Polymorphic Type Reconstruction for Garbage Collection without Tags

Benjamin Goldberg
Department of Computer Science
New York University[1]

Michael Gloger
Department of Computer Science
Technical University of Darmstadt[2]

Abstract

Several papers ([Appel89],[Goldberg91]) have recently claimed that garbage collection can be performed on untagged data in the presence of ML-style type polymorphism. They rely on the ability to reconstruct the type of any reachable object during garbage collection. The bad news is that this is false – there can be reachable objects in the program whose type cannot be determined by the garbage collector. The good news is that tag-free garbage collection can be performed anyway – any object whose type cannot be determined by the collector is, in fact, garbage. Such objects can be discarded by the collector. This is the key result of this paper.

We present a type reconstruction algorithm that can determine the type of any non-garbage object. Unfortunately, the implementation of the tag-free collector for a polymorphically typed language is difficult in ways that were not described in the previous papers, and we address some implementation issues as well. However, we mainly describe how to perform type reconstruction during garbage collection and do not attempt to address practical issues of the garbage collection process.

1. Address: 251 Mercer Street, New York, NY 10012, USA. email: goldberg@cs.nyu.edu. Phone: (212) 998-3495.
2. Address: D-6100 Darmstadt, Magdalenenstr. 11c, Germany. email: gloger@pi.informatik.th-darmstadt.de.

This research has been supported, in part, by the National Science Foundation (#CCR-8909634) and DARPA (DARPA/ ONR #N00014-91-J1472).

1992 ACM LISP & F.P.-6/92/CA
© 1992 ACM 0-89791-483-X/92/0006/0053...$1.50

1. Introduction

Methods for performing garbage collection for strongly typed languages in the absence of tagged data have been around since the early days of ALGOL68. These methods have been based on the ability to associate type information with the code of each procedure, such that when the garbage collector is traversing a procedure's activation record, the type information can be extracted and used by the collector. This type information can be encoded into type descriptors that are interpreted by the garbage collector, or encoded as garbage collection routines that are called by the garbage collector. Since the type of each variable in a procedure definition is the same for all calls to the procedure and is known at compile time, the compiler can produce the encoded type information for each procedure.

Recently, there have been several papers describing tag-free garbage collection for polymorphically typed languages such as Standard ML [MLH90]. In Standard ML, the same function may be applied to arguments of different types, and thus the types of the variables in different activation records for the same function may vary. The solution to this problem was initially suggested in [Appel89], in which it was noticed that the types of the variables in the activation record of a function f may depend on the types of the variables in f's caller, in f's caller's caller, and so on. Thus, the type of the variables in an activation record can be determined by examining the type information associated with the other activation

records on the stack.

As a possible implementation of this scheme, [Goldberg91] described a method in which type information was propagated up the stack (from the oldest frame to the newest frame) in order to determine the types of each variable in each activation record during garbage collection.

The details of the methods previously proposed are described in the following section. Unfortunately, what each of these methods failed to address is that there can be objects that are reachable by the collector but whose types cannot be reconstructed. While this would seem to preclude tag-free garbage collection for polymorphic languages, we show in this paper that this is not so. We prove that tag-free garbage collection is possible for languages like Standard ML, and extend the methods described in [Appel89] and [Goldberg91] to insure safe garbage collection.

Some difficult implementation issues arise for polymorphic tag-free garbage collection that were not addressed in previous papers. We describe these problems, and suggest some solutions in section 6.

Throughout this paper, however, we mainly concentrate on type reconstruction without being concerned with the efficiency of the garbage collection process. We felt that by ignoring some of the garbage collection details, we could present a clearer explanation of the type reconstruction algorithm.

2. Related Work

The fundamental idea behind tag-free garbage collection is that compile-time type information is retained at run-time, but is not retained in the form of type tags. When the garbage collector traverses a data structure, it must be able to find the type information for that structure. How that type information is found, and how it is represented, differs in previously published algorithms.

In 1970, Branquart and Lewi [BL70] described two tag-free garbage collection algorithms for Algol68. The first, called the *interpretive method*, associates a template with each type that describes the structure of variables of that type. As the garbage collector traverses a data structure, it also traverses the corresponding template to determine the appropriate actions. The second method, called the *compiled method*, associates a garbage collection routine (which we shall refer to as a gc_routine) with each type in the user program. The gc_routine is executed when the garbage collector encounters a variable of the corresponding type. These garbage collection routines are produced by the compiler based on the types defined in the user's program.

Both methods described by Branquart and Lewi rely on a run-time table mapping stack locations representing local variables to type templates (for the interpretive method) or gc_routines (for the compiled method). This table must be updated whenever a new local variable is created. As the garbage collector traverses the stack, it looks in the table to find the type template or garbage collection routine for each variable. Because Algol68 discourages heap allocation of structures bound to local variables, the table seldom needs to be updated. Type templates or gc_routines can be associated directly with global variables, since their location is fixed during the computation.

In 1975, Diane Britton [Britton75] extended both the interpretive and compiled methods for Pascal. Instead of a table mapping locations to type templates or gc_routines, each activation record in the stack contains an extra pointer to a template or gc_routine corresponding to all the variables in the activation record.

In 1989 Peyton Jones and Salkid [PJS89] described a garbage collection scheme for their tagless graph reduction abstract machine. Because they implement a lazy language, each data structure is contained in a closure (that may represent an unevaluated value). The closure contains a pointer to a table of appropriate routines for the closure, including the garbage collection routines. It is not clear how this method would be adapted for strict languages like Standard ML.

Also in 1989, Andrew Appel [Appel89] outlined an interpretive method in which the template describing the variables in each activation record is accessed via the return pointer stored in the activation record. The beauty of this method is that no run-time overhead is incurred during normal execution. The only overhead occurs during the garbage collection process itself. Appel also suggest-

ed how tag-free garbage collection could be extended for polymorphically typed languages, and we discuss this in section 3.

In 1991, Goldberg [Goldberg91] described a compiled method that, like Appel's method, used the return pointer of an activation record to find the gc_routine to trace the variables in the activation record. The paper described how common program analyses, like live variable analysis, can optimize the garbage collection process by not copying reachable structures that will not subsequently be referenced. In [Goldberg92] an incremental version of the tag-free garbage collection algorithm is described.

Details of a tag-free garbage collection algorithm using the compiled method for a monomorphically typed language can be found in [Goldberg91]. For our purposes here, it is sufficient to describe it as follows:

- For each function f in the program, the compiler generates a gc_routine for tracing the variables in an activation record for f. Actually, since the number and types of variables in the activation record at different points of execution of the body of f may differ, several gc_routines are generated – one for each function call in f during which garbage collection could occur. Garbage collection can only occur during a function call, since the only way to allocate heap space is by calling a primitive function such as **cons** (For ease of presentation, we assume the compiler does not inline the code for **cons**. This assumption really has no effect on the algorithms described here, though.).
- In the code for f, the address of a gc_routine is associated with each call instruction (at some offset within the code segment). Thus the appropriate gc_routine for an activation record for f can be found by examining the return address of the next activation record on the stack.
- If a variable in f's activation record is bound to a closure representing some function g, the type of g will not necessarily reflect the types of the variables stored in the closure. Therefore, associated with the code for g (pointed to by the closure's code pointer) is a gc_routine for tracing the variables in the closure. This gc_routine is found at some offset from the start of g's code.
- When the garbage collector encounters an activation record for f, it simply calls the gc_routine associated with the current call instruction in f's code.

Throughout this paper, we will assume that a simple two-heap copying garbage collector is used. In this scheme, objects are preserved by the collector by copying them from the heap currently in use, labeled FROM space, to the other heap, labeled TO space. When all reachable objects have been copied to TO space, execution resumes and the labels of the heaps are switched.

3. Polymorphism and Tag-Free GC

In a polymorphically typed language, different activation records for the same function may contain objects of different types. Thus, the gc_routines for the function (which all activation records for the function share) cannot know the precise types of its variables.

Appel [Appel89] suggested a solution to this problem. The types of the arguments passed to a polymorphic function determine the types of its parameters and local variables. Thus, if the garbage collector cannot determine the type of a variable in a polymorphic function's activation record, then the calling procedure (found by the return address and dynamic link) is examined to determine the type of the arguments. If the calling procedure is itself polymorphic, then its caller may have to be examined, and so on. The traversing of the stack continues until the precise type of each variable in the current activation record can be determined. Since the types of the variables in the outermost function in the program (corresponding to the bottom activation record on the stack) are known, each traversal of the stack will terminate successfully.

Since Appel's solution may require many traversals of the stack, [Goldberg91] described a method requiring a single traversal of the stack. The garbage collector starts at the bottom of the stack by calling the gc_routine of the outermost function. As each successive activation record is encountered, its gc_routine is passed encoded type information, in the form of templates or gc_rou-

tines, from the previous (i.e. caller's) activation record's gc_routine. This encoded type information describes the types of the arguments that were passed in the call that created the activation record during execution. Each gc_routine reconstructs the types of the local variables from the encoded type information that was passed to it. It then passes encoded type information to the next activation record's gc_routine.

3.1. Closures - A Problem?

When a closure is created, either through a partial application of a curried function or via the evaluation of a lambda abstraction, a gc_routine must be associated with the closure's code pointer such that the gc_routine can determine how the elements (i.e. captured free variables) of the closure can be traced. In a monomorphic language, this is easily done, since the type of each variable in the closure is known at compile time.

In a polymorphic language, however, this becomes more difficult. Different closures that are created by different partial applications of the same function or by multiple evaluations of the same lambda abstraction will all contain the same code pointer. Thus, they will have the same gc_routine associated with them. But, the types of the variables that are captured may differ from closure to closure. This is because the variables in the curried function or lambda abstraction can have different types in different instantiations of the function or lambda abstraction.

The solution to this problem for an activation record representing a function call was to parameterize the function's gc_routine with encoded type information corresponding to the types of the arguments in that particular call to the function. However, this cannot be done for closures, because the types of the arguments to the curried function, or the types of the free variables captured in a lambda abstraction, can no longer be determined.

Consider the following program fragment:

```
let
   fun f x =
      fn z => if length x = 1
              then z+1
              else z-1
   val g = if ... then f [1,2,3]
                  else f [[1,2],[3,4]]
in
   g (hd [1])
end
```

where the type of `f` is α *list*$\rightarrow$ *int*$\rightarrow$*int* and the type of `g` is *int*$\rightarrow$*int*. Suppose that garbage collection occurs during the construction of the list `[1]` in the result expression. The garbage collector would need to copy the closure representing `g` into TO space.

The problem lies in reconstructing the type of the variable `x` captured in the closure. In `f`, the type of `x` is α *list* and its actual type in the closure does not appear in the type of `g`. The type of `x` in the closure depends on the type of the argument to `f`. Unfortunately, the activation record representing the call to the `f` that created the closure for `g` no longer exists, and there is no way, in general, to determine which application of `f` created the closure. Thus, the garbage collector will not be able to pass the encoded type information corresponding to the types of `f`'s arguments to the gc_routine for `g`.

On closer inspection of the program, we appear to get lucky. The gc_routine associated with the code for `g` only knows that `x` is a list and knows nothing about the type of `x`'s elements. But, notice that the values of the elements of the list are never needed! Only the length of `x` is required, and finding the length of a list does not require examining its elements.

Thus, when the garbage collector encounters the closure for `g`, it only needs to preserve the spine of `x` (i.e. just the top-levels cons cells) in the closure. The elements can be discarded. In a copying collector, an object is discarded simply by not copying it into TO space. No type information about the object is required to *not* copy it.

In this case, the garbage collector was unable to reconstruct completely the type of a variable captured in a closure, but luckily did not have to. The obvious next question is "What happens if we aren't so lucky, and

there are variables whose types cannot be reconstructed by the garbage collector but whose values must be preserved?". As it turns out, this case cannot occur. **Any object whose type cannot be reconstructed by the garbage collector is garbage!** In section 5, we prove this assertion.

Before we continue, we need to remark that our assertion is only true in the absence of the polymorphic equality operation (and the whole notion of equality types) provided in Standard ML. We rely on the fact that any operator that needs to examine the values of its operands induces a concrete (i.e. monomorphic) type on the operands. We discuss polymorphic equality (which is an example of a dynamically overloaded operator) in section 8, along with overloading supported by Haskell's type classes, and describe how tag-free garbage collection can still be supported. Thus, until section 8, assume that ML has only monomorphic equality operators.

It is worth noting that tail recursion, like closures, can lead to variables in activation records whose types cannot be reconstructed. The following program illustrates this point.

```
let
   fun f x y = length (x :: y)
   fun g n =
     if (n = 3)
        then f 5 [4,3,2]
        else f [1,2][[3,4],[5,6]]
in g x
end
```

In this case, if `g`'s call to `f` is implemented as a tail call, then `g`'s activation record will be overwritten with `f`'s activation record, and `g`'s gc_routine will not be accessible in order to indicate what types were passed to `f`. Thus, if garbage collection occurs during the execution of the `::` (cons) in the body of `f`, no information about the types of `x` and `y` can be propagated to `f`'s gc_routine. `f`'s gc_routine knows only that `y` is a list of some type. However, like the previous examples, the value of `x` and the values of the elements of `y` are not needed and do not need to be copied into TO space.

For brevity, we will only discuss the loss of type information due to closures and will assume that the compiler does not generate tail calls. However, the techniques described here are easily extended to handle a tail-recursive implementation, and our claim that objects whose types cannot be reconstructed are garbage still holds.

4. Type Reconstruction during GC

This paper concentrates on how the garbage collector reconstructs the types, if possible, of reachable structures. We do not consider in any detail what the garbage collector does with a structure once its type has been reconstructed, nor with the many implementation details that the implementor of a garbage collector needs be concerned with.

4.1. A Simple Example

Consider the following program fragment:

```
let
  fun f g x = fn y => if g x then y
                              else 0
  val c = if ... then
            f (fn x => hd x = 1) [1,2]
          else
            f (fn x => x) true
in
  c (hd [1])
end
```

where the type of `f` is $(\alpha \rightarrow bool) \rightarrow \alpha \rightarrow int \rightarrow int$ and the type of `c` is $int \rightarrow int$. Note that α, the type of the variable `x` in `f`, appears in argument position in the type of `f`'s first formal parameter, `g`. If garbage collection occurs during the construction of `[1]` in the result expression, the type of the variable `x` stored in the closure representing `c` must be reconstructed. The solution in this case (which we later generalize) is as follows:

- The gc_routine associated with `c` is the gc_routine, which we will call `f_gc`, that is associated with any closure that results from the application of `f` to two arguments. The task of `f_gc` is to reconstruct the actual type of `x`, corresponding to the type variable α in the static type of `f`.
- `f_gc` knows that the static type of `g` in the body of `f` is $(\alpha \rightarrow bool)$, where the type of `x` is also α, and that this $(\alpha \rightarrow bool)$ must unify with the type of the function bound to `g`. If this unification binds α to a mono-

morphic type, then the actual type of `x` has been determined.

- The type of the function bound to `g` is determined using the gc_routine accessible from `g`'s code pointer. In the above example, `g` is either `(fn x => hd x = 1)` or the identity function `(fn x => x)`. Each of these functions will have a different gc_routine associated with their closures. In order to unify ($\alpha \rightarrow$ *bool*) with the type of the function bound to `g`, `g`'s gc_routine is passed ($\alpha \rightarrow$ *bool*) as a parameter.
- If `g` is `(fn x => hd x = 1)`, `g`'s gc_routine unifies its static type, namely *int list* $\rightarrow$ *bool* with the type passed to the gc_routine, namely ($\alpha \rightarrow$ *bool*). In this case, the type variable α would be instantiated to *int list*. Thus the type of the variable `x` in `f` is determined to be *int list*.
- In the case where `g` is the identity function, the gc_routine for `g` unifies $\gamma \rightarrow \gamma$, the type of the identity function, with ($\alpha \rightarrow$ *bool*), the argument to the gc_routine. In this case, α is bound to *bool* and thus the type of `x` in `f` is determined to be *bool*.

In the following section, we present the algorithm for type reconstruction at garbage collection time. The above example illustrated a very simple type reconstruction, involving very little use of unification. A subsequent example shows that type reconstruction can be substantially more difficult.

4.2. The Type Reconstruction Algorithm

The real work of type reconstruction is done at garbage collection time by calls to a function called `rcst` (for `reconstruct`). The calls to `rcst` are made by the gc_routines associated with activation records and closures.

`rcst` is passed a unifier θ, mapping type variables to their current reconstructions, a list `u` consisting of type expressions that need to be unified, a list `tl` (for `type list`) consisting of variable-type pairs describing the types of variables reachable from the current activation record, and a list `cl` of closures whose gc_routines must still be called. `rcst` is defined as follows:

```
function rcst(θ,u,tl,cl) =
begin
   θ := unify(θ,u);
   (tl,cl) := subst(θ,tl,cl);
   while (cl != []) do
      (c,tc) := pop(cl);
      (θ,tl,cl) := c.gc(tc,θ,tl,cl)
   end do;
   return(θ,tl);
end;
```

where `c.gc` is the gc_routine associated with a closure `c`.

The function `subst(θ,tl,cl)` applies the unifier θ to the lists `tl` and `cl`, giving more precise type information about variables and closures. In addition, if there is any component of a variable in `tl` that had type α such that θ maps α to a function type, then place all such components are placed in `cl`, along with their types. For example, if there was a variable L in `tl` whose type was α list and θ mapped α to $\tau_1 \rightarrow \tau_2$ for some types τ_1 and τ_2, then for each element e of L, $(e, \tau_1 \rightarrow \tau_2)$ would be placed in `cl`.

We now describe how the gc_routines are generated for tracing an activation record representing the call to a function (rather than the closure representing a function). As we mentioned previously, a different gc_routine will be associated with each call instruction in the function that could lead to garbage collection.

For a function f defined as

$$\texttt{f}(\texttt{x}_1, \ldots, \texttt{x}_n) = \texttt{body}$$

of type $\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau$, the garbage collection routines generated for `f` by the compiler are parameterized by the types $\texttt{t}_1 \ldots \texttt{t}_n$ of the arguments to `f` and the result type `t`. The code generated for each gc_routine does the following:

```
procedure f_gci(t1, ... ,tn,t) =
begin
   tl := [(v1,tv1),..,(vm,tvm)];
      /* each vj is a variable in the activation record
         and tvj is its static type */
   θ := {};
   u := [tl=τ1,..,tn=τn,t=τ];
      /* τ1 ... τn are the static types of f's
```

```
        parameters, with unique variable names
        created to avoid name conflict */
    cl := [(c1,tc1), ..., (cp,tcp)];
        /* each ci is an object in f that is statically
           recognizable as a closure and whose type is
           inferred to be tci */
    (θ,tl) := rcst(θ,u,tl,cl);
    trace_ar(current_ar, tl);
        /* copy into TO space all the reachable
           structures from the activation record, using
           tl to determine their types */
    next_ar.gc(θ(ta1),...,θ(tam),θ(tr));
        /* call the gc_routine for the next activation
           record. The actual types of the arguments
           passed to the callee, and its result, are
           reconstructed by applying θ to their
           static types ta1,...,tam, and tr, resp. */
end;
```

When `trace_ar` encounters a structure whose type in `tl` is a type variable α, it does not copy that structure (see the discussion in section 6).

Now we need to describe what the gc_routine for a closure `c` looks like. The gc_routine parameters are:

- `tc` – the type of `c` to the extent that it has been reconstructed by the garbage collector
- θ – mapping type variables to type expressions
- `tl` - a list of variable-type pairs associating variables with their reconstructed types.
- `cl` - a list of closure-type pairs representing those closures whose gc_routines have yet to be called.

Here is what the compiler would generate for a gc_routine for a function `c` represented by a closure:

```
function c_gc(tc,θ,tl,cl) =
begin
    u := [(tc,t)];
        /* t is the static type of c */
    tl :=[(v1,tv1),...,(vm,tvm)] ^ tl;
        /* append (^) to tl the list of variables v1...vm
           captured in c, associated with their static
           types tv1...tvm */
    cl := [(c1,tc1)...(cj,tcj)]^ cl
        /* append to cl a list of closures c1 ... cj
           captured in c along with their static types
           tc1 ... tcj */
    (θ,tl) := rcst(θ,u,tl,cl);
    return(θ,tl)
end
```

When garbage collection occurs, all that is necessary is to call the gc_routine of the bottommost activation record. This activation record corresponds to the program itself, and since it generally takes no parameters, the gc_routine is not parameterized as well.

In order to traverse the stack from the bottom activation to the top, an initial traversal of the stack might be necessary to reverse the dynamic chain. However, if the gc_routine for each function knows precisely the size of its activation record and the offset to the next one, then this initial traversal might be unnecessary.

4.3. A More Complex Example

Consider the following program:

```
fun h1 x = fn (y::l) => y::x::l

fun h2 z =
   let fun g (y :: l) = if z y then l
                        else y :: g l
   in g
   end

fun h(fl,l) =
   fn (i,j) =>
      length (nth(fl,i) (nth(fl,j) l))

val G = h([h1 [true,false], h2 hd],
          [true,false,true])
```

Where `nth(l,i)` selects the i^{th} element from a list `l`. The types are as follows:

`h1`: $\alpha \rightarrow \alpha$ list $\rightarrow \alpha$ list

`h2`: $(\beta \rightarrow$ bool$) \rightarrow \beta$ list $\rightarrow \beta$ list

`h`: $(\gamma$ list $\rightarrow \gamma$ list) list * γ list $\rightarrow$ int * int $\rightarrow$ int

and the type of `G` is (int*int) $\rightarrow$ int. Notice that the type of the first argument to `h` is (bool list list $\rightarrow$ bool list list) list. For ease of presentation, we have chosen unique names for each type variable.

If garbage collection occurs after the creation of `G` then the garbage collector had better be able to determine

that `f1`, captured in the closure for `G`, is a list of functions and be able to reconstruct the types of the objects captured in the closures representing those functions.

The garbage collection routines are defined as follows, based on the material discussed in the previous section. `h1_gc` and `h2_gc` are the gc_routines for the closures formed by applying `h1` and `h2` to one argument, respectively. `hd_gc` is the gc_routine for the closure representing the function `hd` (i.e. `car`). `h_gc` is the gc_routine for the closure created by applying `h` to two arguments (that is, to the arguments for `f1` and `l`), as is done in the definition of `G`.

```
function h1_gc(t_h1,θ,tl,cl) =
begin
   u := [(t_h1, α list → α list)];
   tl := [(x,α)] ^ tl;
   (θ,tl) := rcst(θ,u,tl,cl);
   return(θ,tl);
end;

function h2_gc(t_h2,θ,tl,cl) =
begin
   u := [(t_h2,β list → β list)]
   tl := [(z,β→bool)] ^ tl
   cl := [(z,β→bool)] ^ cl
   (θ,tl) := rcst(θ,u,tl,cl)
   return(θ,tl)
end;

function h_gc(t_h,θ,tl,cl) =
begin
   u := [(t_h, (int*int) → int)
   tl :=[(f1,(γlist →γlist)list),
         (l,γlist)] ^ tl;
   for c in f1 do
      cl := [(c,γlist →γlist)] ^ cl;
       /* for each closure c in the list f1, put c,
          along with its static type, onto cl */
   (θ,tl) := rcst(θ,u,tl,cl);
   return(θ,tl);
end

function hd_gc(t_hd,θ,tl,cl) =
   u := [(t_hd, δ list → δ)]
   (θ,tl,cl) := rcst(θ,u,tl,cl)
   return(θ,tl)
```

Suppose garbage collection occurred after the creation of G. Since G is a variable of type (int * int) → int in the bottommost activation record (representing the main program), the gc_routine for the main program simply calls `G`'s gc_routine, namely `h_gc`, as follows:

`h_gc((int*int)→ int,{},[],[])` is called:

- `u` = [] (no use unifying two identical monomorphic type expressions)
- `tl = [(f1, (γlist →γlist) list), (l,γlist)]`
- `cl = [(h1_gc, γlist →γlist), (h2_gc, γlist →γlist)]`
- `rcst({},u,tl,cl)` is called.

`rcst ({},[(f1, (γlist → γ list) list), (l,γlist)],[(h1_gc,γlist→γ list), (h2_gc, γlist → γ list)])`
does the following:

- `(tl,cl) = subst({},tl,cl)`, so
  - `tl = [(f1,(γ list → γlist) list), (l,γ list)]`
  - `cl = [(h1, (γlist → γlist), (h2, (γlist → γ list)]`
- `h1_gc(tc,{},tl,cl)` is called.

`h1_gc(γlist → γlist, {}, tl, [(h2_gc, (γlist →γlist)])`
does the following:

- `u = [(γlist →γlist, αlist →αlist)]`
- `tl = [(x,α),(f1,(γlist→γlist)list), (l,γlist)],`
- `cl = [(h2_gc, (γlist → γlist)]`
- `rcst({},u,tl,cl)` is called

`rcst` unifies γ with α, so

- `tl = [(x,γ),(l,γ list), (f1,(γ list →γ list)list)]`
  and
  θ = { α = γ }.
- `(c,tc) = (h2_gc,(γlist→γlist)list)`
- `cl = []`

- `h2_gc` is called. Notice that even though `h1_gc` has finished, the type of `x` in `h1` is still unknown.

`h2_gc(γ list → γ list, {α=γ}, tl, [])` does the following:

- `u = [(γ list→γ list, β list →β list)]`
- `tl = [ (z,β→bool), (x,α), (l,γ list), (f1, (γ list → γ list) list)]`
- `cl = [(z,β→bool)]`
- `rcst(q,u,tl,cl)` is called.

`rcst` unifies β with γ, so

- `tl = [(z,γ→bool), (x,γ), (l,γ list) (f1, (γ list → γ list) list)]`
- `cl = [(z,γ→bool)]]` and θ = { α = γ, β = γ }.
- `(c,tcl) = (z,γ→bool)` and `cl = []`
- `z.gc`, which is really `hd.gc`, is called.

`hd_gc( γ→bool,{α=γ,β=γ}, [(z,γ→bool), (x,γ), (f1, (γ list→ γ list) list), (l,γ list)], [])` does the following:

- `u = [(γ→bool, δ list →δ)]`
- `rcst(q,u,tl,cl)` is called, unifying δ with bool and γ with bool list. θ becomes { δ = bool, γ = bool list, α = bool list, β = bool list}.
- θ is applied to `tl`, giving:

```
[(z,bool list→bool), (x,bool
list),
(f1, (bool list list→bool list list)
     list),
(l,bool list list)]
```

Since there are no more closures in `cl`, the unifier θ and the list `tl` of the types of the variables are returned all the way back to the gc_routine for the activation record. Now that all the types are known, the collector can copy all the reachable structures into TO space.

One of the issues we have ignored is when the closures are copied into TO space. During the reconstruction process, most of each closure can be copied. It is just the variables captured in each closure whose types have not yet been reconstructed that cannot be copied yet. In fact, in order for the reconstruction not to repeat calls to the same closure's gc_routine if the closure is shared, the closure should be copied when it is first encountered.

5. Correctness Proofs

Lemma 1 *In Standard ML (minus polymorphic equality), no primitive operator p whose behavior depends upon the value of an operand (and therefore must access the value of that operand) can be polymorphic with respect to that operand. That is, in the type signature of the operator, the type term corresponding to the required operand cannot be a simple type variable. Likewise, if p requires the value of some component of its operand, then the type of that component in the type signature of p cannot be a simple type variable.*

Proof This can be shown by a simple enumeration of the primitive operators in Standard ML (minus the polymorphic equality operator). Intuitively, this simply says that in order for the value of an object to be accessed, its type must be known. ❑

Lemma 2 *If o is a variable with a static type σ in the body of a function f such that there is an expression p(o) in f, where p is a primitive operator whose parameter type is τ, then σ must be an instance of τ.*

Proof In the ML type system, the type of an argument to a function or operator must be an instance of the parameter type of the function or operator. Thus, σ must be an instance of τ.

Lemma 3 *For any function f, the static type of each function g referenced in the body of f can be determined at type reconstruction time.*

Proof If *g* is a known (i.e let-bound) function, then the type of each instance of *g* in *f* has been determined by the static type inferencer. Otherwise, *g* is an unknown (i.e. lambda-bound) function represented by a closure and its static type is known by the gc_routine accessible from the closure's code pointer.

Theorem 1 *Suppose there is an object o in the activation record for a function f, such that the static type of o*

is σ and the value of o will subsequently be passed to a primitive operator p whose corresponding argument type is τ. Suppose also that the type of the call to f has been reconstructed. The type reconstruction method described here will reconstruct the actual type of o to a new type that is an instance of τ.

Proof There are three ways in which the value of o could be passed to p:

case 1. p(exp) appears in f, such that the value of exp could be the value of o. In this case, *exp* must have type σ (guaranteed by ML's strong typing) and the static type system would have insured that σ is statically an instance of τ. No type reconstruction need occur in this case.

case 2. The value of o is returned by f to its caller and the value is subsequently passed to p. Thus, σ appears in the static result type of f, but an instance of τ must appear in result component of the reconstructed type of the call to f. The type reconstruction method unifies the type of the call to f with the static type of f. Therefore σ would be unified with an instance of τ. In this way, the type of o is determined to be an instance of τ.

case 3. The value of o is passed to a function f_1 which passes the value to f_2 and so on until the value of is passed to a function f_n in which p is applied to the value. The static type of the parameter of f_n (corresponding to the value of o) is an instance of τ. Each of $f_1 \ldots f_n$ can be either a known (i.e. let-bound) function or a unknown (lambda-bound) function. If all of $f_1 \ldots f_n$ are known functions, then the static type inferencer would have induced a type on o that is an instance of τ, and no type reconstruction is necessary.

If $f_{i_1}, f_{i_2}, \ldots, f_{i_j}$ are lambda-bound functions in the sequence of calls, for $1 \le i_1 \le i_2 \ldots \le i_j \le n$, then the static argument types (corresponding to where o is passed) of the calls to $f_{i_1}, f_{i_2}, \ldots, f_{i_j}$ must be precisely σ, since they cannot be polymorphic. Since the scope of σ is only the body of f, each f_{i_k} must be a lambda-bound function in f or a lambda-bound function inside some other f_{i_m}. f_{i_1} must be a lambda-bound function in f.

In addition, the type of the parameter of f_{i_j} must be an instance of τ (since each call in the sequence of calls to known functions after f_{i_j} is statically inferred to have an argument type that is an instance of τ). This means that the closure for each f_{i_k} will be traversed during the type reconstruction of f's activation record, such that the input types of all these lambda-bound functions are unified with each other and σ. Since the input type of f_{i_j} is an instance of τ, σ will be unified with that type and the type of o will be reconstructed as an instance of τ.

Theorem 2 *Suppose there is an object o whose static type is σ captured in a closure representing a function g which is reachable from the activation record for a function f, such that the value of o will subsequently be passed to a primitive operator p whose corresponding parameter type is τ. Suppose also that the type of the call to f has been reconstructed. The type reconstruction method described here will reconstruct the actual type of o to a new type σ' that is an instance of τ.*

Proof The proof is essentially identical to the proof of Theorem 1. There are three cases in which the value of an object stored in a closure could be accessed by a primitive operator, and these are same three cases as when the object is stored in an activation record.

Theorem 3 *The type reconstruction method described here will, during garbage collection, reconstruct the types of all reachable, non-garbage, objects.*

Proof All that remains to be shown is that the actual type of the function call that created each activation record can be reconstructed. If so, then Theorem 1 and Theorem 2 show that the type of each reachable object in each activation record, such that the object's value will be passed to a primitive operator (i.e. it is non-garbage), will be reconstructed.

This is easily proved by induction on the distance d of the activation record from the bottom of the stack.

- d=0: The activation record at distance 0 from the bottom of the stack is the procedure representing the program. It is called with no parameters and has a known return type (void, or possibly some value).
- Assume that for all activation records at a distance k or less from the bottom of the stack, the type of every reachable non-garbage object in the activation record can be reconstructed. This means that the types of the parameters and result type of the function call corresponding to the activation record at distance k+1 can be reconstructed.

- By Theorem 1 and Theorem 2, since the types of the parameters and the result type of the call to the function has been reconstructed, the types of non-garbage objects reachable from the activation record can be reconstructed.

6. Implementation Issues

Theorem 3 indicates that if the type of an object cannot be reconstructed during garbage collection, then it can be discarded. For a copying garbage collector, this simply means that any such object need not be copied into TO space. There are some difficult implementation issues that arise, however, when such an object is not copied, especially when that object is an element of some larger aggregate structure.

Consider the following program:

```
let
   fun f x =
      fn z => if length x = 1
              then z+1
              else z-1
   val L = [[1,2],[3,4]]
   val g = if ... then f [1,2,3]
                  else f L
in
   g (hd (map hd L))
end
```

This is the same example as we saw earlier, except that now the value of the variable `x`, which is a list whose element types cannot be reconstructed, is the shared list `L` whose elements are needed in another part of the program (in this case, in `(hd (map hd L))`).

A problem arises if garbage collection occurs during the execution of `map`. When the gc_routine for `g` is executed, it only copies the spine of `L` into TO space. When the gc_routine for `map` is subsequently executed, it sees that the first cell in `L` is in TO space and would normally assume that the entire list has already been copied.

The most straightforward, and costly, solution to this problem is to remove the assumption that if a cons cell has already been copied to TO space, then the garbage collector need not trace the car and cdr of the cell. Not only is this terribly inefficient, but may lead to infinite garbage collections on circular structures.

Note however, that if the gc_routine for `map` had copied the entire list `L` into TO space before the gc_routine for `g` attempted to copy its spine, then `g`'s gc_routine would see that the first cell of the list had been copied into TO space and would correctly assume that the whole spine had been. Thus, a second solution to the above problem would be to make sure that if a list (or another aggregate structure) is shared, then the gc_routine that copies the most of the list would do so first.

One way of doing this would be to defer the *partial* copying of any structure until it is certain that the structure will not be fully copied during the current garbage collection. This can be done by keeping a defer-list at garbage collection time of the structures that need to be partially copied and inserting the addresses of the structures onto the list, along with a type descriptor. This is a solution similar to that used for *weak pointers* in many LISP and Smalltalk systems.

After the garbage collector has finished traversing the stack, it copies the structures on the defer-list, if necessary. Notice that the garbage collector might attempt to place a structure on the defer-list more than once. Even worse, the garbage collector might attempt to partially copy a structure several times in several different ways. For example, in the program

```
let
  fun  f [] n = n
    |  f ((x,_) :: l) n = x + f l (n+1)
  fun  g [] n = n
    |  g ((_,y) :: l) n =
          if y then n else g l (n+1)
in
  (fn L => [f L, g L])
       [(1,false),(2,true),(3, false)]
end
```

If garbage collection occurs after the application of `f` and `g` to `L` then the type of the list `[(1,false), (2,true), (3,false)]` is reconstructed as (int * α) list in the closure for `f` and as (β * bool) list in the closure for `g`. Thus, a pointer to the list will be placed on the defer-list two different times with two different types.

If a pointer to the list appears with two different types on the defer-list then the problem of recognizing when the list has been copied is compounded. The solution is

to place each structure on the defer-list at most once, and to unify the types of the different reconstructions of the structure. In the above example, the list would first be placed on the defer-list with the type (int * α) list. Then, when the collector tries to place the same list on the defer-list with type (β * bool) list, the two types would be unified to (int * bool) list and the list would only appear once on the defer-list. The defer-list should probably be implemented as a hash table in order to recognize when the same structure is encountered twice with two different partial types.

But, there is even a more serious difficulty. Suppose a list S is shared between two structures A and B that are to be partially copied. When A is partially copied, it might copy the spine of S. However, B might require more of S to be copied. When B sees that the first cons cell of S has been copied, it still can't make any assumption that S has been copied sufficiently. The problem is that the types of the two structures A and B put on the defer-list will not be unified in order to determine the maximum copying of S required. At this point, we do not have a reasonable solution, although one possibility is to leave type information in the evacuated cells so that the garbage collector can later determine how much of a structure was actually copied.

7. Sharing Analysis to Optimize Polymorphic Tag-Free GC

Many papers (e.g. [Deutsch90], [ISY88], [JLM89], [Park91]) have been published recently describing algorithms for detecting sharing of aggregate structures in a program. We do not describe a new method for detecting sharing, but show how sharing detection could be used to improve tag-free garbage collection.

A difficulty arises, as described in the previous section, when an aggregate structure that is shared is only partially copied into TO space. Thus, to avoid this problem, the structure is placed on a defer-list, (or some other way of ordering the copying of structures must be used).

Sharing analysis can solve this problem, because it can often determine when an aggregate structure is not shared. When an unshared aggregate structure is encountered such that the structure needs only to be partially copied, then the structure can be copied as soon as it encountered by the garbage collector.

8. Operator Overloading and Type Classes

In languages like Haskell, operators which reference the value of their operands can be dynamically overloaded (much like the = operator is for equality types in Standard ML). This is accomplished in the type system by defining type classes, which are sets of types over which certain operators and functions are defined. For any two types in a class, the definition of the same operator may be different.

The theorems proved in section 5 rely on the fact that an object's value can't be referenced within a function without inducing a specific (ground) type on the object. Thus, our garbage collection method would not be correct in the presence of dynamic overloading. However, this is easily fixed.

In order to implement type classes, the code for a particular operation is determined at run-time. In order to accomplish this each function, whose parameter type t can be any member of a certain type class, contains an implicit extra argument - a dictionary containing the code for the operations supported by the class. When an operator is applied to elements of t, the code for that operator is found in the dictionary.

In languages like Haskell, the encoding for a given type can simply be associated with the method dictionary for the type. The garbage collector, as it traverses the stack, can examine the method dictionaries stored in each activation record to determine how the variables of each type should be traced.

A similar method can be used to implement the polymorphic equivalence operator in Standard ML. Each function expecting parameters with equality types could be passed the appropriate equality operator. Each equality operator could have type information associated with it for the garbage collector to use.

It is worth noting that a method dictionary can be considered a kind of tag that is associated with values during execution. In that sense, our garbage collection algorithm is no longer "tag-free", since it relies on the

method dictionaries. If a language's type system requires tagged data, there is certainly nothing a garbage collector can do to remove that requirement!

9. A Final Word

The reader of this paper is probably left with a feeling that type reconstruction by the garbage collector is probably so expensive, with its liberal use of unification, that any benefit of omitting tagged data is outweighed by the cost of reconstruction. That may be true, but it is worth noting that the examples used in this paper were unusual in the number of closures used and the extent of reconstruction required. Much like static type inference, type reconstruction by the garbage collector is typically much easier than in the worst case. Naturally, experiments are needed to quantify this.

References

[Appel89]
Appel, A.W. Runtime Tags Aren't Necessary. In *Lisp and Symbolic Computation*, 2, 153-162, 1989.

[BL70]
Branquart, P. and Lewi, J. A Scheme of Storage Allocation and Garbage Collection for Algol-68. In *Algol-68 Implementation*, North-Holland Publishing Company, 1970.

[Britton75]
Britton, D.E. *Heap Storage Management for the Programming Language Pascal.* Master's Thesis, University of Arizona, 1975.

[CW86]
Cardelli, L. and Wegner, P. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17,4, 1985.

[Deutsch90]
Deutsch, A. On determining lifetime and aliasing of dynamically allocated data in higher order functional specificiations. *Proceedings of the 1990 ACM Conference on Principles of Programming Languages*, January 1990.

[Goldberg91]
Goldberg, B. Tag-free garbage collection for strongly typed programming languages. *Proceedings of the ACM SIGPLAN'91 Symposium on Programming Language Design and Implementation*, June 1991.

[Goldberg92]
Goldberg, B. Incremental Garbage Collection Without Tags. *Proceedings of the 1992 European Symposium on Programming*, Feb. 1992.

[ISY88]
Inoue, K., Seki, H. , and Yagi, H. *Analysis of Functional Prgrams to detect run-time garbage cells.* ACM TOPLAS, Vol. 10, No. 4, pp 555-578, 1988.

[JLM89]
Jones, S.B. and Le Metayer. Compile-time garbage collection by sharing analysis. *Proceedings of the 1989 Conference on Functional Programming Languages and Computer Architecture*, Sept. 1989.

[Mitchell90]
Mitchell, J.C. Type Systems for Programming Languages. In *Handbook of Theoretical Computer Science*, (J. van Leeuwen, ed.), Elsevier Science Publishers, 1990.

[MLH90]
Milner, R., Tofte, M., and Harper, R. *The Definition of Standard ML.* MIT Press. 1990.

[Park91]
Park, Y.G. *Semantic Analyses for Storage Management Optimizations in Functional Language Implementations.* Ph.D. Thesis, New York University, 1991.

[PJS89]
Peyton Jones, S. L and Salkid, J. The spineless tagless G-machine. *Proceedings of the 1989 Conference on Functional Programming Languages and Computer Architecture*, London, Sept. 1989.