LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Compiler-directed Type Reconstruction for Polymorphic Languages

Computation Structures Group Memo 348
March 29, 1993

Shail Aditya

Alejandro Caro

To appear in Proc. Functional Programming Languages and Computer Architecture, Copenhagen, Denmark, June 9-11, 1993

The research described in this paper was funded in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-89-J-1988.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Compiler-directed Type Reconstruction for Polymorphic Languages

Shail Aditya Alejandro Caro

MIT Laboratory for Computer Science
545 Technology Square, Cambridge, MA 02139
{shail,acaro}@abp.lcs.mit.edu

Abstract

In tagless implementations of polymorphic languages, the run-time types of data objects may not be completely determined at compile-time. With ML-like static type-checking, a static type template can be produced for each polymorphic function that may be instantiated at run-time according to the types of its actual arguments. Still, as noted in [5], it may not be possible to reconstruct the types of some objects that are hidden inside a closure. This creates problems for applications like garbage collection and source debugging that need to understand the entire run-time state of the machine.

In this paper, we present a compiler-directed type reconstruction scheme for ML-like languages that reconstructs complete type information of all objects at run-time without using universal type-tags. Our scheme explicitly propagates compiler generated hints at run-time whenever there is a danger of losing the type information otherwise. We show a compilation strategy to automatically detect, generate and propagate such hints and a type reconstruction algorithm that uses them in the context of a source debugger [2] for the Id language [10]. We also show several compiler optimizations that reduce the run-time overhead of propagating these hints.

1 Introduction

Polymorphic programming languages provide the flexibility of code reuse by allowing objects with different types to share the same pattern of computation. A simple example is the polymorphic `length` function that counts the number of elements in a list of any type. But this feature creates problems for applications like garbage collection and source debugging that need to know the exact type of every object participating in a computation at run-time.

Traditionally, programming environments of dynamically-typed languages such as Lisp maintain this type information in the form of run-time tag descriptors on each object. Such implementations pay the price of universal tag management either in complex specialized hardware or in extra memory space and time for managing the tags in software [1].

Recently, several type reconstruction schemes have been proposed for statically-typed polymorphic languages like ML [9] that do not incur the run-time tag management overhead [1, 4, 5]. In these schemes, static type information is combined with clues from the dynamic state of the machine (the call stack) in order to reconstruct the exact types of run-time objects when required. Unfortunately, these schemes are not able to reconstruct complete type information for all run-time objects in the presence of higher-order functions [5]. Types of objects hidden inside the environment part of a closure are sometimes not reconstructible because the computation that produced the closure may have terminated and no run-time clue is available.

In this paper, we propose a general scheme for reconstructing the full run-time types of all objects without using universal type-tags while incurring a small run-time overhead controlled explicitly by the compiler. Our scheme can be viewed as a compiler-directed explicit tagging of objects, though the extra tag information is generated only where necessary and propagated explicitly at run-time in order to reduce overhead. We present our scheme in the context of a source debugger [2] for the Id programming language [10] which is a parallel, non-strict, polymorphic language with a Hindley/Milner type system [3, 8]. The main thrust of the paper is to show that explicit tagging needs to be done in very few cases that plug the informational holes in the previous schemes and that it can be set up by the compiler automatically with minimal run-time overhead and support.

The outline of the paper is as follows. First, in Section 2 we describe the problem of type reconstruction in more detail showing examples where complete reconstruction is not possible without some run-time book-keeping. In Section 3, we set up the reconstruction problem in a theoretical framework and characterize the minimum information that needs to be propagated at run-time to allow complete type reconstruction. Then, we present our compilation scheme for propagating this information in Section 4 and the type reconstruction algorithm used by the Id debugger in Section 5. In Section 6, we show a series of compiler optimizations and variations on our compilation scheme that may further reduce the book-keeping overhead of the current scheme. Finally, Section 7 presents conclusions and directions for future work.

2 Type Reconstruction Problem

The problem of type reconstruction for Id can be described as follows. At some point during the execution of a program, we wish to take a snapshot of the state of the machine and determine the type of every object accessible within the computation. These types can then be used to display the corresponding objects.

We assume that the program is statically typed and that the run-time environment does not keep *any* type information. In particular, Id run-time objects do not carry any type-tags. Therefore, the type reconstruction information so obtained may also be useful for garbage collection.[1]

Clearly, only polymorphic objects and functions pose some challenge; complete type information can be obtained at compile-time for monomorphic objects. Also note that the exact nature of the desired information depends on the application that uses it. For example, a source debugger may wish to inspect any particular object from the current run-time state of the machine whereas a garbage collector only needs to traverse those that are still in use. Also, most garbage collectors only need to differentiate between scalars and pointers to structures while a source debugger needs exact type information in order to display the object properly. In general, we would like to devise a flexible strategy that can be optimized according to the level of information desired.

2.1 A Preliminary Type Reconstruction Scheme

The compile-time type of an object is a good starting point for the reconstruction of its run-time type. The basic idea is to instantiate this compile-time type with additional type information based on the run-time call tree in order to obtain its full run-time type.

Appel noted in [1] that the types of the objects inside a polymorphic function depend on the types of its arguments. Further, the compile-time type-instances of the arguments of a polymorphic function are recorded statically at the call site in its caller. At run-time, the call site of a function can be determined by examining the return address information in the run-time call stack (which is the visible, suspended part of the dynamic activation tree). The run-time types of the arguments are then determined from their static type-instances inductively by following up the call chain possibly up to the root where the run-time types of the user-supplied arguments are available. At that point, all polymorphic functions in the call chain can be correctly instantiated revealing the run-time types of their internal objects. We illustrate this idea with a small example[2]:

Example 1:

$$
\begin{array}{l}
\texttt{def enlist x}_{t_0}\texttt{ = x:nil;}\\
\texttt{def map f nil = nil}\\
\texttt{~|~ map f (y:ys)}_{(list\, t_1)}\texttt{ = (f y}_{t_1}\texttt{):(map f ys);}\\
\\
\texttt{ex1 = map enlist (1:2:nil)}_{(list\, int)}\texttt{;}
\end{array}
$$

The function `enlist` has a type $\forall t_0.t_0 \rightarrow (list\ t_0)$ and `map` has a type $\forall t_1 t_2.(t_1 \rightarrow t_2) \rightarrow (list\ t_1) \rightarrow (list\ t_2)$. We also show the type instances of some internal identifiers as subscripts. The evaluation of `ex1` unfolds into a call to `map` which in turn calls `enlist`. If we wish to examine the `x` argument of `enlist` during one of these calls, then the run-time instantiation of its static type t_0 can be determined by following up the call chain within the definition of `map`. Here, t_0 can be related to the static type t_1 of the actual argument `y` at that call site. This relates to the type of the second argument ($list\ t_1$) of `map` which is found to be ($list\ int$) at its call site inside `ex1`. Then, both t_1 and t_0 can be instantiated to *int* giving the actual type of `x` as desired. Goldberg showed in [4] that this process can be conducted in one pass from the root of the activation tree (bottom of the call stack) to its leaves (top of the call stack) instantiating the types of all objects correctly in a single sweep.

2.2 Problems with Closures and Free Variables

Goldberg and Gloger noted in [5] that sometimes types of objects hidden inside a closure are impossible to reconstruct. Consider the following example:

Example 2:

$$
\begin{array}{l}
\texttt{def f2 x}_{t_0}\texttt{ y}_{t_1}\texttt{ = y;}\\
\texttt{g2 = if ... then f2 1}_{int}\texttt{ else f2 "foo"}_{string}\texttt{;}\\
\\
\texttt{ex2 = g2 2;}
\end{array}
$$

Here, `f2` has a type $\forall t_0 t_1.t_0 \rightarrow t_1 \rightarrow t_1$, and therefore `g2` gets bound to a partially applied function closure with type $\forall t_2.t_2 \rightarrow t_2$ that says nothing about the type of the data hidden inside it. In fact, this type cannot be determined at compile-time because it depends on the value of the predicate (...). Besides, during the evaluation of `ex2` the return address information on the call stack would point to the call site of `g2` inside `ex2`, which does not help in determining the contents of that closure either. Thus, we cannot reconstruct the type of the argument `x` within the activation of `f2` because the computation that created its closure is no longer available as part of the dynamic activation tree.

It may appear that this problem arises only when an argument of a function is never used within its body, but the following example adapted from [5] shows that this is not the case[3]:

Example 3:

$$
\begin{array}{l}
\texttt{def f3 x}_{(list\, t_0)}\texttt{ =}\\
\texttt{~~\{ def h3 z}_{t_1}\texttt{ = if length x}_{(list\, t_0)}\texttt{ == 1}\\
\texttt{~~~~~~~~~~~~~~~~~~then z:nil}\\
\texttt{~~~~~~~~~~~~~~~~~~else z:z:nil;}\\
\texttt{~~~in h3 \};}\\
\texttt{g3 = if ...}\\
\texttt{~~~~~then f3 (1:nil)}_{(list\, int)}\\
\texttt{~~~~~else f3 (true:nil)}_{(list\, bool)}\texttt{;}\\
\\
\texttt{ex3 = g3 2}_{int}\texttt{;}
\end{array}
$$

Here, the type of the function `f3` is $\forall t_0 t_1.(list\ t_0) \rightarrow t_1 \rightarrow (list\ t_1)$, and therefore the type of the computed closure `g3` is $\forall t_2.t_2 \rightarrow (list\ t_2)$. During the evaluation of `ex3` no information is available in the activation tree whether this closure contains a list of booleans or a list of integers. Goldberg and Gloger argue in [5] that since `h3` does not use the elements of its free variable list `x` but only its spine (to compute its length), a garbage collector can ignore these elements and copy just the spine. But this approach creates problems if these structures were shared in many places and is quite unsatisfactory for a source debugger that needs to display the full object.

Note that we do not have this problem all the time. For instance, the type of argument `z` within `h3` in the above example may be reconstructed to *int* by traversing up the

---

[1] Goldberg and Gloger [5] show that complete type information is not *strictly necessary* for successful garbage collection, but its absence greatly increases the complexity of the process.

[2] We use the Id language [10] for our examples. Briefly, functions are introduced with a *def* keyword and allow pattern-matching on their arguments. (:) is the infix *cons* operation.

[3] Let-bindings in Id are enclosed within {}. The result of such a block is the value of the expression following *in*.

call stack to its call site inside ex3. Some functions like map of Example 1 never have this problem:

Example 4:

```
g4 = (map enlist)(list t0)→(list (list t0));
ex4 = g4 (1:2:nil)(list int);
```

Even though here map is partially applied to enlist to yield a closure g4 with type $\forall t_0.(list\ t_0) \rightarrow (list\ (list\ t_0))$, we have not lost any type information. Instantiation of t_0 to *int* at the call site of g4 inside ex4 yields complete type information about all the internal identifiers of both map and enlist. The problem with Examples 2 and 3 is that sometimes the types of closures do not have any connection with the types of objects hidden inside them. In such cases, we are in danger of losing type reconstruction information because the closure creation site may no longer be available on the call stack.

Another interesting point is that polymorphic objects with universally quantified types do not pose this problem. The run-time type of such an object cannot be more specific than its compile-time definition type. For instance, in the following example the variable x within the body of f5 has the universally quantified type $\forall t_0.(list\ t_0)$.

Example 5:

```
def f5 y =
  { x = nil;
    def h5 z_t1 = if length x_(list t0) == 1
                  then z:nil
                  else z:z:nil;
   in h5 };
```

Now, there is no question about the contents of the closure formed by h5 over its free variable x. It can never contain an object whose type is more specific than $\forall t_0.(list\ t_0)$. For our purposes, this means that the compile-time type of a polymorphic object provides sufficient information for its run-time type reconstruction.

3 Type Reconstruction Framework

3.1 The Expression Language

We will describe the basic theoretical idea behind our type reconstruction scheme using the following small expression language.

$$e ::= c \mid x \mid \lambda x.e \mid e_1\ e_2 \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2$$

Here c and x are meta-variables for constants and variables respectively. We use such a small language in order to emphasize the fundamental nature of this problem. Of course, in a realistic language such as Id, we will have to model many other features such as multiple-arity functions, recursion, data-structures etc. But, the essential idea behind type reconstruction remains the same.

The standard Hindley/Milner typing rules and call-by-value dynamic semantics rules for this simple language are straightforward and we will not show them here. The reader is referred to [3, 8, 11] for a detailed description.

3.2 Run-time Model of Program Execution

A program in our expression language is simply a set of nested let-bindings where the innermost expression is the user query to be evaluated within their scope. Below, we describe an abstract model of evaluation for such a program.

A function application executes in the context of an **activation frame** which records the actual arguments bound to its formal parameters, the run-time objects bound to its free variables, and the values of all its local variables during execution. We assume that at any time during execution, all accessible objects either reside directly in activation frames or in global variables, or are heap data-structures accessible through the frames or the global variables. These objects, along with the current expression being evaluated, define the complete state of the machine that we are interested in deciphering. Note that we record every local variable of a computation in its activation frame so that it can be examined later on. As an optimization, it is also possible to omit type reconstruction of objects that are no longer in use as shown in [4].

In general, evaluation may proceed in parallel, so the run-time state of the machine at any moment is essentially a tree of active or suspended activation frames. We assume that at any time during execution, it is possible to examine and traverse the activation tree. In particular, the function corresponding to each activation frame is known and for each leaf activation frame, all its ancestors are visible. For the moment, this precludes the possibility of "tail calls" which will be discussed later.

Typically, the evaluation of a program is carried out in several phases. First the top-level bindings are type-checked and converted into object code at **compile-time**. Then, at **load-time**, these definitions are installed into the root activation frame. This process builds the global static and dynamic environments under which the user query is to be evaluated. When the query expression is supplied at **invocation-time**, first we type-check it in the global static environment. At this point, the exact types of all top-level objects in the root activation frame are known by construction. These typed top-level objects together with the typed query expression constitute the complete *initial state* of the machine. Finally, we begin the evaluation of the query expression at **run-time** within the context of the root activation frame. As the evaluation proceeds, the dynamic activation tree unfolds exposing more objects and closure applications. Starting from the initial state as described above, we can view type reconstructibility as an invariant condition to be maintained at each subsequent evaluation step:

Proposition 1 (Type Reconstruction Invariance) *Assuming that the types all objects in the root activation frame and the query expression are known initially, the exact types of all accessible objects can be reconstructed after each subsequent step of dynamic evaluation.*

We will show how to preserve this invariance in the next section.

3.3 Typed Dynamic Semantics

It is possible to satisfy the invariance proposition 1 trivially by propagating the type information at each evaluation step. Figure 1 shows a typed dynamic semantics for our expression language that maintains types as it evaluates. Here, ι and α are meta-variables for type-constructors and type-variables respectively. Notation $\sigma \succeq \tau$ means that the type τ is an instance of the type-scheme σ, and the sentence $E : TE \vdash e \Rightarrow v : \tau$ reads as "under the typed environment $E : TE$ the expression e with a type τ evaluates to a value v with that type". Note that all syntactic expressions are fully typed even though we do not show these types to avoid clutter.

Syntactic Categories		
$\tau \in$ Type	$::=$	$\iota \mid \alpha \mid \tau_1 \rightarrow \tau_2$
$\sigma \in$ Type-Scheme	$::=$	$\tau \mid \forall\alpha.\sigma$
Closure	$::=$	$[x : \tau', e : \tau, E : TE]$
$v \in$ Value	$::=$	Constant $\mid$ Closure
$E : TE \in$ Typed-Env	$::=$	$\{x_i \mapsto v_i : \sigma_i\}$

Inference Rules

(VAR)

$$\frac{(x \mapsto v : \sigma) \in E : TE \qquad \sigma \succeq \tau}{E : TE \vdash x \Rightarrow v : \tau}$$

(ABS)

$$\frac{TE + \{x \mapsto \tau'\} \vdash e_1 : \tau}{E : TE \vdash \lambda x.\, e_1 \Rightarrow [x : \tau', e_1 : \tau, E : TE] : \tau' \rightarrow \tau}$$

(APP)

$$\frac{\begin{array}{c} E : TE \vdash e_1 \Rightarrow [x_0 : \tau', e_0 : \tau, E_0 : TE_0] : \tau' \rightarrow \tau \\ E : TE \vdash e_2 \Rightarrow v_2 : \tau' \\ E_0 : TE_0 + \{x_0 \mapsto v_2 : \tau'\} \vdash e_0 \Rightarrow v : \tau \end{array}}{E : TE \vdash e_1\ e_2 \Rightarrow v : \tau}$$

(LET)

$$\frac{\begin{array}{c} E : TE \vdash e_1 \Rightarrow v_1 : \tau' \\ E : TE + \{x \mapsto v_1 : Gen(TE, \tau')\} \vdash e_2 \Rightarrow v : \tau \end{array}}{E : TE \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Rightarrow v : \tau}$$

Figure 1: Typed Dynamic Inference Rules.

Figure 1 may be viewed as the semantics of a tagged implementation that pays the price of maintaining types at each evaluation step. A type reconstruction scheme can then be modeled by carrying out just the value computation embedded in these rules and delaying the type computation. In order to preserve the invariance proposition 1 under such a scheme, we must make sure that the delayed type computation associated with each evaluation rule can be performed later during a type reconstruction phase. Then, by induction on the size of the evaluation tree at any given time, we will be able to reconstruct the types of all accessible objects, starting from the initial state of the machine. We discuss the various evaluation rules below.

The first observation is that the ABS-rule in Figure 1 is the only rule that has a premise depending only on the static type inference rules. That premise says that the body of the function $\lambda x.e$ should be typable under the type environment present at run-time. It is possible to omit this run-time typing entirely because it can be reconstructed from the most general Hindley/Milner typing of the function inferred at compile-time, given the right run-time instantiation of its static type environment. This fact follows directly from the completeness of the Hindley/Milner type inference algorithm [3].

Secondly, in both VAR-rule and LET-rule, we can reconstruct all the type information present in the premises by looking at the corresponding information in the conclusion sentence. This is because the premises involve only subexpressions of the expression present in the conclusion. Finally, we discuss the APP-rule below.

3.4 The APP-rule and Opacity of Closures

The APP-rule is special because it expands the state of the machine both in terms of new expressions to evaluate and new types that they are instantiated to. This expansion happens when the operator expression e_1 of the application $(e_1\ e_2)$ evaluates to a closure that contains a typed expression body $e_0 : \tau$ and a typed free variable environment $E_0 : TE_0$. We can reconstruct these run-time types by appropriately instantiating the static type of the function body e_0 within the reconstructed run-time type environment TE. The relevant static type information for a function is gathered in a *type-map* as follows:

Definition 1 (Type-map) *Given a function $\lambda x.e$ and the most general static typing of its body $TE_{static} + \{x \mapsto \tau_1\} \vdash e : \tau_2$, the* **type-map** *of the function records the following information:*

1. *The function type, $\tau_1 \rightarrow \tau_2$.*
2. *The type-schemes of all the free variables of the function, $TE_{static} \downarrow$FREE-VARS$(\lambda x.e)$.*
3. *The type-schemes of all the local variables within the function body e.*
4. *The type-instance of the expression e_1 at all application sites $(e_1\ e_2)$ within the function body e.*

A type-map is essentially a compile-time type-description of the activation frame of a function with additional type information about its internal call sites. The set of free type-variables occurring in items 1 and 2 above capture the essential part of the static type environment that needs to be instantiated at run-time to yield the appropriate run-time types of all objects in the type-map. We will denote this set by TYPE-VARS(*type-map*)[4]. This set automatically excludes the bound type-variables in the type-schemes of polymorphic free variables of the function as pointed out in Section 2.2.

Looking back at the APP-rule, we notice that the only available information about the closure in the conclusion sentence is its overall run-time type $\tau' \rightarrow \tau$. In particular, there is no directly available information about the run-time types of its free variables. Also, in curried applications of multiple-arity functions, the available closure type will correspond to just the remaining arguments of the function and may not be sufficient to instantiate the previously accumulated argument types present in its type-map. For example, recall that in Example 2 the run-time type of the closure `g2` within `ex2` did not provide any hint about the hidden first argument of the function `f2`.

The above discussion leads us to the important observation that closure types are sometimes *opaque* so that they are not sufficient to completely instantiate the type-map of the function body present inside them. We capture this fact in the following property of general multiple-arity functions:

Definition 2 (Type Conservation) *A function f with arity k and type-scheme $\forall\alpha_1 \ldots \alpha_n.\tau_1 \rightarrow \cdots \rightarrow \tau_k \rightarrow \tau_{k+1}$ is said to be* **type-conserving** *if,*

$$\text{TYPE-VARS}(\textit{type-map}_f) = \text{TYPE-VARS}(\tau_k \rightarrow \tau_{k+1})$$

Furthermore, the type-variables TYPE-VARS$(\tau_k \rightarrow \tau_{k+1})$ *are said to be* **conserved** *at the application site of its final argument.*

[4] In general, Type-Vars(T) denotes the set of free type-variables of T, where T may be a type, a type-scheme, or a type environment.

Informally, a type-conserving function can correctly instantiate its entire type-map with just the run-time type of its final application closure. It is easy to check that `map` and `enlist` from Example 1 are type-conserving, while `f2` from Example 2 and `f3` and `h3` from Example 3 are not, which is why we were losing type information in those cases.

Definition 2 may be used by a compiler to detect functions that are not type-conserving. The next question is what type reconstruction strategy should be devised for such functions? Our scheme is to make every closure object **self-sufficient**, which means that a closure for a non-type-conserving function is required to contain special "tag" objects in its environment that are inserted at the time of its creation and are deposited into the dynamic function activation. These tags are compiler-generated *type-hints* that are interpreted at run-time in order to correctly instantiate all the type-variables that were not conserved by the function. We will show such a compilation scheme in the next section.

Given self-sufficient closures as described above, all types in the premises of the APP-rule in Figure 1 can be correctly instantiated thereby preserving the invariance proposition 1 in all cases. This implies that complete type reconstruction is possible for every activation frame in the dynamic activation tree by instantiating its type-map using type-hints and the call site information available from its caller's type-map. The details of this reconstruction scheme appear in Section 5.

4 Compiling for Type Reconstruction

The basic insight of this paper is that the problem of information propagation from closure creation sites to their call sites for non-type-conserving functions may be formulated as an *overloading resolution* problem which is then handled using well-known techniques in the literature [6, 7, 13]. These techniques systematically translate overloading into parametric polymorphism by replacing unresolved instances of overloaded variables in a function with explicit parameters that are supplied at its call site. In our scheme, these parameters are the explicit type-hints that are used by the type reconstruction algorithm rather than the function itself.

We will not discuss the overloading resolution schemes here; the reader is referred to [6, 13] for a detailed description. Instead, we will show how to formulate our problem in terms of overloading *via* examples and describe the exact nature of the type-hints.

4.1 Detecting Violations of Type-Conservation

The first step in our compilation process is to identify the functions in the program that may require additional type-hints. First, we type-check each function and generate its type-map according to definition 1. Then, using this information we determine which type-variables, if any, in its type-map are not being conserved according to definition 2. For example, the type-map for function `h3` from Example 3 is shown below:

FUNCTION	DEFINED TYPE
`h3`	$t_1 \rightarrow (list\ t_1)$

FREE VARIABLES	DEFINED TYPE-SCHEME
`x`	$list\ t_0$
`length`	$\forall t_3.(list\ t_3) \rightarrow int$
`nil`	$\forall t_4.(list\ t_4)$

LOCAL VARIABLES	DEFINED TYPE-SCHEME
`z`	t_1

CALL SITE	FUNCTION TYPE INSTANCE
`length x`	$(list\ t_0) \rightarrow int$

Looking at the type-map above we have,

$$\begin{aligned} \text{TYPE-VARS}(\textit{type-map}) &= \{t_0, t_1\} \\ \text{TYPE-VARS}(t_1 \rightarrow (list\ t_1)) &= \{t_1\} \end{aligned}$$

Therefore, the type-variable t_0 is not being conserved in `h3`.

4.2 Reconstruction as Overloading

The next step is to determine what additional type information is required to correctly instantiate the non-conserved type-variables of a function, and how to propagate this information to the function's dynamic activation. This is accomplished by viewing these type-variables as unresolved instances of a fictitious overloaded operation. This allows the standard overloading resolution mechanism to pick up these type-variables as candidates for information propagation from external call sites where this information may be available. We show this process for the function `h3` below:

Example 6:

```
def f3(tr? t0) x =
  { def h3(tr? t0) z = if length x == 1
                       then z:nil
                       else z:z:nil;
    in h3(tr? t0) };
g3 = if ...
     then f3(tr? int) (1:nil)
     else f3(tr? bool) (true:nil);
```

Here, we have added an *overloading predicate*[5] ($tr?\ t_0$) as a subscript on the function `h3`. In general, a predicate is added for every non-conserved type-variable in the function's type-map. Subsequently, the standard overloading resolution mechanism automatically propagates this predicate to the place where `h3` is referenced and to the enclosing lexical function `f3` because it remains uninstantiated (and hence unresolved) in its body. Finally, this predicate propagates to the call sites of `f3` where it is completely instantiated according to the types of the arguments being supplied to `f3` and is considered to be resolved.

Intuitively, the propagation of a predicate asssociated with a function represents a lack of local type information which must be supplied from the call site where this predicate is instantiated. Furthermore, the type contained in the instantiated predicate reflects the types of the objects

[5] We follow the terminology of [6, 13] where the usual Hindley/Milner type of a function is extended with *predicates* to model overloaded variables. In Haskell [7] these are known as *class assertions*. The predicate name *tr?* in our scheme stands for *type-reconstructible?*.

present at that call site. This is exactly the information required for type reconstruction by the function that introduced the predicate. In the next section, we will show a simple strategy that directly encodes this type information as data objects and passes them as additional hint arguments to the function.

Note that in the propagation scheme described above, a predicate only corresponds to a non-conserved type-variable of a function or one of its lexical children; it does not correspond to the full type of any of its formal parameters or its free variables as suggested in [4]. That additional type information is already present in the function's type-map. Also, predicate instantiations involving polymorphic type-variables are always considered as resolved and are not propagated outwards in the light of the discussion in Section 2.2. For instance, g3 in the above example might have been defined as:

Example 7:

$$\begin{array}{l}\texttt{g3 = if ...}\\ \quad\quad\texttt{then f3}_{(tr?\,(list\,t))}\ \texttt{(nil:nil)}\\ \quad\quad\texttt{else f3}_{(tr?\,bool)}\ \texttt{(true:nil);}\end{array}$$

Here, $(tr?\ (list\ t))$ is an instantiation of f3's predicate according to its polymorphic argument (nil:nil). Even though this predicate has an uninstantiated type-variable t, it is not propagated any further because it is polymorphic at this point. It follows immediately that there can be no unresolved predicates at the top-level because there are no free type-variables in the top-level type environment by construction.

4.3 Program Translation and Hint Generation

The final step in our compilation process is to add extra hint parameters to the function definitions that possess unresolved predicates and generate type-hints at their call sites according to the instantiations of these predicates. It is possible to either add one hint parameter for each unresolved predicate or group the hints together in a single *hint-record* from which the individual hints may be fetched. Our current scheme adds one hint parameter per predicate in front of its regular parameters, because in our system, passing a small number of additional parameters is cheaper than allocating and fetching from heap data-structures. We record the mapping between the type-variables present in the unresolved predicates of a function and its additional hint parameters in a *hint-map* as follows:

Definition 3 (Hint-map) *Given a function with unresolved type-variables* $\alpha_1, \ldots, \alpha_n$*, its* **hint-map** *is the mapping* $\{(\alpha_1 \mapsto x_1), \ldots, (\alpha_n \mapsto x_n)\}$*, where* $x_1, \ldots, x_n$ *are its new additional hint parameters.*

For example, the hint-map of h3 in Example 6 above is generated as follows:

Type Variable	Hint Parameter
t_0	h3_hint_1

A predicate $(tr?\ \tau)$ appearing at a call site within a function is transformed into a type-hint that is passed as an explicit argument at that call site. This type-hint encodes the type τ using the following Id data-type:

```
type id_hint = none | tc string (list id_hint);
```

The disjunct none is used to encode polymorphic type-variables that do not require any hint. The disjunct tc encodes a type-constructor by its name and a list of encoded type-parameters. The free type-variables in τ are replaced by their corresponding hint parameters given by the hint-map of the function. For instance, the Example 6 above will be translated as follows:

Example 8:

```
def f3 f3_hint_1 x =
  { def h3 h3_hint_1 z = if length x == 1
                         then z:nil
                         else z:z:nil;
   in h3 f3_hint_1 };
g3 = if ...
     then f3 (tc "int" nil) (1:nil)
     else f3 (tc "bool" nil) (true:nil);
```

Notice how the hints generated within g3 propagate into h3 *via* the hint parameters of f3 and h3. The appropriate hint will now be available in a dynamic activation of h3 where it may be used along with its type-map to reconstruct the exact run-time type of x. We describe this reconstruction scheme in the next section in the context of a source debugger for Id.

5 Type Reconstruction in the Id Debugger

The problem of type reconstruction from the perspective of the Id Debugger [2] is broad and simple: reconstruct the type of every object in an activation frame. Once the type of an object is known, the debugger can invoke a graphical browser specialized for that particular type should the user wish to inspect the object.

The debugger utilizes both compile-time and run-time information for type reconstruction. The compile-time information consists of the type-map (definition 1) and the hint-map (definition 3) of a function that are stored in the symbol table entry for that function. The run-time information consists of the activation tree that is built dynamically, as the program executes, by the procedure linkage code. When the machine is halted, the debugger can inspect the activation frame of a function and extract a pointer to its caller (its parent in the activation tree) and pointers to its callees (its children in the activation tree). Access to the activation tree is crucial since it permits correct instantiation of the type-variables that are conserved in a function's type-map, according to the call site information in its caller's type-map. Finally, as described in the previous section, the additional compiler-generated type-hints for the non-conserved type-variables in a function's type-map are also available from its activation frame.

5.1 The Type Reconstruction Algorithm

Figure 2 shows the pseudo-code for the reconstruction algorithm Reconstruct-Type which is invoked by the Id Debugger to reconstruct the types of all variables in a function's activation frame. Reconstruct-Type takes the current activation as a parameter and returns a fully instantiated type-map for that activation. For ease of presentation, the algorithm makes use of several auxiliary functions which we will explain where necessary.

Reconstruct-Type is divided into several sections. We begin by extracting the name of the activation function from the current activation. The first section, lines 2–4, instantiates the type-map of the function with fresh type-variables by building a type substitution S_{copy}. This is necessary so

RECONSTRUCT-TYPE*(activation)*

1 *activation-fn* $\leftarrow$ ACTIVATION-FN*(activation)*

$\triangleright$ Copy the function's type-map.

2 *type-map* $\leftarrow$ TYPE-MAP*(activation-fn)*

3 $\{\alpha_1, \ldots, \alpha_n\} \leftarrow$ TYPE-VARS*(type-map)*

4 $S_{copy} \leftarrow \{\alpha_i \mapsto \beta_i\}$ where $\beta_1, \ldots, \beta_n$ are new.

$\triangleright$ Process the type-hints.

5 *hint-map* $\leftarrow$ HINT-MAP*(activation-fn)*

6 $S_{hint} \leftarrow$ { **forall** $(\alpha \mapsto x)$ **in** *hint-map*

7 $\quad\quad \tau \leftarrow$ INTERPRET-TYPE-HINT*(x, activation)*

8 $\quad\quad$ **collect** $(S_{copy}\alpha \mapsto \tau)$}

$\triangleright$ Return if the type-map is fully instantiated.

9 **if** TYPE-VARS*($S_{hint}S_{copy}$(type-map))* $= \phi$

10 **then** **return** $S_{hint}S_{copy}$*(type-map)*

$\triangleright$ Obtain call site information from the caller.

11 **else**

12 $\quad$ *parent-activation* $\leftarrow$

13 $\quad\quad$ PARENT-ACTIVATION*(activation)*

14 $\quad$ *parent-type-map* $\leftarrow$

15 $\quad\quad$ RECONSTRUCT-TYPE*(parent-activation)*

16 $\quad \tau_{use} \leftarrow$ USE-TYPE*(activation, parent-type-map)*

17 $\quad \tau_{def} \leftarrow$ DEF-TYPE*(activation-fn, S_{copy}(type-map))*

18 $\quad S_{def\text{-}use} \leftarrow$ UNIFY-ALIGNED*(τ_{def}, τ_{use})*

19 $\quad$ **return** $S_{def\text{-}use}S_{hint}S_{copy}$*(type-map)*

Figure 2: The Type Reconstruction Algorithm

that types from multiple activations of the same polymorphic function do not inadvertently interfere with each other.

The next section, lines 5–8, builds a type substitution S_{hint} for all the non-conserved type-variables of the function as prescribed by its hint-map. This is achieved *via* the auxiliary function INTERPRET-TYPE-HINT which extracts the encoded data-structure bound to a hint parameter from the given activation frame, and converts it into a type according to the encoding scheme shown in Section 4.3.

Following this, line 9 checks to see if all free type-variables of the type-map have been instantiated to either ground or polymorphic types. If so, the reconstruction is complete and the instantiated type-map is returned at line 10. Otherwise, lines 13–18 obtain the remaining information from the activation tree as follows.

First, the type-map of the parent of the current activation is reconstructed by calling RECONSTRUCT-TYPE recursively with the parent's activation frame. Using this type-map and the current activation, the auxiliary function USE-TYPE obtains the reconstructed type-instance of the call site responsible for invoking the current function (see item 4 of definition 1). This type-instance, τ_{use}, is then unified with the defined type of the current function, τ_{def} that is available in the current type-map. Note that for multiple-arity curried functions, τ_{def} will be the full function type involving all its curried arguments, while τ_{use} may simply correspond to the final curried application type of that function. Therefore, UNIFY-ALIGNED must "align" these types properly before unification. This unification fully instantiates all the remaining type-variables in the current type-map which is then returned at line 19.

A few observations about our reconstruction algorithm are worth pointing out. First, the entire activation frame of a function is reconstructed at once. This is possible because the types of all objects present in an activation frame share the same set of free type-variables which are precisely captured and instantiated using its type-map. This also obviates the need to traverse the activation tree several times for each variable separately. Of course, it is still possible to have several smaller type-maps for the same activation frame if the entire information is not required as suggested in [4] in the context of garbage collection.

Second, we traverse the activation tree from the current activation frame only as far up as necessary. This avoids traversing the activation tree from the root activation frame to all its leaves as suggested in [4] which would involve a lot of activation frames in our parallel system. Reconstructed type-maps can always be cached, so that no activation frame may need to be examined more than once.

5.2 An Example

Consider the problem of type reconstruction for the code in Example 3. Assume that the (...) in the definition of `g3` evaluates to `false` at run-time. Furthermore, suppose the program is halted when `h3` is invoked due to the application of `g3` during the evaluation of `ex3`. The problem is to reconstruct the types of the objects in `h3`.

Our compilation scheme described in Section 4 translates `f3` and `g3` of Example 3 into the code shown in Example 8. Here, `h3` is augmented with an extra parameter `h3_hint_1`. The purpose of this parameter is to carry type information concerning the non-conserved type-variable t_0 within `h3`'s type-map as shown in Section 4.1.

At run-time, RECONSTRUCT-TYPE performs the following steps when invoked on the activation of `h3`. First, it decodes the type-hint bound to the hint parameter `h3_hint_1` and generates a substitution $S_{hint} = \{(t_0 \mapsto \mathit{bool})\}$ using `h3`'s hint-map. Since the free type-variable t_1 in `h3`'s type-map is still uninstantiated at this point, RECONSTRUCT-TYPE recursively reconstructs the type-map of `h3`'s parent activation frame. Using this type-map, the call site type-instance of `h3` is found to be $\tau_{use} = \mathit{int} \rightarrow (\mathit{list}\ \mathit{int})$ corresponding to the application `(g3 2)` within `ex3`. Then, τ_{def} is assigned the type $t_1 \rightarrow (\mathit{list}\ t_1)$ from `h3`'s type-map. The unification of these two types yields the substitution $S_{def\text{-}use} = (t_1 \mapsto \mathit{int})$. Finally, the substitutions S_{hint} and $S_{def\text{-}use}$ are applied to `h3`'s type-map which results in a type of $(\mathit{list}\ \mathit{bool})$ for `x` and *int* for `z`.

In contrast with our scheme, the type reconstruction algorithms proposed by Appel [1] and by Goldberg and Gloger [4, 5] fail to reconstruct the type of `x` in the body of `h3`. The reason is that the only run-time information they use to reconstruct types is that contained in the run-time stack of the activation frames. When closures such as `g3` are created, the function that created the closure, `f3`, may not be present on the stack when the closure is actually *invoked.* Any clues that `f3` might have provided to the type reconstruction algorithm are therefore not accessible.

6 Current Status and Future Optimizations

The compilation scheme outlined in Section 4 and the reconstruction algorithm of Section 5 have been implemented in the Id programming environment for Monsoon [12]. It might seem that our compilation scheme incurs a lot of run-time overhead but our experience has been that realistic programs contain very few (if any) non-type-conserving functions, so the overhead of generating and propagating their type-hints is reasonably small. Besides, this scheme permits the Id debugger to reconstruct the types of all identifiers in any Id program successfully which is an extremely desirable feature.

Although our current performance is adequate, we hope to be able to improve our scheme through several compiler optimizations that are discussed below.

6.1 Rearranging the Hint Parameters

Currently, all hint parameters that need to be added to a function definition are placed in front of its regular parameters as shown in Section 4.3. This is not strictly necessary. We can place a hint parameter anywhere before (or just after) the regular parameter whose type contains the non-conserved type-variable that is encoded by the hint parameter. This observation follows from definition 2 by viewing every curried application closure of a multiple-arity function as attempting to conserve the overall type information contained in it either in the form of explicit type-hints in its dynamic environment or in its remaining type signature. It follows that only those hint parameters need to be placed up front that correspond to the non-conserved type-variables in the types of the free variables of a function.

The benefit of such rearrangement is that it may reduce the propagation overhead of type-hints by removing some extra parameters altogether using η-reduction. For example, the following alternate translation for Example 6 is also valid (compare with Example 8):

Example 9:

```
def f3 x =
  { def h3 h3_hint_1 z = if length x == 1
                         then z:nil
                         else z:z:nil;
   in h3 };
g3 = if ...
     then f3 (1:nil) (tc "int" nil)
     else f3 (true:nil) (tc "bool" nil);
```

Here, the parameter `f3_hint_1` of `f3` was pushed after its parameter `x` which made this η-reduction possible.

6.2 Arity Analysis

Definition 2 conservatively prescribes that the only type-variables that are conserved in a multiple-arity function are those present in its final application type because the function could be curried over its initial arguments. This definition can be generalized to include the types of all the arguments present at a call site, if that call site is guaranteed to be accessible through the dynamic activation tree. This is true for all first-order (or full-arity) applications of a function where all its arguments are supplied at once. In such cases, all the type-variables in the function's signature can be instantiated from its call site, though it may still require type-hints to reconstruct the types of its free variables.

In our current scheme, it is not possible to optimize away the type-hints at a first-order application site of a function because it may still need additional hint parameters in its definition due to higher-order application sites present elsewhere. This is just a consequence of our choice to provide type-hints by adding extra parameters to a function's definition. Alternatively, we can either generate a special first-order version of the function that does not carry any type-hints and use it wherever possible, or choose another mechanism for hint propagation that is transparent to the usual parameter passing conventions. Then we would be able to tailor the type-hints according to the information available at a particular call site without affecting the function's definition. We are currently investigating such schemes.

6.3 Escape Analysis

Along with first-order call site information, if the types of the free variables of a function are also known to be reconstructible *via* the currently visible activation tree, then no extra types-hints may be necessary in its definition even if it was determined to be non-type-conserving by definition 2. Escape analysis of function closures offers this information. Specifically, if analysis shows that a function closure does not escape from the lexical scope where it was defined, then the correct instantiations of its free variables would still be available from the activation of this ancestor in the activation tree. In that case, we do not need to set up extra type-hints to reconstruct these instantiations within the function's activation.

6.4 Tail Calls

Our current scheme does not deal with tail calls where the usual caller–callee relationship is violated. A tail call removes the caller's activation frame from the activation tree and connects the callee to the parent of the caller directly. In such a situation, the call site information for the callee is lost. Consider the following example:

Example 10:

```
def f10 x = 1 + length x;
def g10 n = if n == 1
            then f10 (1:2:nil)
            else f10 (true:nil);

ex9 = g10 ...;
```

Without tail calls, the type of `x` in an activation of `f10` can be determined by locating its call site within the `then` or the `else` branch of the conditional inside `g10`. But, if these applications were compiled as tail calls, then the `f10`'s activation will get directly connected to the top-level and the call site information will be lost.

It is easy to extend our scheme to deal with this situation. We simply modify definition 2 to reflect the fact that no call site information is available for `f10` and therefore explicit type-hints may be needed for all of its free type-variables. This leads to the following translation:

Example 11:

```
def f10 f10_hint_1 x = 1 + length x;
def g10 n =
  if n == 1
  then f10 (tc "int" nil) (1:2:nil)
  else f10 (tc "bool" nil) (true:nil);

ex9 = g10 ...;
```

Now, all the type information is available from within the activation of `f10`. Of course, this scheme is not optimal because it ignores the call site information even when it is available using regular calling conventions. In order to incorporate that flexibility, we need to generate site specific type-hints as described in Section 6.2.

6.5 Compiled *vs* Interpreted Schemes

Goldberg showed in [4] how to *compile* function-specific and site-specific garbage collection routines that understand the structure and the liveness properties of the local variables of a function. Our scheme, on the other hand, generates and *interprets* encoded type information in order to reconstruct the types of all local and free variables of a function. We do not take any position on what to do with these types. This strategy is adequate and desirable for a source debugger because it may wish to manipulate an object in many different ways. But, for a specific application like garbage collection, a compiled scheme may be more efficient.

It is possible to apply the principle of type conservation (definition 2) and the information propagation strategy (Section 4.2) in any specific context to allow complete analysis of run-time objects in that context. Specifically for garbage collection, we can generate specialized GC-routine(s) for a function instead of its type-map, parameterized by GC-routine parameters that correspond to the free type-variables in its type-map. Then, we can generate and propagate closures of GC-routines in the same way as type-hints which will be picked up automatically by the GC-routine(s) of a function from its activation frame. We will investigate such garbage collection schemes for Id in the future.

7 Conclusions

We have presented a general framework for complete type reconstruction of run-time objects in ML-like polymorphic languages without using universal type-tags. We have provided some insight into the lack of type information for objects hidden inside closures and have shown a compiler-directed scheme to explicitly propagate this information where necessary as extra arguments. Finally, we have shown a type reconstruction algorithm that combines compile-time and run-time type information to successfully reconstruct the types of all run-time objects.

Our scheme incurs a small overhead in generating and propagating the extra arguments which may be further reduced using the optimizations described in Section 6. We have implemented this scheme in the context of a compiler for the Id programming language [10] and its source debugger [2] for the Monsoon dataflow machine [12].

Our type reconstruction framework is based on a property of functions called *type conservation* that identifies the call sites in a program where type information may be lost unless it is propagated explicitly. The use of this property easily extends to more general language features and other system applications than those considered in this paper. Further work on showing the theoretical completeness of our framework and improving its performance is in progress.

8 Acknowledgments

The research described in this paper was funded in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-89-J-1988.

References

[1] Andrew W. Appel. Runtime Tags Aren't Necessary. *Lisp and Symbolic Computation*, 2:153–162, 1989.

[2] Alejandro Caro. A Debugger for Id. Master's thesis, Massachusetts Institute of Technology, February 1993.

[3] L. Damas and R. Milner. Principle Type Schemes for Functional Programs. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[4] Benjamin Goldberg. Tag-Free Garbage Collection for Strongly Typed Programming Languages. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, Toronto, Ontario, Canada*, pages 165–176, June 1991.

[5] Benjamin Goldberg and Michael Gloger. Polymorphic Type Reconstruction for Garbage Collection without Tags. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 53–65, 1992.

[6] Shail Aditya Gupta. An Incremental Type Inference System for the Programming Language Id. Master's thesis, MIT, Laboratory for Computer Science, September 1990. Available as Technical Report MIT/LCS/TR-488.

[7] P. Hudak and P. Wadler (editors). Report on the programming language Haskell, a non-strict purely functional language (Version 1.0). Technical Report YALEU/DCS/RR777, Yale University, Department of Computer Science, April 1990.

[8] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

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

[10] Rishiyur S. Nikhil. Id Language Reference Manual Version 90.1. Technical Report CSG Memo 284-2, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, July 15 1991.

[11] Mads Tofte. *Operational Semantics and Polymorphic Type Inference.* PhD thesis, University of Edinburgh, Department of Computer Science, 1988. Also published as ECS-LFCS-88-54.

[12] Kenneth R. Traub, Gregory M. Papadopoulos, Michael J. Beckerle, James E. Hicks, and Jonathan Young. Overview of the Monsoon Project. In *Proceedings of the 1991 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 150–155, October 1991. Also published as CSG Memo 338, MIT and Motorola Technical Report MCRC-TR 15.

[13] Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages, Austin, Texas*, pages 60–76, January 1989.