107–117. University of Melbourne TR 94/9, June 1994.

[17] G. Kane and J. Heinrich, editors. *MIPS RISC Architecture*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[18] P. Lee and M. Leone. Optimizing ML with run-time code generation. In *Proc. ACM SIGPLAN '96 Conf. on Prog. Lang. Design and Implementation*, pages 137–148. ACM Press, 1996.

[19] X. Leroy. Unboxed objects and polymorphic typing. In *Nineteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 177–188, New York, Jan 1992. ACM Press. Longer version available as INRIA Tech Report.

[20] X. Leroy. Manifest types, modules, and separate compilation. In *Twenty-first Annual ACM Symp. on Principles of Prog. Languages*, pages 109–122, New York, Jan 1994. ACM Press.

[21] D. MacQueen and M. Tofte. A semantics for higher order functors. In *The 5th European Symposium on Programming*, pages 409–423, Berlin, April 1994. Spinger-Verlag.

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

[23] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts, 1997.

[24] G. Morrisett. *Compiling with Types*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, December 1995. Tech Report CMU-CS-95-226.

[25] 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), July 1991.

[26] S. L. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *The Fifth International Conference on Functional Programming Languages and Computer Architecture*, pages 636–666, New York, August 1991. ACM Press.

[27] E. Poulsen. Representation analysis for efficient implementation of polymorphism. Master's thesis, DIKU, University of Copenhagen, 1993.

[28] J. C. Reynolds. Towards a theory of type structure. In *Proceedings, Colloque sur la Programmation, Lecture Notes in Computer Science, volume 19*, pages 408–425. Springer-Verlag, Berlin, 1974.

[29] Z. Shao. *Compiling Standard ML for Efficient Execution on Modern Machines*. PhD thesis, Princeton University, Princeton, NJ, November 1994. Tech Report CS-TR-475-94.

[30] Z. Shao. Flexible representation analysis. Technical Report YALEU/DCS/RR-1125, Dept. of Computer Science, Yale University, New Haven, CT, April 1997.

[31] Z. Shao. An overview of the FLINT/ML compiler. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation*, June 1997.

[32] Z. Shao. Typed cross-module compilation. Technical Report YALEU/DCS/RR-1126, Dept. of Computer Science, Yale University, New Haven, CT, May 1997.

[33] Z. Shao and A. W. Appel. A type-based compiler for Standard ML. In *Proc. ACM SIGPLAN '95 Conf. on Prog. Lang. Design and Implementation*, pages 116–129. ACM Press, 1995.

[34] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. ACM SIGPLAN '96 Conf. on Prog. Lang. Design and Implementation*, pages 181–192. ACM Press, 1996.

[35] D. R. Tarditi. *Design and Implementation of Code Optimizations for a Type-Directed Compiler for Standard ML*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, December 1996. Tech Report CMU-CS-97-108.

[36] P. J. Thiemann. Unboxed values and polymorphic typing revisited. In *The Seventh International Conference on Functional Programming Languages and Computer Architecture*, pages 24–35, New York, June 1995. Spinger-Verlag.

[37] A. K. Wright. Polymorphism for imperative languages without imperative types. Technical Report Tech Report TR 93-200, Dept. of Computer Science, Rice University, Houston, Texas, February 1993.