Types behind the mirror :
a proposal for partial ML type reconstruction at runtime

Emmanuel Chailloux   Pascal Manoury   Bruno Pagano
LIP6 & University of Paris VI, 4, place Jussieu, 75005 Paris, France
{Emmanuel.Chailloux,Pascal.Manoury,Bruno.Pagano}@litp.ibp.fr

May 11, 1997

Abstract

Polymorphic parametric languages, statically typed, *à la ML*, do not need to keep types inside their values during runtime to have a safe execution. Nevertheless this type information gives new possibilities for debugging, allows to extend the language to manipulate types as values. It can be stored either in stack or in heap.

To manage the additional type information, we propose in this paper to divide the heap in a value space and a mirror space. The first space is used to store values, the second one is used to store type information and garbage collecting. We describe a new Stop & Copy GC algorithm handling the divided heap with low memory cost. We finally give proposals for partially rebuilding types from informations stored in the mirror space.

1 Introduction

Polymorphic language implementations can be done without tags [2], then it is not immediate to know the type of an object. However it is possible to reconstruct (more or less) runtime types. There are two main applications needing that kind of information: memory management and program debugging. For the garbage collector (see the survey [7] on GC's), a tagless implementation can use a GC with ambiguous roots (i.e. without type information), but, recently several proposals have been made in order to rebuild types when needed. In [3] types are only partially rebuilt because objects can be hidden in a closure. This is not a problem because, as written in [3], "any object whose type cannot be reconstructed by the GC is garbage". In [1] and [6] types are completely rebuilt and are used in the context of a source debugger. Additya's scheme is a compiler-directed explicit tagging of objects though the extra tag information. The extra tags are passed as added arguments to polymorphic functions and propagated explicitly at runtime.

Type informations are not complete in the Goldberg's GC, but they are sufficient to run it. In the Additya's algorithm, extra informations must be added inside the stack, and must be evaluated in some cases. So, the types rebuilding has an important cost, and the program execution differs whether types are needed or not. It is not penalizing for a debugging tool, but this solution is not reasonable to use type information to extend an ML dialect.

For this purpose, we propose to add type information only inside the heap, in fact, a minimal information, which allows to rebuild type only when it is necessary. For non-functional values we can keep only their structural type, that we called in this paper the global types. For instance, in ML, the global type of an α *list* value is *list*. This is the Lisp notion of the *typeof* function. This global type is known during memory allocation, because the allocation process needs some information as size, and so on ... In ML, allocation is performed by constructors that know the global type. For functional values, the type constructor → is not sufficient. We need to keep type information computed at compile time. In order to be able to rebuild the real closure type from its proper information and the type informations contained in its environment.

Our proposal has a double interest. Firstly we do not use additional informations inside the stack. This scheme only changes allocations and GC during the program execution. Thus the compiler modifications are well localized. Secondly, we can (without unification) rebuild the actual type of every value (including closures) from the global and functional type informations only. We have to distinguish the type of immediate values in order to be complete.

To decrease the memory cost of adding type informations, we propose a new algorithm for a Stop & Copy GC. The main idea is to use the unused zone, that we call the mirror space, to keep the needed informations. This algorithm is slower than the classical implementations but its additional cost is low.

For each declared type, we generate an associated function, which can determine the type of parameter types. Having complete type informations at run time allows us to explore different applications as debugging, language extensions as the dynamics, and some optimizations for the GC.

The rest of this paper is organized as follows. Section 2 introduces different solutions to keep type informations inside heap and argue for using the unused space involved by Stop & Copy GC. We give then our new Stop & Copy algorithm and discuss its advantages. Section 3 explains how to rebuild type from the heap information. A short conclusion presents the future work.

2 Type tags inside heap

In this section we present different ways to keep type informations inside the heap. Our actual goal is to rebuild type values at runtime for statically typed language, but this GC algorithm can be used in other frameworks. The idea is that each allocated value has an extra information which allows us to rebuild its type. We present in the first subsection how to store type informations inside values and outside values in the heap. And we discuss the consequences for the classical GC algorithms Mark & Sweep and Stop & Copy. In the second subsection, we present a new Stop & Copy, which slices the heap in three zones : a value zone, a displacement zone and a type zone. This GC copies all alive cells and their type in place, respectively inside the value zone and the type zone. The third subsection contains the precise description of an optimized version of the algorithm which uses only two zones. The displacement zone and type zone are merged in only one, called the mirror zone. In the last subsection, we show several basic benchmarks and discuss some properties of the algorithm.

2.1 Storing type informations inside heap

Our goal is to associate a type with each allocated value. This type information must be directly accessible from the pointer of the allocated value. One first possibility is to extend the value with its type. The overhead cost memory is at least one word. For a Mark & Sweep GC, it is quite decent, but for a Stop & Copy GC it seems really expansive because this additional memory space must be also duplicated into the *to-space*.

Another way is to store the type information outside the associated value. To optimize the memory occupation we can use a separate part of the heap as an array of cells less than one word. To preserve relationship between values and their type, they are stored in an homothetic spaces. This becomes fine for the Mark & Sweep but it is always mandatory to double type space for a Stop & Copy.

Our key idea is to use a new Stop & Copy with only one type space using the homothetie between value space and type space. Therefore we call mirror space the type space.

In the two following subsections, we describe this algorithm. In a first step, we use three zones and present the main ideas of this algorithm. In the second step, we optimize the space occupation by merging type space and displacement space and we detail the algorithm in a pseudo-C language.

2.2 Stop & Copy in place

The Stop & Copy algorithm that we present requires a 64-bits alignment. All allocated objects will have an even number of 32-bits words. We have the same heap allocation for 32-bits and 64-bits machines.

Our algorithm permutes values inside the *value-space* using *displacement-space* and preserves informations into *mirror-space*. The type information of the content of each cell of the value space is in the mirror space at the same offset.

We represent a double cell allocation in 1 : The v_i's are values of cell, t_1 is the associated type information and the p_i's represent the cell displacement computed by GC. The word named WIG (Where I Go) is intended to contain the address of the future location of the cell. The one named WWC (Which Will Come) is intended to receive the address of the incoming cell.

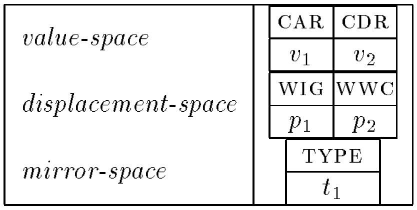

Figure 1: Value, displacement and mirror spaces

The algorithm is performed in four steps. The two first steps determinate what cells move and where they have to go. This information is kept inside the *displacement-space*. The third step modifies values of the living pointers according to information collected during the previous steps. These new addresses will be correct after displacement (step four). In the final step, cells and types are moved effectively. There are three cases : simple move into a free place, linear chained moves or cyclic moves.

We detail now the algorithm in its optimized version, which merges *mirror-space* and *displacement-space* in only one zone. Indeed, for our intended application (rebuilding types for staticaly typed language) we can encode both type information and displacement pointers inside the two words of a cell of the *mirror-space*.

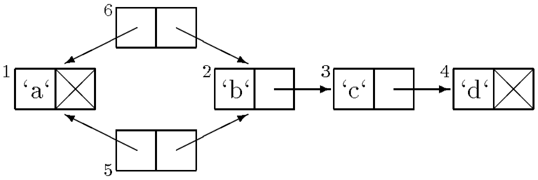

Figure 2: A stylized Caml-value

2.3 Types without type space

We now explain how we use the WIG and WWC fields and type information to perform the garbage collecting. The leading idea of the GC is to compute a permutation of cells from *value-space* to *value-space* and to save it in the WIG and WWC fields.

We present the algorithm in a pseudo-C language with its execution on a small example. Imagine a Caml object built by the code :

```
let l  =  [‘b‘;‘c‘;‘d‘] ;;
let a  =  match ( [‘a‘] , l )
          with  p  ->  ( fst p , snd p ) ;;
```

Firstly, this code allocates two lists in the heap, followed by a cell for pairing the two lists. After computations on the pair, a second pair is allocated to point on the lists. At last, the second pair is a root, but the first one is a dead cell. With a particular implementation of lists allowed by dynamic typing, the heap contains the structure shown on figure 2. The heap itself is represented in figure 3. It is worth to notice that *mirror-space* contains only types and values of p_i's are the null pointer. This must remain true after garbage collection has been completed.

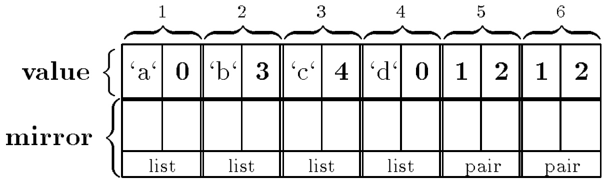

Figure 3: Heap before Garbage Collecting

We give the pseudo-code of each step and show the resulting heap. We use the following roots set `[3;6]`. Moreover, we assume there is somewhere (in the value or in the type) a bit for each word to distinguish between immediate values and pointers.

Let `VS` and `MS` be respectively the *value-space* and the *mirror-space*. We note `VS[wh]`, the wh-th cell, `CAR[wh]`, the CAR value of the wh-th cell, `CDR[wh]`, the CDR value of the wh-th cell, `WIG[wh]` is the p_1 pointer of the wh-th cell and `WWC[wh]` is the p_2 pointer of the wh-th cell.

The variable `free` denotes a pointer to the next free cell. As we assumed that each value is tagged to be an immediate value or a pointer, we have the functions:

```
int is_addr_car (addr);
int is_addr_cdr (addr);
```

The two first passes compute the permutation of cells. The future location of each cell is saved in the corresponding `WIG` field, and the address of the incoming cell is saved in the corresponding `WWC` field. This is performed by the function:

```
void Treat(addr a) {
  if ( WIG[a] == NULL) {
    WIG[a] := free;  WWC[free] := a;
    free++;
  }
}
```

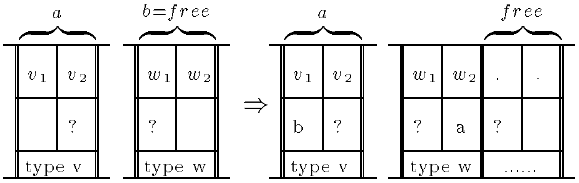

Figure 4: Calling Treat(a)

Pass 1) Processing the roots set (`RS`) :

```
free=1;
for (i=1;i<length(RS);i++) Treat(RS[i]);
```

Pass 2) Processing `WIG` and `WWC` for each moved cell, including those moved during pass 2 :

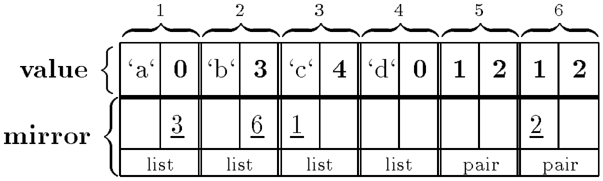

Figure 5: Heap after pass 1

```
for ( wh=1 ; wh < free ; wh++ ) {
  addr a = WWC[wh];
  if (is_addr_car(a))  Treat(CAR[a]);
  if (is_addr_cdr(a))  Treat(CDR[a]);
}
```

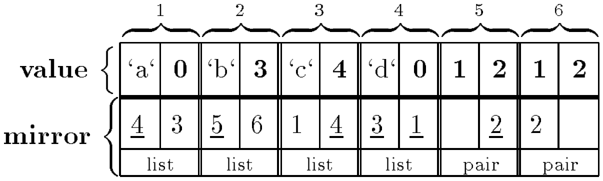

Figure 6: Heap after pass 2

Pass 3) Updating pointer values :

```
for (i=1;i<length(RS);i++) RS[i]=WIG[RS[i]];
for ( wh=1 ; wh < free ; wh++ ) {
  addr a = WWC[wh];
  if (is_addr_car(a))  CAR[a] = WIG[CAR[a]]
  if (is_addr_cdr(a))  CDR[a] = WIG[CDR[a]]
}
```

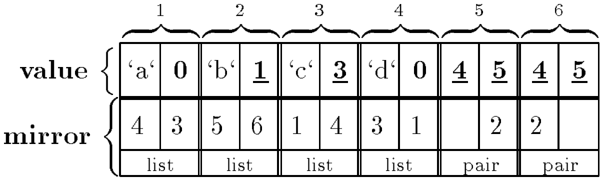

Figure 7: Heap after pass 3

Pass 4) It's time to really move cells: we need two auxiliary functions to move one cell or to move chained cells:

```
void move (addr src, addr dest) {
  VS[dest] = VS[src];
  MS[dest]->type = MS[src]->type;
  WWC[dest] = NULL;
  WIG[src] = NULL;
}

void move_chain (addr a) {
  while (a != NULL) {
    addr b = WWC[a];
    move(b,a);
    a = b;
  }
}
```

So the algorithm used to move the cells is:

```
  for ( wh=1 ; wh < free ; wh++ ) {
    addr a = WWC[wh];
    if (a != NULL)  {
      /* a stays in place */
      if (WIG[wh] == a ) {
         WIG[wh] = NULL;
         WWC[wh] = NULL;
      }
      else
        /* simple move */
        if (WIG[wh]==NULL) move(a,wh);
        else {
          addr b = WIG[a];
          while ( b!=a && WIG[b]!=NULL)
            b=WIG[b];
          /* linear chained move */
          if (WIG[b]==NULL)
            move_chain(WIG[b])
          /* circular move */
          else {
            move(a,buffer);
            move_chain(a);
            move(buffer,WIG[a]);
          }
        }
    }
  }
```

For the so called "circular move", we claim that our treatment is enough because the only possible cycles are the one ending with the first cell (namely `a` in our pseudo-code). If it were the case, a cell would have two destinations. And this can't happen. In our example, we detect a cycle during the treatment of the first cell: 1 ← 3 ← 4 ← 1, so the **`buffer`** variable is used to save the content of cell number 3, and the three cells are moved. After this movement, we treat cell number 2 which can move to the free cell 5. At the end cell 6 is moved to cell 2. The result corresponds to the figures 8 and 9.

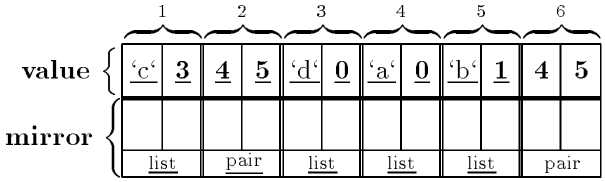

Figure 8: Heap after pass 4

In order to simplify the presentation we have assumed that there were only objects of size two. To handle the general case, we only have to iterate the two first passes on each joined cells of an object. We can tag joined cells reserving a bit of type information to indicate that the current cell is aggregated to the following one. In this way, when two cells are joined, we ensure that their new locations are decided successively and they remain contiguous. Remark that the moving pass is totally independent of the nature of the moved objects.

The two first passes don't change the data, and the third pass can be reversed at any time. Before the fourth pass has begun, it is possible to interrupt the garbage collecting process.

This algorithm does not allow references on middle

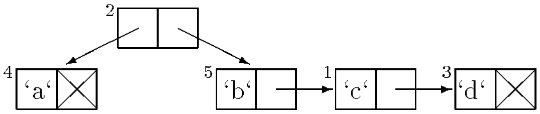

Figure 9: The alive Caml value

of data; to perform this, we just need to add this case in the two first passes.

2.4 Discussion

We made some basic evaluation tests on a toy implementation of mini-ML.

Figure 10 compares our algorithm (column "in place") with a classical Stop & Copy (column SC) on a program of list manipulation : determination of prime integers using the Erathostene algorithm. We run it on a Sparc 10. There are two versions of the algorithm (Era 1 and Era 2). The second version optimizes the roots set, so heap is smaller after GC.

Era 1			
heap size	GC nb	SC	in place
10000	830	18.5 s	24.8 s
100000	66	17.1 s	19.9 s
1000000	6	18.0 s	20.4 s
Era 2			
10000	80	9.4 s	9.7 s
100000	7	9.5 s	9.8 s
1000000	0	10.0 s	10.9 s

Figure 10: Small benchmarks

The time values represent complete computation including the GC part. The classical Stop & Copy does not handle types. Our GC is a little slower. The worst case is the first line in Fig 10. The small heap runs a lot of GC, and the price to pay is to move more surviving cells. In this case, the "in place" GC is about twice time slower than the classical Stop & Copy.

The growing memory space is reasonable. The stack size remains the same. The heap size grows only for structures needing an odd number of cells. If all structures need three cells, we loose 25% of the heap. This worst case is scarce because an ML program allocates much more value structures of different sizes than closures or tuples. Moreover this GC allows to optimize data representation inside the heap by using type informations. For example a floating point value (double) can be stored only on two words without tags inside its value. Another argument is the 64 bits alignment which is used in many implementations. In this last case there is no lost space.

It is possible to use this algorithm in a generational GC or a mixed GC (Mark & Sweep and Stop & Copy). The type space can be used to store the generation number inside cells that survive from GC.

3 Rebuilding type

We stated in the previous section how to manage type information at runtime with a low cost in time and space. Our aim now is to use this type information to dynamically rebuild type of values stored in the heap. We will discuss this point for non functional values and then for functional values. But before this, we need to put the emphasis on the difference between the statically infered type and the dynamically rebuilt type.

3.1 Statically infered type *vs* dynamically rebuilt type

We assume that each value is tagged with its structural type information. Then, while static type inference assigns the type `int list` to the expression `(tl [1])`, the dynamical type reconstruction gives the more general type `'a list` because we cannot recover the type parameter's value of an empty list only tagged with its structural type.

The second case where the rebuilt type is more general than the infered type is the following.

```
let mk_pair x = (x,x);;
(mk_pair []);;
```

The function `mk_pair` has the static type `'a -> 'a*'a`. So `(mk_pair [])` is statically typed with `'a list * 'a list`, whereas the value `([],[])` is dynamically typed with `'a list * 'b list`. This is partly due to the lost of information about type parameters we noticed above but this also comes from the lost sharing between the two components of the pair.

We guess that the dynamically rebuilt type is equal or more general than the statically infered type, for non functional values as well as functional values. This difference might cause damage depending on the applications we have in mind.

3.2 Non functional allocated values

Assuming the structural type information can be recovered at runtime, we can define a generic `type_of` function to compute the type from it. For each declared type `t`, we define, at compile time, a specialized `build_type_t` function.
For instance, if in the source code we have

```
type ('a,'b) t1 = C1 of 'a | C2 of 'b * 'a;;
```

we define

```
let build_type_t1 x = match x with
 C1 u -> (type_of u, undef) type_t1
|C2(u,v) -> (type_of u, type_of v) type_t1;;
```

where `type_t1` is the structural name of the type `t1` and `undef` indicates an unused type parameter. Then the main function `type_of` dispatches the work to specialized ones according to the structural type information tagging the values.

Let's consider now the following

```
type 'a t = C of 'a * 'a;;
```

In the rebuilding type function, we could take into account the type of only one component of the pair `(u,v)` to write

```
let build_type_t x = match x with
 C(u,_) -> let a = type_of u
           in (a,a) type_t;;
```

Then `C([],[1])` will be typed `'a t` althought we have here all what is needed to recover the `int t` type. To be more precise in our type rebuilding, we have to unify the two rebuilt types of the arguments of the constructor and write instead :

```
let build_type_t x = match x with
 C(u,v) ->
   let a = unify (type_of u) (type_of v)
   in (a,a) type_t;;
```

Actually, because we want to dynamically rebuild types of statically typed objects, we don't need a whole unification mechanism : we know that neither clash nor occur check are needed.

For recursive types we have also to guarantee that this exploration does not loop infinitely. If we declare the `z` type and the value `x` as follows:

```
type 'a z = K of 'a z;;
let rec x = K x;;
```

we obtain a classical circular value. To avoid infinite loop we need to mark the cells visited by the algorithm.

3.3 Functional values

With only a structural type information, there is no way to identify the type of a closure more precisely than to say this is a `<fun>`. To be able to be more precise we need to have, at runtime, type infomations coming from compile time. Then we can use the static information and the type information extracted from the environment together in order to rebuild the type of functionnal values occurring at runtime. Let's consider the following declarations :

```
let p = function x -> function y -> (y x, x);;
let f = (p 3);;
```

To the both abstractions `function x` and `function y`, we associate respectively the type infomation : $\alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow (\beta * \alpha)$ and $\sigma_x, (\sigma_x \rightarrow \beta) \rightarrow (\beta * \sigma_x)$. For the type reconstruction mechanism, σ_x will be a reference to the (rebuilt) type of x in the environment. We can then recover the type of the partial application defining `f` without having to propagate dynamically the type information when a function is applied. In our example, we get `'b -> 'b * int` as the dynamical type of `f`.

We have, at the moment, no precise idea about the implementation of the type reconstruction mechanism for closures. One possibility is to store the static type information in a table built at compile time and use the type information stored in the mirror space as an index to it. Then we could design a generic type reconstruction function using it. As it happened for non functionnal values, we may rebuild a dynamical type more general than the static one. And here again, we can improve the precision of the rebuilt type by adding unification constraints.

3.4 The weak type variables

Before concluding on our dynamical type rebuilding, we talk about the problem of weak type variables.

Up to now, we know that there exists some unavoidable imprecision of type rebuilding using structural type information. This is what we illustrate in section 3.1. Moreover, we do not obtain the same precision on dynamic type whether we perform unification or not. When we only consider the partial type rebuilt without unification, we do not have to handle with weak variables. Because, in this case types are always *totally generalized*, so weak variables or intance of it have the same treatment: a polymorphic value is generalized. Now, if we want to rebuild a more precise type without unification, weak variables may introduce some imprecision. To solve this problem, we can assume that the type rebuilding functions described in previous sections are defined after the typing pass of the compile time, that is after weak variables have an assignment. In that case, some trouble still remains when weak variables are unified together and this implies some restrictions to the use of the dynamic typing.

3.5 Possible applications

We claim that our proposal for dynamic type reconstruction allows to display and explore any value. Then we are able to design a more precise trace mechanism than what is usualy done. Moreover, when the rebuilt type of a value is monomorphic we can use this value as a persistent object or dynamic.

When using the two kinds of information extracted from the static typing (type expressions for all the abstractions and unification constraints) and when there is no polymorphic mutable value, we conjecture that for all values we are always able to construct the same value with a static type equal to its dynamic type. For instance, the dynamic polymorphic type of the empty list `(tl [1])` is nothing but the static polymorphic type of the empty list `[]` and `(tl [1])=[]`. Then we can safely apply a closure to any value in accordance with the rebuilt type.

The last but still negative point is that we cannot allow to modify values in the heap when the dynamic type remains polymorphic.

4 Conclusion

We have described a new algorithm for Stop & Copy GC which allows to handle additional informations inside the heap in a separate area. To encode global type information, the memory cost is very low and the execution cost is depending on the number of surviving cells. Using this type information, we proposed a partial type reconstruction at runtime without type propagation. In fact we have a "call by need" partial type reconstruction. The only additional cost is the one of the GC. One can use the rebuilt type to design a trace mechanism and to allow the manipulation of closures including polymorphic values. When the rebuilt type is monomorphic one can also physically modify them (in accordance with their rebuilt type) or use them as dynamics.

Our future research is to formalize the dynamic typing system in order to establish the well foundness of the intended applications. Then we could turn to a precise design of a dynamic type reconstruction algorithm joined with our GC.

References

[1] S. Aditya and A. Caro. Compiler-directed Type Reconstruction for Polymorphic Languages In *Functional Programming and Computer Architecture*, pages 74-82, June, 1993 Copenhagen.

[2] A. W. Appel. Runtime tags aren't necessary *Lisp and Symbolic Computation*, pages 153-163, 1989.

[3] B. Goldberg and M. Gloger Polymorphic Type Reconstruction for Garbage Collection without Tags In *1992 Conference on Lisp and Functional*

Programming, pages 53-65, June 1992, San Francisco.

[4] X. Leroy. The Caml Light system, release 0.7 - Documentation and user's manual INRIA, 1995.

[5] Xavier Leroy and Michel Mauny, Dynamics in ML In *Journal of Functional Programming*, 1993, volume 3, number 4, pages 431–463

[6] A. P. Tolmach Debugging Standard ML, Phd thesis, Princeton University, October, 1992

[7] Paul R. Wilson Uniprocessor Garbage Collection Techniques In *Proceedings of the International Workshop on Memory Management St Malo, France*, pages 1-42 Springer-Verlag, September 1992, LNCS 637.

[8] A. K. Wright Polymorphism for imperative languages without imperative types Technical Report TR93-200, Rice University Dept. of Computer Science, Feb, 1993.