Collecting More Garbage

Pascal Fradet
INRIA/IRISA
Campus de Beaulieu, 35042 Rennes Cedex, France
fradet@irisa.fr

Abstract

We present a method, adapted to polymorphically typed functional languages, to detect and collect more garbage than existing GCs. It can be applied to strict or lazy higher order languages and to several garbage collection schemes. Our GC exploits the information on utility of arguments provided by polymorphic types of functions. It is able to detect garbage that is still referenced from the stack and may collect useless parts of otherwise useful data structures. We show how to partially collect shared data structures and to extend the type system to infer more precise information. We also present how this technique can plug several common forms of space leaks.

1 Introduction

Functional programs tend to be inefficient in their use of store. They usually allocate impressive amount of memory space and spend a significant part of their time garbage collecting. Much work has been done on reducing the overhead of garbage collection. In particular, various sophisticated garbage collectors (GCs) and algorithms have been proposed to this aim [8][29]. However, very little work has been done on extending the collecting power of GCs. In general, they retain the complete reachable structure.

We present a method to detect and collect more garbage than existing GCs. This method is designed for strongly typed languages and it can be seen as an extension of standard GCs for these languages. Our GC is able to detect garbage that is still referenced from the stack and may collect useless parts of otherwise useful data structures. This technique places no overhead on normal execution. It can be applied to strict or lazy higher order languages and can be used to improve different kinds of GCs (stop&copy, mark&sweep, ...). The key property exploited here is parametricity [24][28], a theorem satisfied by Hindley/Milner polymorphic type system [21]. Parametricity can be applied to deduce information on the utility of arguments from the polymorphic type of a function. For example, the function *length* : *List* $\alpha \rightarrow$*Int* can be reduced regardless of the actual elements of the list. This property is just an instance of Wadler's "free theorems" [28]. It holds for all functions of this type and allows a garbage collector to collect the elements of their list argument.

Although our goals are different, our technique shares many common points with tagless garbage collection which we quickly review in section 2. In particular, type information is attached to return addresses and closures, the stack is explored in a bottom-up fashion and the process involves unification. However, if a tagless GC aims at completely reconstructing types, we try on the contrary to minimally instantiate polymorphic types. This point is crucial since only structures associated with a type variable will be collected. We illustrate this difference on an example in section 3.1. We then formalize the collection process and prove its correctness using parametricity theorem (section 3.2). Function *member* seems to invalidate our approach: it has type $\alpha \rightarrow$*List* $\alpha \rightarrow$*Bool* but does need the elements of the list. This comes from polymorphic comparison operators which are treated in section 3.3.

We suggested that *length* can have the elements of its list argument collected but what if this list were shared by a function needing the complete structure? This problem of partial collection of shared structures is addressed in section 4.

Obviously, Hindley/Milner type checking has not been designed as a utility analysis and it loses information useful for our purposes in many cases. Section 5 introduces two simple extensions of the standard typing allowing to infer more suitable (i.e. less instantiated) types. The inference algorithm remains simple and close enough to the standard one.

Many functional programs suffer from space leaks and traditional garbage collection is usually not a big help. Such programs often run out of memory and fail to terminate. This is one of the most interesting applications of our technique and we present in section 6 how it can plug several common forms of leaks. After discussing some implementation issues in section 7, we conclude by a short review of related work.

LISP 94 - 6/94 Orlando, Florida USA
© 1994 ACM 0-89791-643-3/94/0006..$3.50

To simplify the presentation, we often suppose an underlying copy garbage collection scheme. We also assume a stack based implementation in which the GC is invoked only at the beginning of a function and all the roots of the accessible structure are in the stack. The same ideas would adopt to other implementation choices.

2 Tagless Garbage Collection

Statically-typed languages do not need run time tags for normal execution. However their implementations do use tags to support garbage collection and this inflicts a time and space overhead on program execution. The basic idea of tagless garbage collection is to keep the static type information associated with each variable and procedure parameter available at run time. Usually this information is placed in the code (just before the return address) as templates (interpretive method) or as GC routines (compiled method). The GC uses the return address of each activation record on the stack to access the associated information in the code and to determine the type of all the variables of that activation record. It is extended to higher-order languages by associating type information with the function part of closures. The advantage of removing tags is that it saves space and that overhead is placed on garbage collection rather than on normal execution. This method is easy to implement for Pascal-like languages where every variable has a fixed type [6]. Tag-free garbage collection gets more complicated with polymorphically typed languages [2][11]. A polymorphic function is usually implemented by a single code and the template associated with its activation record contains polymorphic types. Relying only on this information the GC would be unable to trace the structures associated with type variables. However, types in an activation record depend on the types of the arguments the function was applied to ; that is, they depend on the types associated with the previous activation record. The solution is to unify the type of the function called with the types of its arguments found in the previous activation record. By traversing the stack from the oldest activation record (corresponding to the top level expression which is not polymorphic) to the most recent, all type variables will be bound using unification.

Example 1 Let us consider the following program and its evaluation using call-by-value.

let rec *append* l_1 l_2 = ***case*** l_1 ***in***

$$\begin{array}{ll} \textit{nil} & : l_2 \\ \textit{cons x xs} & : \textit{cons x (append xs } l_2\textit{)} \end{array}$$

in *length (append [[1];[2]] [[3];[4]])*

append : List α→*List* α→*List* α *length : List* α→*Int*

If the garbage collector is invoked at the beginning of the first call to *cons* the stack looks like

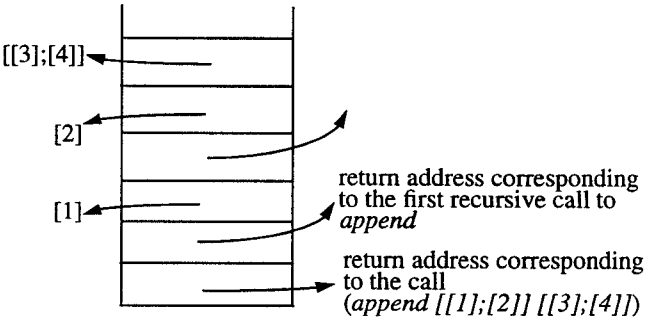

Figure 1

The information associated with a return address includes the types of local variables and the actual types of the arguments of the callee (corresponding to the next activation record). Here the first activation record does not have local variables and the two parameters of the callee (*append*) are of type *List (List Int)*. The GC analyzes the first return address ; no local variables have to be traced and the next return address is analyzed with the information that *append* has been called with two arguments of type *List (List Int)*. The second activation record corresponds to a polymorphic function with two parameters of type *List* α ; it has a local variable of type α and the two parameters of the callee are of type *List* α. Type variable α is unified with *List Int*, the local variable is found to be of type *List Int* and the two parameters of the callee of type *List (List Int)*. The GC traces and copies the local variable according to its type (*List Int*) and continues the exploration of the stack. ❑

This method is described in [12]. There are cases involving higher-order polymorphic functions where types cannot be completely reconstructed during garbage collection. A solution is to generate explicit tagging when necessary [1]. These tags are taken as extra arguments by functions and will be propagated at run time.

3 Collecting More Garbage

3.1 The basic method

Our collecting method is based on the same technique as tagless garbage collection. Type information is associated with return addresses and the GC explores the stack, from the oldest activation record to the most recent one, performing unification. However, our goal is to detect parts of the reachable objects which are unnecessary for a correct execution. Hindley/Milner type checking yields some kind of utility information: a function of type *List* α→*Int* does not need the elements of the list for its execution*. Contrary to tagless garbage collection, we do not attempt to completely reconstruct the types of acces-

* We assume for now a language with no polymorphic comparison operators ; they are considered in section 3.3.

sible objects ; we rather try to minimally instantiate polymorphic types.

A return address can be seen as a continuation of the form $\lambda x_1. \ldots \lambda x_n.\lambda r.\ E$. The x_i's represent the local variables of the activation record and r the result of the function call corresponding to this return address. Thus an activation record can be seen as a closure $(\lambda x_1. \ldots \lambda x_n.\lambda r.\ E)\ X_1 \ldots X_n$ and the type of the function is used by the GC to trace the record.

Our basic method can be described as follows

- The information associated with each call (i.e. return address) is just the type of the continuation. For example, in *length (append [[1];[2]] [[3];[4]])* the continuation of the call to *append* is $\lambda r.length\ r$ which has type $List\ \alpha \rightarrow Int$.
- The GC explores the stack as a tagless GC but does not unify the type of the callee with the types of the actual arguments of the call. It unifies the type expected by the continuation with the result type of the next activation record. When the GC has explored an activation record

  $$(\lambda x_1 \ldots \lambda x_n.\lambda r.\ E)^{\sigma 1 \rightarrow \ldots \rightarrow \sigma n \rightarrow \sigma r \rightarrow \sigma}\ X_1 \ldots X_n$$

  it analyzes the next activation record

  $$(\lambda y_1 \ldots \lambda y_m.\lambda r.\ E)^{\tau 1 \rightarrow \ldots \rightarrow \tau m \rightarrow \tau r \rightarrow \tau}\ Y_1 \ldots Y_m$$

  by unifying the type τ of this record with the type σ_r expected by the previous record (i.e its continuation).

Example 2 Let us return to our previous example where the GC was invoked during the first call to *cons*. At this point the stack can be represented as follows

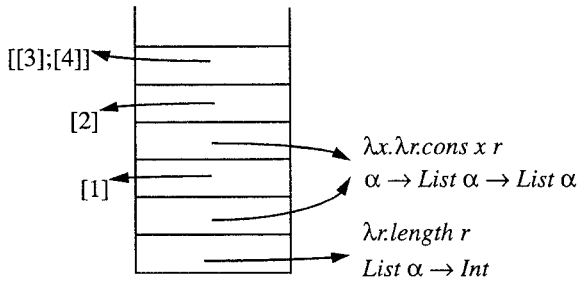

Figure 2

The oldest activation record has type $List\ \alpha \rightarrow Int$ and no local variables, the GC analyzes the next return address which has type $\alpha \rightarrow List\ \alpha \rightarrow List\ \alpha$ (the unification with the type expected by $\lambda r.length\ r$ leaves it unchanged). The argument *([1])* in this activation record has type α and is collected. The GC analyzes the last return address ; its type $\alpha \rightarrow List\ \alpha \rightarrow List\ \alpha$ allows the GC to reclaim the first argument *([2])* and the elements of the second. All the sublists *([1], [2],[3],[4])* have been collected and the new heap contains only two cons cells and a nil. ❑

3.2 Formalization

We represent a stack of activation records as applications of closed functional expressions:

$$(\lambda x_1. \ldots \lambda x_m.\lambda r.\ E)\ X_1 \ldots X_m\ ((\lambda y_1. \ldots \lambda y_n.\lambda r.F)\ Y_1 \ldots Y_n\ (\ldots \ldots\ ((\lambda z_1. \ldots \lambda z_p.G)\ Z_1 \ldots Z_p) \ldots))$$

The functions $(\lambda x_1. \ldots)$, $(\lambda y_1. \ldots)$ represent return addresses and the last function $(\lambda z_1. \ldots)$ is the function which has invoked the GC. The arguments X_i, Y_j, Z_k contains the roots of the accessible structure and belong (as the stack itself) to the following set of expressions:

Objects $S ::= V \mid S_1.S_2 \mid nil \mid f\ S_1 \ldots S_n$

where V represents basic values (integers, booleans, ...), $E_1.E_2$ and *nil* lists and $f\ S_1 \ldots S_n$ closures (f being a function (i.e. code)). We consider only closures and lists but the approach can be readily extended to deal with user-defined types. The associated types are defined by

Types $\mathcal{T} ::= \mathcal{V} \mid \mathcal{B} \mid \mathcal{T} \rightarrow \mathcal{T} \mid List\ \mathcal{T}$ $\quad$ $\mathcal{P} ::= \mathcal{B} \mid List\ \mathcal{P}$

where $\mathcal{V}$ represents type variables and $\mathcal{B}$ basic types (*Int, Bool,...*). $\mathcal{P}$ represents printable values and we assume that types of programs (i.e. stacks) belong to $\mathcal{P}$. We use the following conventions:

$$\alpha,\beta,\chi,\delta,\varepsilon \in \mathcal{V} \quad b \in \mathcal{B} \quad \tau,\sigma \in \mathcal{T} \quad \pi \in \mathcal{P}$$

The stack in Example 2 can be represented by the expression of type *Int*

$$f_1(f_2\ [1]\ (f_2\ [2]\ [[3];[4]]))$$

with $f_1 \equiv \lambda r.length\ r$ and $f_2 \equiv \lambda x.\lambda r.cons\ x\ r$

The intuition that polymorphic functions do not need the complete structure of their arguments for proper execution is formalized by Reynolds' abstraction or parametricity theorem [24]. Wadler has shown in [28] how to use parametricity to derive theorems from types. Actually those theorems are just what we need to justify our approach. For example, the fact that a function f of type $List\ \alpha \rightarrow Int$ can be reduced regardless of the elements of the list is formalized by its associated "free" theorem: $\forall a: A \rightarrow A'\ \ f = f \circ (map\ a)$. In the theorems from types, functions (here a) are associated with type variables and the theorems hold (in the pure polymorphic λ-calculus) whatever these functions are. However, for practical languages, where a fixed point operator is added as primitive, the theorems hold only for strict functions.

Here, we formalize the collection process by associating with each type variable the strict function $\lambda x.\perp$. For example, the theorem deduced from type $List\ \alpha \rightarrow Int$ implies that

$$f = f \circ (map\ (\lambda x.\perp))$$

so f can have the list elements of its argument reclaimed by the GC.

The collecting process of a stack of monomorphic type π can be described as gc_π *stack* with

(gc1) $gc_\alpha(E) = \bot$

(gc2) $gc_\tau(f\,X_1 \ldots X_n) = f\,(gc_{\tau 1}\,X_1) \ldots (gc_{\tau n}\,X_n)$
with $\vdash f{:}\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau$

(gc3) $gc_b(V) = V$

(gc4) $gc_{\mathrm{List}\,\tau}(E.F) = gc_\tau\,E\,.\,gc_{\mathrm{List}\,\tau}\,F$

(gc5) $gc_{\mathrm{List}\,\tau}(nil) = nil$

An important point to note is that in order to carry on garbage collection (gc_τ) inside a closure the current type information τ must be the type of the closure (rule (gc2)). In practice, as with activation records, this property is ensured by unification. When a closure is encountered, gc_τ takes the type information associated with the function of the closure, say $\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau_{n+1} \rightarrow \ldots \rightarrow \tau_p$; from the numbers of components, say n, of the closure it deduces the result type, $\tau_{n+1} \rightarrow \ldots \rightarrow \tau_p$ which is unified with τ. So, activation records, higher-order functions and unevaluated arguments, which are all represented by closures, involve preliminary unifications.

Example 3 Let us return to our previous example

$stack \equiv f_1(f_2\ [1]\ (f_2\ [2]\ [[3];[4]]))$

with $f_1 \equiv (\lambda r.length\ r){:}List\ \alpha \rightarrow Int$ and $f_2 \equiv \lambda x.\lambda r.cons\ x\ r$

$gc_{\mathrm{Int}}\ stack = f_1\ (gc_{\mathrm{List}\,\alpha}(f_2\ [1]\ (f_2\ [2]\ [[3];[4]])))$ (gc2)
$\vdash f_1{:}List\ \alpha \rightarrow Int$

$= f_1\ (f_2\ (gc_\alpha[1])\ (gc_{\mathrm{List}\,\alpha}(f_2\ [2]\ [[3];[4]])))$ (gc2)
$\vdash f_2{:}\alpha \rightarrow List\ \alpha \rightarrow List\ \alpha$

$= f_1\ (f_2\ \bot\ (f_2\ (gc_\alpha[2])\ (gc_{\mathrm{List}\,\alpha}[[3];[4]])))$ (gc1),(gc2)

$= f_1\ (f_2\ \bot\ (f_2\ \bot\ ((gc_\alpha[3]).(gc_{\mathrm{List}\,\alpha}[[4]])))))$ (gc1),(gc4)

$= f_1\ (f_2\ \bot\ (f_2\ \bot\ [\bot;\bot]))$ (gc1),(gc4),(gc5)

If the stack was $f_1(f_3\,f_4\ [(1,2);(3,4)])$

with $f_3 \equiv \lambda f.\lambda l.map\ f\ l : (\beta \rightarrow \delta) \rightarrow List\ \beta \rightarrow List\ \delta$

and $f_4 \equiv \lambda x.(fst\ x)+1 : (Int,\varepsilon) \rightarrow Int$

then α and δ would be unified with *Int* and β would be unified with (Int,ε). Only the second component of pairs would be reclaimed (i.e $gc_{\mathrm{Int}}\ stack = f_1\ (f_3\,f_4\ [(1,\bot);(3,\bot)])$). ❐

Activation records are basically treated the same way as closures. However, with a strict semantics, there may be functions which do not use the result of the next activation record, that is functions of type $\tau_1 \rightarrow \ldots \rightarrow \alpha \rightarrow \tau$. As specified, the function gc_τ would collect all the activation records above. We do not allow this and consider only garbage collection of heap objects. The following proposition states the correctness of the collection.

Property 4 *For all closed stack of type* π $stack = gc_\pi\ stack$

Proof. [sketch] For brevity sake, we do not redescribe the parametricity property, how to read types as relations and to derive theorems ; the reader is referred to [28]. In our case we fix the functions (relations) corresponding to types variables to be $\lambda x.\bot$. We first prove, by structural induction, that for any closed stack object S and any type τ for which $gc_\tau\ S$ is defined then $(S, gc_\tau\ S) \in T$, T being the relation corresponding to τ. Furthermore, it is easy to show that the relation corresponding to a type π in $\mathcal{P}$ is the identity relation hence $(S, gc_\pi\ S) \in I$ that is $S = gc_\pi\ S$ ❐

3.3 Polymorphic comparison operators

In Example 3, if we replace the function $\lambda r.length\ r$ by

$\lambda r.if\ member\ (hd\ r)\ (tl\ r)\ then\ 1\ else\ 0$

of the same type $List\ \alpha \rightarrow Int$ our collecting function gc_τ becomes incorrect. The polymorphism comes here from the polymorphic equality operator in *member* which has type $\forall\alpha.\ \alpha \rightarrow \alpha \rightarrow Bool$ but does need the value of its arguments. Polymorphic comparison operators cannot be defined in the pure polymorphic λ-calculus and parametricity does not hold with such operators. In [28] Wadler gives polymorphic equality a special type, $\forall^{=}\alpha.\ \alpha \rightarrow \alpha \rightarrow Bool$, and enforces that function a associated with the type variable α respects equality (i.e. $x=y$ iff $a\ x = a\ y$). Our collecting function certainly does not respect equality and theorems derived for expressions involving polymorphic equality are useless for our purposes. We have to assume that polymorphic comparison operators need completely their arguments and we give them type $\mathsf{T} \rightarrow \mathsf{T} \rightarrow Bool$. Special type T belongs to $\mathcal{P}$ and means that the complete structure has to be saved. In order to propagate this information, the unification of any (non functional) type with T yields T. The type inferred for $\lambda r.if\ member\ (hd\ r)\ (tl\ r)\ then\ 1\ else\ 0$ is now $\mathsf{T} \rightarrow Int$ and its list argument will be preserved by the GC.

4 Partial Collection of Shared Structures

We assume that all reachable objects can be traced by the GC. For now, we also assume that programs cannot create circular structures. We consider this point in section 4.3.

4.1 The problem

The collection is done according to the type information (after the necessary unifications) associated with each pointer in the stack or in closures. For example, a pointer with type information α does not have to be traced and the elements of a structure with type information *List* α can be reclaimed. A problem arises when such structures are shared. For example, in Figure 3 when the first list L_1 with type *List* α is traced only its spine is copied. The type *List* (*List* β) enforces the GC to copy the spine of the sublists of L_2. When the shared cell is found, the GC cannot rely on the standard assumption that all its descendants have been copied and must traverse it again.

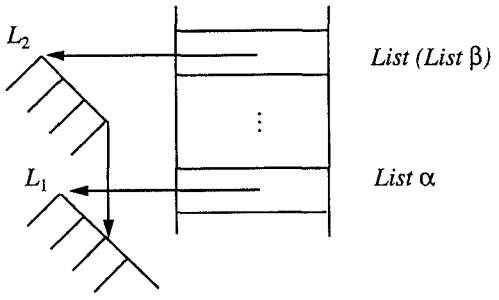

Figure 3

A simple solution would be to keep within each cell the type information it was encountered with. The standard assumption would become: if a copied cell has an associated type more instantiated than the current one, the garbage collector does not have to trace its fields. Still, the same structure may be traced many times. For example, a list of tuples $(i_1,...,i_n)$ shared by n pointers with associated types *List* $(Int,\alpha_2,...,\alpha_n),...,List(\alpha_1,...,\alpha_{n-1},Int)$ would still have to be traversed n times.

4.2 The basic technique

This problem can be overcome using two *complete* scans of the reachable structure. That is, useless substructures are not copied but they are traversed anyway. Each structure will not be copied more (nor less) than required by the type information. For example, if the lists represented in Figure 3 are not further shared, the first two sublists of L_1 will be considered as garbage and reclaimed, the remaining structure will be copied according to type *List* (*List* β). We just sketch the technique here ; section 7 contains further details on both scans.

The first scan computes for each vertex its reference counter. Each shared vertex is placed in a defer-list (i.e. a shared cell will point to the defer-list) along with an initial type (say α) and its reference counter. The second scan performs the partial collection according to types. Each pointer in the stack is followed: if a vertex is not shared then its fields are traced ; otherwise (i.e the vertex points to the defer-list) its reference counter is decremented, the type in the defer-list is unified with the current one and the exploration continues with another root. The type associated with each pointer represents the utility of the structure for a particular activation record. The unification of types for a shared cell eventually yields the type representing the utility of this structure for the whole program. When all the roots have been followed as much as possible we still have to trace the pointers in the defer-list. Assuming no cycles, the defer-list must contain a vertex with a reference counter equal to zero. We call it a *resolved* vertex : all the pointers referencing it have been found and the maximally instantiated type has been computed. Its fields can then be traced according to this type ; another resolved vertex is taken from the defer-list and so on until the defer-list is empty. So, a cell is copied only when all the structures pointing to it have been traced (as a side effect, forwarding pointers is not necessary).

Property 5 *When the active (i.e. not pointed from the defer list) reachable structure has been traced, the defer list is either empty (the collection is complete) or contains a resolved vertex.*

Let the defer-list contain n vertices, which are the roots of the remaining unexplored structure. A vertex N_1 is not resolved iff there is a pointer in the remaining structure pointing to it i.e. there must a path from a vertex in the defer list to N_1. Since there is no cycle it cannot be N_1, let us call it N_2. Applying the same reasoning on N_2 it must be either resolved (and we are done) or there must be a vertex leading to N_2 and it must be different from N_1 and N_2. And so on until the last vertex N_n: assuming no cycle, $N_1,...,N_{n-1}$ cannot lead to N_n, and N_n cannot lead to itself so N_n must be resolved. ❑

Let us take an example to illustrate why we cannot avoid traversing substructures associated with type variables. In Figure 3, such a partial traversal, starting from the first root, would only mark the spine of L_1 and, starting from the second root, would stop at the shared cell. The following sublists will not be explored and shared vertices may stay unannotated. Imagine that another list say L_3 shares the fourth sublist of L_1 with type annotation *List Int*: it will not be detected as shared. During the second phase *List* α and *List* (*List* β) will be unified in the defer-list, the fourth sublist of L_1 will be copied with type information *List* β whereas its integers elements are needed by L_3. For the same reason the second scan must be complete. The first scan may have marked structures shared by pointers of type α. In order to resolve such vertices, useless substructures annotated with type variables are not copied but are traversed anyway.

4.3 Extensions

Optimizations. We tried to keep the partial collection process simple and did not mention several possible optimizations.

- Structures associated with a monotype can be copied as soon as encountered and pointers forwarded as usual. No reference count has to be computed ; if such a structure is shared we can rely on the standard assumption (all its fields have been traced and copied).
- The roots in the stack annotated by a type variable do not have to be traced. They cannot be shared and can just be ignored during the two scans.
- Functional compilers sometimes integrate a sharing analysis to perform safe destructive updates or to avoid updating unshared closures. The collection process can benefit from such an analysis. An unshared object can be copied according to its type right away.

Cycles. Many implementations of pure functional languages do not create circular structures and for those our method directly applies. Implementations based on graph reduction sometimes implement recursion (combinator **Y**) using cycles. This would require an extension of the partial collection process. We have not investigated this issue thoroughly, however two trivial solutions come to mind. The strong components of a graph can be computed in linear time [26]. We can then use our technique on the reduced graph (whose vertices are the strong components). We also have a conservative option: we apply our technique as usual ; if there are cycles we end up with a defer-list with only unresolved vertices which are completely copied.

5 Detecting More Garbage

In many cases the standard polymorphic typing loses utility information. Here, we propose two extensions of the standard typing allowing to infer more information on the utility of arguments and so to collect even more garbage. We suppose that programs are well-typed and we use type inference as a utility analysis. The inference algorithm remains simple and close enough to the standard one.

5.1 First extension

A first loss of information comes from the standard types of constructors. For example, from the type of *cons* ($\alpha \to$*List* $\alpha \to$*List* α), the GC deduces that *cons* needs the structure of its second argument for a proper reduction. However the arguments of a *cons* does not have to be traced by the GC if the result is useless. Also, both λ*xs.cons 1 xs* and λ*x.cons x [1;2]* have a monomorphic type (*List Int*$\to$*List Int* and *Int*$\to$*List Int*) whereas their argument could be (partially) collected if the enclosing function needed only the spine of the result.

We avoid this loss of information by giving *cons* type $\alpha \to \beta \to \delta$ along with the constraint set {*List* $\alpha \gg \delta$, $\beta \gg \delta$} where the relation "$\gg$" reads "is more instantiated than". In doing so, we have to distinguish constructors used as patterns which keep their standard types.

Let us consider the closure (λ*x.*λ*xs.cons x xs*) L_1 L_2. The type information associated with its function is [$\alpha \to \beta \to \delta$;{*List* $\alpha \gg \delta$, $\beta \gg \delta$}]. During garbage collection type δ is unified with a type representing how much the value of the closure is needed (see section 3) ; the utility of the arguments will be determined by satisfying the constraints.

- If the result of the closure is not needed by the context (i.e. δ has been unified with another type variable), the constraints are satisfied and the two arguments can be collected.
- If the context needs the spine of the result, say δ=*List* γ, the constraint set enforce β to be more instantiated than *List* γ ($\beta \gg$*List* γ). In order to satisfy the constraints, β and List χ (with $\chi \gg \gamma$) will be unified. The function has then type $\alpha \to$*List* $\chi \to$*List* χ and the first argument and the elements of the second are collected.
- If the context needs the complete structure of the result, say δ=*List Int*, in order to satisfy the constraint, α is unified with *Int* and β with *List Int*. The closure function has then type $\alpha \to$*List Int*$\to$*List Int* and both arguments are completely saved by the GC.

Example 6 Let us consider the function

rec *app* l_1 l_2 = ***case*** l_1 ***in***

nil : l_2

cons x xs : ***if*** *x=0* ***then*** *app xs* l_2

else *cons x (app xs* l_2*)*

The type of this function is *List Int*$\to\beta\to\delta$ with {$\beta \gg \delta$}. If, during garbage collection, the stack is

(λ*r.length r*) (($\lambda l_1.\lambda l_2$.*app* l_1 l_2) *[0;1] [2;3]*)

then δ and *List* α are unified, β is unified with *List* γ and the elements of second list can be reclaimed. The standard typing of *app* (*List Int*$\to$*List Int*$\to$*List Int*) would have enforced the GC to retain completely both lists. ❐

Our constraints can be seen as a way to delay unification until necessary. A type and a constraint set are inferred individually for each function representing a return address. If the function has type σ using the classic type inference, the type τ inferred now is such that $\sigma \gg \tau$. Satisfying such inequalities is closely related to the semiunification problem (i.e. finding a substitution, called a semiunifier, *S* such that $S\sigma \gg S\tau$ for all inequalities) [13]. Since we consider programs already typed by the Hindley/Milner system, this problem, undecidable in general, becomes tractable. The relation "$\gg$" is defined only on unifiable types, so our systems of inequations are unifiable and therefore semiunifiable. Constraint sets are satisfied using the following rules.

$S \cup \{\pi \gg \tau\} \to S$ $\qquad$ $S \cup \{\alpha \gg \alpha\} \to S$

$S \cup \{$*List* $\sigma \gg$ *List* $\tau\} \to S \cup \{\sigma \gg \tau\}$

$S \cup \{\sigma \to \sigma' \gg \tau \to \tau'\} \to S \cup \{\sigma \gg \tau\ ;\ \sigma' \gg \tau'\}$

Furthermore, variables on the lhs of constraints of the form $\alpha \gg \pi$, $\alpha \gg \tau \to \tau'$ and $\alpha \gg$*List* τ are unified with respectively π, $\alpha_1 \to \alpha_2$ and *List* α_3 (α_1,α_2 and α_3 being fresh variables).

This is actually a simplified version of the algorithm described in [13]. It has been proved that for semiunifiable inequations this algorithm terminates and finds a most general semiunifier. A simpler alternative is to unify the lhs and rhs of constraints of the form $\alpha \gg \tau \to \tau'$ and $\alpha \gg$*List* τ instead of introducing fresh variables. The resulting types would possibly be more instantiated than needed but the complexity of unification would remain linear. We considered only *cons* but similar constraint sets can also be associated with each user-defined constructor.

5.2 Second extension

Another imprecision comes from the data types themselves which identify many different structures (e.g. singletons or empty lists are not distinguished from general lists). For example, the function returning the head of a list is defined as

$$\begin{array}{l} hd\ l = \textbf{\textit{case}}\ l\ \textbf{\textit{in}} \\ \qquad cons\ x\ xs\ : x \qquad (*) \\ \qquad otherwise : fail \end{array}$$

The inferred type for *hd* is *List* $\alpha \rightarrow \alpha$ and we have lost the information that only the first element of its argument list has to be saved by the GC. Instead of using predefined types, we infer more precise recursive types. Trying to infer very precise types quickly leads to undecidable problems. Here, we are only interested in inferring types as less instantiated as possible and we stay close from the standard algorithm. Recursive data types are noted explicitly using the operator μ. For example, *List* α becomes ($\mu t.\ nil + cons\ \alpha\ t$). To unify recursive data types we have to replace standard unification by unification of rational trees [14] (both have the same complexity).

Let e an expression with the Hindley/Milner type t defined by $type\ t = C_1\ \sigma_{11}...\sigma_{p1} +...+ C_n\ \sigma_{1n}\ ...\sigma_{pn}$, then the new typing of e will be of the form $\mu t.\tau_1 +...+ \tau_n$, τ_i being a type variable or a constructor $C_i\ \upsilon_{11}\ ...\upsilon_{p1}$. We suppose that recursive data structures can only be scrutinized by pattern matching in case expressions and that patterns are simple one-level patterns.The typing rule for a saturated case expression becomes

$$\frac{\text{for all } i,\ 1 \le i \le n,\quad \Gamma \vdash e : \mu t.\ C_1\ \tau_{11}...\tau_{p1} +...+ C_n\ \tau_{1n}...\tau_{pn} \qquad \Gamma \cup \{x_j : \tau_j\}_{j=1i}^{pi} \vdash e_i : \tau}{\Gamma \vdash (case\ e\ in\ C_1\ x_{11}...x_{p1} : e_1\ \ ...\ \ C_n\ x_{1n}...x_{pn} : e_n) : \tau}$$

If one alternative corresponding to the user-defined type is missing (e.g. $hd\ l = \textbf{\textit{case}}\ l\ \textbf{\textit{in}}\ cons\ x\ xs : x$) or an alternative is a default variable not occurring in the rhs (as *otherwise* in the definition above (*)) then it will be associated with a type variable. The type inferred for *hd* is ($\mu t.\ \delta + cons\ \alpha\ \beta$) $\rightarrow \alpha$. When the GC encounters a closure ($\lambda l.hd\ l$) L, the tail of the list will be reclaimed (assuming it is not shared).

Example 7 $\begin{array}{l} rec\ f\ l = \textbf{\textit{case}}\ l\ \textbf{\textit{in}} \\ \qquad nil \qquad : 0 \\ \qquad cons\ x\ xs : \textbf{\textit{case}}\ xs\ \textbf{\textit{in}} \\ \qquad\qquad nil \qquad : 0 \\ \qquad\qquad cons\ y\ ys : y + f\ ys \end{array}$

The type inferred for f is

$$(\mu t.\ nil + cons\ \alpha\ (nil + cons\ Int\ t)) \rightarrow Int$$

and elements occurring at odd places in the list may be reclaimed. ❐

This extension is related to [22] which aims at inferring types without explicit type declarations. Their type inference produces accurate types and we could make use of them. However it is also more general than we need: their type system accepts constructors to be overloaded and the sum of two arbitrary types to be a type. This complicates their type inference which is defined only for first-order languages.

As with the first extension, if the function has type σ using the classic type inference, the type τ inferred is such that $\sigma \gg \tau$. It seems that these two extensions can easily be combined. Relation "»" can be extended on recursive data types with the rules

$$S \cup \{\mu t.\ \sigma_1 +...+ \sigma_n \gg \mu t.\ \tau_1 +...+ \tau_n\} \rightarrow S \cup \{\sigma_1 \gg \tau_1\ ;\ ...\ ;\ \sigma_n \gg \tau_n\}$$

$$S \cup \{C\ \sigma_1\ ...\ \sigma_n \gg C\ \tau_1\ ...\ \tau_n\} \rightarrow S \cup \{\sigma_1 \gg \tau_1\ ;\ ...\ ;\ \sigma_n \gg \tau_n\}$$

Still, it would be necessary to formalize the corresponding type system and to prove the analogue of Property 4 (parametricity does not apply directly for such types systems). We could have envisaged more sophisticated (e.g. semantics based) analyses to detect the uselessness of data structures. We claim that the two extensions proposed above are a good compromise between cost and precision.

6 Space leaks

Some functional programs use much more space than the programmer would expect. This phenomenon, called a space leak, usually appears when pointers are kept on structures which have become (partially) useless. A traditional GC preserves those objects and seemingly innocent programs may run out of heap space and fail to terminate [15][23]. We present here three common forms of space leaks easily fixed using our garbage collection technique.

Recursive functions. In many implementations, a recursive call involves pushing a new context on the stack, the old context being kept for the continuation of the call. If the continuation does not use all the arguments in the context this may cause a space leak. For example, the code generated for

$$f\ x\ l = ...\ else\ x + f\ (x\text{-}1)\ (tl\ l)$$

should not keep the argument l in the stack during the recursive call. Reorganizing the stack before function calls is costly. Usually the solution is to overwrite with a special constant (a hole) arguments when they become useless. Still, this "blackholing" induces an overhead at run time.

In our framework, types of continuations give us enough information about the utility of the different arguments and no blackholing is necessary. In the preceding example, the continuation $\lambda x.\lambda l.\lambda r.x+r$ has type $Int \rightarrow \alpha \rightarrow Int \rightarrow Int$ and the useless part of l would be reclaimed if the GC was triggered during a recursive call to f.

Updatable Closures. A similar problem occurs in implementations of lazy languages. During the evaluation of a closure, a pointer on the closure is kept in order to update it later by its value. The same technique of blackholing is needed to avoid space leaks [18]. Here, we make the update explicit using an operator *updt* with type $\mathbf{U}\rightarrow\beta\rightarrow\beta$; **U** being a special type associated with the pointer needed for the update and β the type of the result. Structures with types **U** are not traced but the GC replaces them by a "blackhole" closure. This closure serves to retain enough space for the update and to detect certain forms of non-termination if it is accessed before the update [18]. In order to deal with shared closures the unification of any type with **U** yields **U**.

Tuples. Another class of space leaks has been described by Hughes [15]. In lazy functions returning tuples, the result, say *r*, is often retained in expressions such as (*fst r*) or (*snd r*). Hughes showed that some of these functions are inherently leaky, no matter how they are expressed. One solution, proposed by Wadler [27], is to modify the GC to perform the simplification rules *fst(x,y)=x* and *snd(x,y)=y*. In our approach, space leaks treated by Wadler's technique are naturally avoided. The type of *fst* (resp. *snd*) being $(\alpha,\beta)\rightarrow\alpha$ (resp.$(\alpha,\beta)\rightarrow\beta$), our GC will reclaim its second (resp. first) argument.

Tables. When using structures like hash tables or memo tables it is important to be able to delete accessible but useless entries [10][16]. This is usually done by introducing a notion of weak pointers which are treated distinctively by the GC. Each table entry is a weak pointer to a structure which will be copied iff it is also referenced by a strong pointer. In our framework this might be done simply by fixing the type of such tables to be *table*[**W**]. Type **W** acts like a type variable except that it is not pointers associated with **W** but the structure they point to which are replaced by $\perp$. If such a structure is shared by a (strong) pointer, the unification in the defer-list will enforce the GC to retain it.

Note that we have fixed the different forms of space leaks presented here without resorting to extensions of section 5. This method does not entail any overhead at run time and other kinds of leaks (e.g. related to lists) can also be plugged.

7 Implementation issues

We are currently integrating this extension into our transformation based compiler [9]. This compiler transforms expressions into functional terms which can be seen as a stack-based machine code. It can compile strict or lazy languages ; its standard GC is a simple stop&copy and pointers are distinguished from values using a tag bit (as in the SML New Jersey compiler [3]). We saw in section 6 that some low level information (like updates or stack layout during a recursive call) must be taken into account in the type information. We choose to perform a new type inference on the functional machine code produced by our compiler. It could also be done at the same time as the type checking of source expressions but this information should be carried along all compilation steps. Type information is placed in the code just before the address it is attached to.

When the extended GC is triggered it first computes the reference counts of shared nodes. This first scan is a depth first traversal of the live data graph (using the to-space as an explicit stack). Shared cells point to the defer-list which contain their reference count and an initial type. It is well known that typically few cells are shared so we may hope that the defer-list remains reasonably small. Anyhow, we may fix its size and treat overflows conservatively by copying completely the remaining shared structures. This is one advantage of our approach: when it becomes too costly or complicated we can still rely on standard techniques (standard types or standard garbage collection).

The second scan examines the stack in a bottom-up fashion in order to perform unification and the copy in the to-space. With first-order programs local variables can be traced as soon as the type information associated with the activation record is read and unified. The space needed by the unification remains small since a type becomes useless when it has been unified with the type of the next activation record. However higher-order programs may involve many unifications before tracing. For example, in the activation record

($\lambda a.\lambda l.\lambda r.$ (last l) a) X [length;length;...;sum] ...

the local variable *X* can only be traced after unifying the type of all the functions of the list. All the necessary unification could be done during the first scan but (potentially large) memory space would be needed to store the substitution. We take a pragmatic approach and limit the number of unifications by activation record. This is sufficient to cope with most common uses of higher-order functions but, for example, functions in lists of functions will be assumed to need completely their arguments. The tracing of structures according to type information is a depth-first traversal using Shorr-Waite algorithm which uses a link reversal technique to avoid the need for a stack [25]. Pointers on useless structures are replaced by a special constant (the $\perp$ of section 3).

The GC process is not directly concerned by the bad worst case complexity of ML-like type inference [20]. It only uses unification which has a linear time complexity. However, types can be of exponential length and, for those pathological cases, the type annotations generated by the compiler should be approximated (e.g using $\top$ as in section 3.3).

This approach can be used with several garbage collection schemes [8]: a mark&sweep GC does not mark structures associated with type variables whereas a stop&copy GC does not copy them. The method mixing reference counts with classical garbage collection (usually performed as a last resort) can also benefit from it. In this case, the first scan can be avoided.

It is clear that our technique is more costly than traditional garbage collection. In particular, it involves unification and the complete structure is scanned twice. On the other hand, it can make leaky or greedy programs terminate. Also, reclaiming more space might prevent or simplify further collections. The policy we advocate is to use a regular GC most of the time ; our extension would be used, from time to time, when heap occupancy exceeds a certain ratio or, at least, as a last resort when the program runs out of memory. For example, in a generational GC, a standard copy algorithm would take care of the youngest generations and the extension would be used for the occasional major collections.

Since our GC can fix different sorts of space leaks, it is not difficult to exhibit programs for which this technique saves arbitrarily large amounts of heap space. Still, it would be interesting to estimate the savings on a wide range of programs. At the moment, our implementation is still in progress and we cannot provide a full set of benchmarks. However we have completed a first incomplete prototype: it deals only with first order strict functional programs and does not implements the extension of section 5.1. Figure 4 gathers the results obtained on a few programs (not written for this purpose): *mirror* is a simple program on trees, *queen* is the usual 10 queens problem, *fft* is a fast Fourier transform applied to multiplication of polynomials and *compress* is a text compression program.

Minimum Heap Space Requirement (call-by-value)				
	mirror	*queen*	*fft*	*compress*
Regular GC	468 Kb	564 Kb	57 Kb	1,280 Kb
Extended GC	352 Kb	404 Kb	42 Kb	792 Kb
Gain	25 %	28 %	26 %	38 %

Figure 4 A few results

These preliminary results are encouraging ; of course we have also encountered programs (e.g. *qsort*) where almost nothing was gained. It is too early to have a precise idea of the cost of this technique. One important optimization, that we still have to implement, is to store the type information in compiled form (i.e. a routine performing the unification and the tracing instead of a template representing the type). In most cases our (unoptimized) prototype is "only" 3 to 4 times slower than the standard stop&copy although in a very restricted context (first-order & call-by-value). In any case, several simplified versions (e.g. with no typing extensions or dealing only with certain forms of space leaks) should be provided as well ; they could be used depending on the price the user is willing to pay.

8 Conclusion

We have proposed a method to collect more garbage for polymorphically typed languages. It is based on parametricity of polymorphic functions and can be applied to strict or lazy higher order functional languages. As tagless garbage collection, the technique needs to attach type information to closures and return addresses. The overheads are placed on garbage collection and on code space (to store type information) but not on normal evaluation. The GC is able to detect garbage that is still referenced from the stack and may collect useless parts of otherwise useful data structures. The partial collection of shared structures is not straightforward and we have described a solution which retains the linear time complexity of garbage collection. We have proposed two extensions of the type system in order to detect more garbage and presented how our GC could plug several forms of space leaks. More sophisticated utility analysis could be designed ; as long as the information produced can be coded into types our technique would still apply. A peculiarity of this approach is to mix a static analysis (typing) and a run time analysis. We benefit from run time information by exploring the stack which describes a more specific program and the heap which provides exact sharing information.

We are not aware of any other general approach to collect useless reachable structures. In [11], Goldberg mentions that a tagless GC could use a live variable analysis to collect dead variables of activation records. In our framework, this is done by reclaiming roots associated with type variables. In [27], Wadler suggests an extension specific to tuples and we saw in section 6 how this is done in our approach. There also exist garbage collectors, adapted to non-deterministic languages, which detect and collect the useless binding values of useful logic variables [5].

Compile-time garbage collection is a static analysis which detects points in the program where part of the store can be collected [17][19]. This information can then be used to reallocate old store. It reduces store use and therefore reduces garbage collection overhead. In the best cases those analyses detect what a traditional GC would detect at run time. We see this approach and ours as complementary.

The most common optimization allowed by Hindley/ Milner type system is to avoid run time checks. In [4], Baker shows how to use ML-like type inference for sharing analysis. We have shown in this paper that polymorphic types can also be used to improve garbage collection. Searching for other applications of ML-like type inference is certainly worthwhile since, contrary to many program analyses, this algorithm is practical as everyday experience shows.

Acknowledgments. Thanks to Daniel Le Métayer for commenting an earlier version of this paper and to Olivier Ridoux for enlightening discussions and useful suggestions.

References

[1] S. Aditya and A. Caro. Compiler-directed type reconstruction for polymorphic languages. In *Proc. of the ACM Conf. on Func. Prog. and Comp. Arch.*, pp. 74-82, 1993.

[2] A.W. Appel. Runtimes tags aren't necessary. *Lisp and Symb. Comp.*, 2, pp. 153-162, 1989.

[3] A.W. Appel. A runtime system. *Lisp and Symbolic Computation*, 3, pp. 343-380, 1990.

[4] H.G. Baker. Unify and conquer (garbage, updating, aliasing, ...) in functional languages. In *Proc. of the ACM Conf. on Lisp and Functional Programming*, pp. 218-226, 1990.

[5] Y. Bekkers, O. Ridoux and L. Ungaro. Dynamic memory management for sequential logic programming languages. In *Proc. of Work. on Memory Management*, LNCS 637, pp. 82-102, 1992.

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

[7] C.J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11), pp. 677-678, 1970.

[8] J. Cohen, Garbage collection of linked data structures. *Computing Surveys*, Vol. 13, 3, 1981.

[9] P. Fradet and D. Le Métayer. Compilation of functional languages by program transformation. *ACM Trans. on Prog. Lang. and Sys.*, 13(1), pp. 21-51, 1991.

[10] D.P. Friedman and D.S. Wise. Garbage collecting a heap which includes a scatter table. *Inf. Proc. Letters*, 5(6), pp. 161-164, 1976.

[11] B. Goldberg. Tag-free garbage collection for strongly typed programming languages. In *Proc. of the ACM SIGPLAN'91 Symp. on Prog. Lang. Design and Implementation*, pp.165-176, 1991.

[12] B. Goldberg and M. Gloger. Polymorphic type reconstruction for garbage collection without tags. In *Proc. of the ACM Conf. on Lisp and Func. Prog.*, pp. 53-65, 1992.

[13] F. Henglein. Type inference with polymorphic recursion. *ACM Trans. on Prog. Lang. and Sys.*, 15(2), pp. 253-289, 1993.

[14] G. Huet. *Résolution d'équations dans les langages d'ordre 1, 2,...ω.* Thèse de doctorat d'état, Université de Paris VII, 1976.

[15] J. Hughes. *The design and implementation of programming languages.* D. Phil. Thesis, Oxford University, 1983.

[16] J. Hughes. Lazy memo-functions. In *Proc. of the ACM Conf. on Func. Prog. and Comp. Arch.*, pp.129-146, LNCS 201, 1985.

[17] K. Inoue, H. Seki and H. Yagi. Analysis of functional programs to detect run-time garbage cells. *ACM Trans. on Prog. Lang. and Sys.*, 10(4), pp. 555-578,1988.

[18] R. Jones. Tail recursion without space leaks. *Journal of Func. Progr.*, 2 (1), pp. 73-79, Jan. 1992.

[19] S.B. Jones and D. Le Métayer. Compile-time garbage collection by sharing analysis. In *Proc. of the ACM Conf. on Func. Prog. and Comp. Arch.*, pp.54-74, ACM Press, 1989.

[20] H.G. Mairson. Deciding ML typability is complete for deterministic exponential time. In *Proc. of the 17th ACM Conf. on Princ. of Prog. Languages*, pp. 382-401, 1990.

[21] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci. 17*, pp. 348-375, 1978.

[22] P. Mishra and U.S. Reddy. Declaration-free type checking. In *Proc. of the ACM Conf. on Princ. of Prog. Languages*, pp. 7-21, 1985.

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

[24] J.C. Reynolds. Types, abstraction, and parametric polymorphism. In *Proc. of Information Processing 83*, pp. 513-523, 1983.

[25] H. Shorr and W.M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10, pp. 501-506, 1967.

[26] R. Tarjan. Depth-first search and linear graph algorithms. *Siam J. Comp.*, 1, pp. 146-160, 1972.

[27] P. Wadler. Fixing some space leaks with a garbage collector. *Software Practice and Experience*, 17(9), pp. 595-608, 1987.

[28] P. Wadler. Theorems for free! In *Proc. of the ACM Conf. on Func. Progr. and Comp. Arch.*, pp. 347-359, 1989.

[29] P.R. Wilson. Uniprocessor garbage collection techniques. In *Proc. of Work. on Memory Management*, LNCS 637, pp. 1-42, 1992.