

# Compiler-directed Type Reconstruction for Polymorphic Languages (Extended Abstract)

Shail Aditya\*

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

## 1 Introduction

Polymorphic programming languages provide the flexibility of code reuse by allowing objects with different types to share the same pattern of computation. 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, dynamic objects in a polymorphic language keep additional type-tags to identify themselves. But, this scheme either requires complex hardware support or costs space and time overhead in managing the tags in software.

This paper [1] proposes a compiler-directed, explicit tag management scheme for Id, which is a polymorphic, strongly-typed language developed by the Computation Structures Group at MIT. The underlying memory model for Id is tag-less, and the explicit tag information is automatically inserted by the compiler where necessary. Thus, the overhead of tag management is considerably reduced and is directly controlled by the compiler. At the same time, complete type information for all run-time objects can be reconstructed, which was not possible in earlier schemes [2, 4].

## 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. The accessible objects are those that reside in the activation frames of functions and in the reachable heap. The reconstructed types can then be used, for example, by a source debugger to display the corresponding objects properly, or by a garbage collector to guide the traversal and marking of the reachable heap objects.

In monomorphic languages, it is sufficient to generate a detailed **type-map** for each function at compile-time that records the type of every local and heap variable accessible within its definition [2]. This type-map can then be used to correctly interpret the raw data present in the activation frames while traversing and displaying accessible objects.

Polymorphic languages pose a new challenge. In a polymorphic function, the types of the actual arguments supplied at a call-site may be more specific than those recorded in its

type-map. For complete type reconstruction, one must instantiate the types stored in the function's type-map with the types of its actual arguments. This is achieved as follows. For each function definition at compile-time, the compiler records the types of all arguments supplied at every call-site in its body within its type-map. At run-time, the type-map of every activation frame present in the activation-stack (starting from the top-of-stack frame down to the root frame) is instantiated with its actual argument types. This is accomplished by recursively matching the defined function type from each type-map with the type recorded at the corresponding call-site in its caller's type-map. The recursion bottoms out at the root frame where the actual type of the top-level call is known completely [2, 4].

Unfortunately, this strategy does not always work for polymorphic, higher-order functions that create and return lexical closures. This is because the types of free variables captured within closures are visible only at closure creation-sites which may not longer be reachable *via* the current activation-stack when those closures are applied later. This problem is illustrated by the following example<sup>1</sup>:

### Example 1:

```
def eq_len? l1 l2 = length l1 == length l2;
test_len = if pred then eq_len? (1:2:nil)(list int)
           else eq_len? ("foo":Nil)(list string);
test_len (true:nil)(list bool);
```

The polymorphic function `eq_len?` compares the length of two polymorphic lists. Its type-map is shown in the table below, where  $t_1$  and  $t_2$  are type-variables that have to be instantiated according to the types of actual arguments supplied at a call-site.

FUNCTION	DEFINED TYPE
<code>eq_len?</code>	$(list\ t_1) \rightarrow (list\ t_2) \rightarrow bool$
FREE VARIABLES	DEFINED TYPE-SCHEME
<code>length</code>	$\forall t_3.(list\ t_3) \rightarrow int$
LOCAL VARIABLES	DEFINED TYPE-SCHEME
<code>l1</code>	$(list\ t_1)$
<code>l2</code>	$(list\ t_2)$
CALL SITES	FUNCTION TYPE INSTANCE
<code>length l1</code>	$(list\ t_1) \rightarrow int$
<code>length l2</code>	$(list\ t_2) \rightarrow int$

\* Faculty Supervisor: Prof. Arvind. 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.

<sup>1</sup>The examples uses the Id language syntax. Functions are introduced with the keyword `def`; `(:)` is the infix *cons* operation; and multi-arity functions can be *curried* to produce lexical closures.

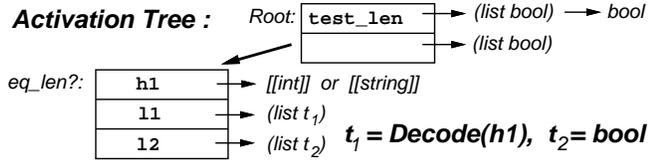


Figure 1: Run-time Type Reconstruction using Type-Hints.

The closure `test_len` captures a list object whose actual type cannot be statically determined because it is selected differently according to a dynamic predicate. When applied to another argument, the closure `test_len` expands into an activation of the function `eq_len?`. Since this call-site is reachable from the current activation-stack, the type of the last argument `l2` can be reconstructed using the strategy described above, giving the type  $(list\ bool)$  for `l2`. But, the call-site of the first argument `l1` is no longer reachable because that computation has already terminated producing the closure `test_len`. Thus, the exact call-site for `l1` cannot be determined and hence its actual type cannot be reconstructed.

### 3 Full Type Reconstruction using Explicit Type-Hints

For complete run-time type reconstruction, we have to provide some extra type information for the objects hidden inside function closures. This information is available at closure creation-sites but not at their final call-sites. Our strategy is to make closures *self-sufficient* by capturing this additional type information in an encoded form within the closures themselves at their creation-sites. These encoded **type-hints** are propagated into the dynamic function activations as extra arguments where they can be decoded to achieve complete type reconstruction. This compiler transformation is shown below for the case of Example 1:

#### Example 2:

```
def eq_len? h1 l1 l2 = length l1 == length l2;
def test_len = if pred then eq_len? [[int]] (1:2:nil)
                else eq_len? [[string]] ("foo":nil);
test_len (true:nil);
```

The compiler adds one extra type-hint parameter `h1` to the function `eq_len?` whose run-time value can be decoded to instantiate the type-variable  $t_1$  in its static type-map as shown in Figure 1. The remaining type information is determined as before using the last available call-site in the root activation frame.

### 4 Hint Generation Criteria and Optimizations

It is clear that explicit type-hints constitute an overhead that must be minimized by the compiler. Their cost has to be compared with schemes using universal type-tags on every object. The advantage of our scheme lies in the fact that explicit type-hints are necessary only for certain kinds of polymorphic, higher-order functions that are in danger of losing type information when creating closures. In most cases, a lot of information can be obtained from a function's static type-map and its final reachable call-site. In fact, for certain **type-conserving** functions, all the information

needed for full type reconstruction is available from the final call-site. Such functions are characterized below<sup>2</sup>:

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

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

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

Informally, a type-conserving function can correctly instantiate its entire type-map with just the actual types available at its final call-site and therefore does not need any extra hint parameters. All single arity closed functions satisfy this criterion trivially. Many polymorphic functions, such as `map`, `reduce`, `append`, `find`, `sort`, etc. also satisfy this criterion with appropriate ordering of arguments. Furthermore, the above criterion can be extended to include the types of all arguments present at the final call-site rather than just the final argument. For example, no type-hints are needed at a *first-order* call-site, which identifies the types of all actual arguments at the same time. With closure escape analysis, it may also be possible to determine whether the creation-site of a closure will be reachable from the activation-stack when that closure is applied later on. In such a case, no type-hints are necessary for the free variables captured in the closure either.

### 5 Current Status and Future Work

Our current scheme adds additional type-hint parameters to all functions that are not type-conserving according to Definition 1. This scheme has been implemented in the Id Source Debugger for Monsoon [3] and is able to successfully display the complete state of an Id program at any time during its execution.

As future work, we plan to implement the call-site specific optimizations as mentioned above in order to further reduce the type-hint overhead. We also plan to apply the type conservation principle and the hint propagation mechanism to automatically generate specialized garbage collection and I/O routines for Id that would understand the complete structure of run-time objects without relying upon universal type-tags.

### References

- [1] Shail Aditya and Alejandro Caro. Compiler-directed Type Reconstruction for Polymorphic Languages. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark*, June 1993. To Appear.
- [2] Andrew W. Appel. Runtime Tags Aren't Necessary. *Lisp and Symbolic Computation*, 2:153–162, 1989.
- [3] Alejandro Caro. A Debugger for Id. Master's thesis, Massachusetts Institute of Technology, February 1993.
- [4] 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.

<sup>2</sup>  $\text{Type-Vars}(T)$  denotes the set of free type-variables of  $T$ .