Tag-free Garbage Collection Using Explicit Type Parameters

Andrew Tolmach
Portland State University
apt@cs.pdx.edu

Abstract

We have constructed a practical tag-free garbage collector based on explicit type parameterization of polymorphic functions, for a dialect of ML. The collector relies on type information derived from an explicitly-typed 2nd-order representation of the program, generated by the compiler as a byproduct of ordinary Hindley-Milner type inference. Runtime type manipulations are performed lazily to minimize execution overhead. We present details of our implementation approach, and preliminary performance measurements suggesting that the overhead of passing type information explicitly can be made acceptably small.

1 Introduction

Parametric polymorphic functions, as found in languages such as ML and Haskell, are traditionally compiled into code that executes uniformly regardless of the types of its arguments. This approach requires adopting a uniform data representation for all types. Typically, one pretends that every value fits in a single machine word; values that do not fit must be pointed to indirectly ("boxed"). The executing code need not be able to distinguish pointers from non-pointers, nor know the format of what is pointed to.

However, the garbage collector does need this information, in order to traverse the live data graph during a collection. Therefore, implementations of ML and similar languages typically use extra data fields to store the lengths of records and to distinguish pointers from integers; this technique derives from earlier Lisp implementations in which type tags were already necessary to support runtime type determination. Doing away with these extra fields could considerably decrease a program's space requirements, and save allocation and reformatting time. Moreover, it would allow use of more "natural" data representations that are closer to the format used by the machine, by other languages such as C, and by externally-specified interfaces such as communication protocols.

"Tag-free" collection is possible in principle if the source language has strong static typing [3]. Thus, an ML compiler has information about the type of every computed value, which can be passed to the garbage collector to guide live-data traversal. This suggests a "two-finger" approach to collection: as it scans, the collector keeps one finger on the data and the other on the corresponding entry in a type map provided by the compiler. No headers or tag bits are needed.

The stumbling block is that compilers usually treat generalized types in polymorphic functions as *abstract*. Appel [3], and subsequently Goldberg [7, 8], have observed that a garbage collector could reconstruct the types of abstract parameters by using the call stack to inspect the types of the actual arguments. Because this technique requires a form of runtime type unification, and a recursion bounded by the depth of the dynamic call chain, it does not appear particularly attractive in practice, although it has recently been used to build a garbage collector for Id [2]. Type reconstruction mechanisms built on similar principles have been implemented in debuggers for Standard ML [18] and Id [1].

One significant disadvantage of stack-based reconstruction is that it doesn't always work: variables in closures may outlive the stack context information that would allow their types to be reconstructed. Although Goldberg [8] has shown that such variables must always contain garbage and so can be safely ignored by a collector, practical considerations involving sharing of data structures make precise type determination very desirable. Aditya [1, 2] uses an approach that involves storing explicit type "hints" in closures when stack-based reconstruction would fail.

A more straightforward approach to providing type information at runtime is to augment *all* polymorphic functions with explicit type parameters, which are instantiated at each mention of the function. This idea has a long "folklore" history, dating back at least to the mid-1970's. Morrison, et al., give a rather informal description of how this mechanism is used to support specialized data representations in the implementation of Napier88 [14]. Very similar techniques have recently been applied in the implementation of Haskell type classes [19, 5, 15, 11].

We suggest that the clearest way to produce an explicitly-parameterized program is to view it as a translation of the original program into an explicitly-typed 2nd-order λ-calculus [9]. Once suitable type representations are chosen, the 2nd-order program can in turn be viewed as an ordinary (1st-order) source program, and compiled and executed directly using the standard compiler and runtime system. Moreover, the compiler already does all the work required to generate this augmented program as a byproduct of ordinary Hindley-Milner type inference. We are not the first to take this approach. For example, the Glasgow Haskell compiler now uses an explicit 2nd-order representation to facilitate type class operations [16]. A more elaborate formal framework also based on this approach has recently been proposed by Harper and Morrisett [10].

Although this idea is not new, to our knowledge it has never been used to build a working garbage collector. There are two main reasons for this lack of implementation experience. First, it has been feared that passing type information explicitly would cause unacceptable runtime overheads. Second, existing compilers are generally not set up to perform the appropriate type manipulations in the front end, nor to pass type information from the front end to the runtime system. This paper attempts to fill the gap by describing a practical implementation of explicit type parameterization as part of a tag-free collector for a dialect of ML.

A key feature of our implementation is that it manipulates runtime type descriptions in a lazy manner. Under this approach, the executing program can keep track of the correct instantiation context for its polymorphic variables using only a single dynamic list pointer.

Although this paper is devoted to garbage collection, the machinery it describes could easily be applied to other "type-conscious" applications such as overloaded printing, debugging, and non-uniform data representation.

2 Explicit Parameterization

We illustrate the principles of explicit parameterization using a simple subset of ML expressions

$$\begin{array}{rl} e ::= & c \mid x \mid e_1 e_2 \mid \lambda x.e_1 \mid \\ & \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \mid \\ & \texttt{letrec } f \texttt{ = } \lambda x.e_1 \texttt{ in } e_2 \mid \\ & \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \mid \\ & e_1 \texttt{ ; } e_2 \end{array}$$

Here x represents a variable and c represents one of a collection of built-in constants including at least unit, integers, floats, booleans, and the usual constructors and destructors for pairs and lists.

We define types as

$$\begin{array}{rl} \tau ::= & \alpha \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \times \tau_2 \mid \tau \texttt{ list} \mid \\ & \texttt{unit} \mid \texttt{int} \mid \texttt{float} \mid \texttt{bool} \mid \ldots \end{array}$$

where α is a type variable. We define type schemes as

$$\sigma ::= \forall\alpha_1 \ldots \alpha_n.\tau$$

Figure 1 gives typing rules for the expression language, which are fairly standard. Type environments TE map (ordinary) variables to type schemes. A type scheme is produced by generalizing a type with respect to some of its type variables. We follow Tofte [17] in folding type generalization into the (LET) rule, and generalizing on exactly those variables of the type that are not free in the type environment; more precisely, we define $\mathrm{Clos}_{TE}(\tau)$ as $\forall\alpha_1.\ldots\forall\alpha_n.\tau$, where $\{\alpha_1,\ldots,\alpha_n\} = \{\alpha \in \mathrm{tyvars}(\tau) | \alpha \notin \mathrm{tyvars}(TE)\}$. Similarly, we fold type instantiation into the (VAR) rule. A type τ' is an instance of $\sigma = \forall\alpha_1.\ldots\forall\alpha_n.\tau$, written $\sigma > \tau'$, if there exists a substitution S with domain $\{\alpha_1,\ldots,\alpha_n\}$ such that $S(\tau) = \tau'$.

For reasons discussed more fully in the next section, we follow Wright [20] in requiring that the defining expression of a generic `let` clause be a syntactic *Value*, i.e., a variable, constant, or λ-expression. As usual, recursive definitions must be explicit functions, and recursive values cannot be used polymorphically within their own definition. `let`-bound functions that bind type parameters are termed *generic*. We term each instantiation point of a variable bound to a generic function a *point of mention* for that function.

Programs in our explicitly-typed 2nd-order language are derived from source language programs by explicitly abstracting over the type variables in each generalizing `let` and `letrec` (using the binding operator Λ), and explicitly applying each generic function to instance types at its point of mention (using curly braces {} to surround the type arguments). To make the role of each type variable clear, we also attach a type annotation to each variable binding. The resulting syntax is shown in Figure 2. The transformed program can be obtained as a byproduct of running standard Hindley-Milner type-inferencing process on the original program. Each application of a (LET_1) or (LETREC) rule results in a type abstraction, whose parameters are exactly the bound types in the type scheme generated using the Clos rule. Each application of a (CONST) or (VAR) rule results in a type application, whose instance parameters are exactly the types in the range of the substitution that witnesses the $>$ relation. We omit abstraction and application when the set of generalized variables is empty. Mitchell and Harper [9] give a formal framework and proof of equivalence between original and 2nd-order programs for a similar language.

Figure 3 shows a simple polymorphic program and its 2nd-order translation.

3 Runtime Type Parameters

The utility of the 2nd-order representation for theoretical studies and compile-time optimizations such as representation analysis [12] is well-established. Our purpose in this paper is to *execute* 2nd-order programs, but using the standard 1st-order compiler. The key idea is to treat type parameters and values as (more or less) *normal* parameters and values. In particular, we define a runtime format for type descriptions, and generate code that builds and passes these descriptions at runtime just like normal parameters. For convenience, we

$$\frac{\text{TypeOf}(c) > \tau}{TE \ \triangleright c : \tau} \tag{CONST}$$

$$\frac{TE(x) > \tau}{TE \ \triangleright x : \tau} \tag{VAR}$$

$$\frac{TE \pm \{x \mapsto \tau'\} \ \triangleright e : \tau}{TE \ \triangleright \lambda x.e : \tau' \to \tau} \tag{ABS}$$

$$\frac{TE \ \triangleright e_1 : \tau' \to \tau \quad TE \ \triangleright e_2 : \tau'}{TE \ \triangleright e_1 e_2 : \tau} \tag{APP}$$

$$\frac{TE \ \triangleright e_1 : \tau' \quad TE \pm \{x \mapsto \text{Clos}_{TE}(\tau')\} \ \triangleright e_2 : \tau \quad e_1 \in \mathit{Values}}{TE \ \triangleright \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 : \tau} \tag{LET$_1$}$$

$$\frac{TE \ \triangleright e_1 : \tau' \quad TE \pm \{x \mapsto \tau'\} \ \triangleright e_2 : \tau \quad e_1 \notin \mathit{Values}}{TE \ \triangleright \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 : \tau} \tag{LET$_2$}$$

$$\frac{TE \pm \{f \mapsto \tau'' \to \tau', x \mapsto \tau''\} \ \triangleright e_1 : \tau' \quad TE \pm \{f \mapsto \text{Clos}_{TE}(\tau'' \to \tau')\} \ \triangleright e_2 : \tau}{TE \ \triangleright \texttt{letrec } f \texttt{ = } \lambda x.e_1 \texttt{ in } e_2 : \tau} \tag{LETREC}$$

$$\frac{TE \ \triangleright e_1 : \texttt{bool} \quad TE \ \triangleright e_2 : \tau \quad TE \ \triangleright e_3 : \tau}{TE \ \triangleright \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \tau} \tag{IF}$$

$$\frac{TE \ \triangleright e_1 : \tau' \quad TE \ \triangleright e_2 : \tau}{TE \ \triangleright e_1 \texttt{ ; } e_2 : \tau} \tag{SEQ}$$

Figure 1: Typing rules

$$\begin{array}{rl} e ::= & c\{\tau_1, \ldots, \tau_n\} \mid x\{\tau_1, \ldots, \tau_n\} \mid \\ & e_1 e_2 \mid \lambda x : \tau.e_1 \mid \\ & \texttt{let } x : \forall \alpha_1 \ldots \alpha_n.\tau \texttt{ = } \Lambda(\alpha_1, \ldots, \alpha_n).e_1 \texttt{ in } e_2 \mid \\ & \texttt{letrec } f : \forall \alpha_1 \ldots \alpha_n.\tau' \to \tau \texttt{ = } \Lambda(\alpha_1, \ldots, \alpha_n).\lambda x : \tau'.e_1 \texttt{ in } e_2 \mid \\ & \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \mid \\ & e_1 \texttt{ ; } e_2 \end{array}$$

Figure 2: Explicitly-typed 2nd-order language.

```
let f = λx.(x,x)
in let g = λ(y,z).(f y, f z)
   in g (3,2.0);
      g (1.0,0)

let f:∀α.α → (α × α) = Λα.λx:α.(x,x)
in let g:∀β,γ.(β × γ) → ((β × β) × (γ × γ)) = Λ(β,γ).λ(y:β,z:γ).(f {β} y, f {γ} z)
   in g {int,float} (3,2.0);
      g {float,int} (1.0,0)
```

Figure 3: A simple polymorphic program and its 2nd-order translation.

tuple together all the type parameters at a given generalization and call the resulting vector a *type environment.* To interpret the 2nd-order program in this way, it suffices to read Λ as λ, the type variables α,β,γ,... as ordinary variables, and the type application curly braces {} as a special sort of data constructor that produces a runtime representation of a type environment from a list of type expressions. We discuss concrete representations in Section 4.

Each generic function now takes an additional curried initial argument which is a (formal) type environment; each mention of a generic function is replaced by an application of that function to an (actual) type environment. The actual types of the parameters thus become available for either explicit or implicit use by the code generated for the body. Explicit uses of type descriptions might include a true polymorphic printing or `eval` function. Garbage collection makes implicit use of these descriptions: the garbage collector may need to know the value of any type parameter that appears in the type of any heap-allocated variable or temporary.

There is one major difficulty with treating Λ as an ordinary abstraction: evaluation cannot proceed under it. Thus, if the body of the generic `let` binding performed any computation, this computation would be repeated each time the function name was mentioned. This is certainly inefficient; more seriously, it is unsound if evaluation of the body causes side-effects. It is for this reason that we adopt Wright's value restriction [20].

The 2nd-order program appears to require many more function calls than the original, since each mention of a generic function now is replaced by a call. Fortunately, whenever the generic function's point-of-mention is an application, it is possible to combine the type instantiation and the call; we *uncurry* the function, so that the type environment parameter becomes simply another argument, which can be passed relatively inexpensively. In fact, many compilers already perform a suitable uncurrying transformation on ordinary user code. We cannot always avoid a curried call, however. Consider the following code:

```
let f:int → (int × int) = λx:int.(x+1,x+1)
in let g:∀α.α → (α × α) = Λα.λy:α.(y,y)
   in (if ... then f else g {int}) 3
```

We cannot avoid a separate partial call to `g`, since, although it is statically clear that either function applied to 3 takes an integer argument, only `g` expects a type environment parameter. Fortunately, such situations appear rare in practice. Partial calls may actually be desirable in some circumstances; see Section 7.1.

Type expressions passed to generic functions often themselves contain type variables, as illustrated in Figure 3. In particular, type expressions of this kind may arise in programs of the form `let f = e₁ in let g = e₂ in e₃`, where both `f` and `g` are generic. This pattern is common in ML, because any sequence of "top-level" declarations (in the interactive system or within a module) is type-checked as if it were syntactic sugar for nested `let` expression of this form. In particular, library functions are defined as if they were `let`-bound in the scope of ordinary user code.

Under the translation scheme of the previous section, the type variables referenced in a function body are not necessarily immediate members of that function's type environment parameter. Consider the code at the top of Figure 4, which illustrates a different, rather less common, way of nesting generic `let` expressions. Note that the type of `x`, namely α, is not present in the type environment passed to `g`, since `g` is not polymorphic in α; in effect α is free in `g`. However, if we perform λ-lifting or equivalent closure conversion, it is easy to arrange that every type variable mentioned in a function *is* included in that function's vector of formal type parameters, as illustrated in the bottom of Figure 4.

It is illuminating to consider the 2nd-order representation of types that cannot be reconstructed solely by stack walking. Consider this example:

```
let f:∀α.α → int → int =
        Λα.λx:α.λy:int.(x; y+1)
in let g:int→int = f {bool} true
   in g 4
```

At the point where the inner λ-expression in `f` is called, the fact that `x` was bound to a `boolean` is no longer deducible from information on the call stack, i.e., the value of α has been lost.[1] The 2nd-order representation makes it evident that α is itself a free variable of the inner nested abstraction (λ`y`), and must be saved in its closure, just like `x`.

4 Representing Types and Type Environments

To compile our 2nd-order language using an ordinary 1st-order compiler, we must choose concrete runtime representations for types and describe when and how polymorphic types are instantiated. These representations are ultimately used by the application that requires runtime type information—in our case, the garbage collector; they also represent types in the type-environment vector passed to generic functions.

Different applications require different levels of type information. Debugging and overloaded printing, for example, require complete source-level type descriptions. For garbage collection, we need only a simplified digest of type information for heap-allocated data, including the size of each allocated record and the locations and (digested) types of any pointers within that record. This section describes a suitable representation scheme for a language whose types include integers, floats, function types, products, and sums (discriminated unions) of products. For simplicity, we neglect the remaining essential constructor, namely arrays.

Figure 5 shows a concrete ML-style definitions for *runtime type descriptions* (`rttds`) and *type environments* (`tenvs`). The representation of monomorphic type expressions is straightforward. Integers, floats, and pointers to non-heap memory (such as code pointers) are represented by `Integer`, `Float` and `Static_pointer` respectively. Type expressions corresponding to heap-allocated records have the form `Record(x)`, where `x` is

[1] Of course, x could be treated as garbage without disturbing this computation, but in more elaborate examples the desirability of obtaining types for all variables becomes clear [8].

```
let f:∀α.α → ((α × int) × (α × bool) = Λα.λx:α.let g:∀β.β → (α × β) = Λβ.λy:β.(x,y)
                                                 in (g {int} 2, g {bool} true)
in f {int} 3; f {real} 3.14

let g':∀β,α.β → α → (α × β) = Λ(β,α).λy:β.λx:α.(x,y)
in let f:∀α.α → ((α × int) × (α × bool)) =  Λα.λx:α.(g' {int,α} 2 x, g' {bool,α} true x)
   in f {int} 3; f {real} 3.14
```

Figure 4: Code involving free type variables, before and after λ-lifting.

```
datatype rttd =
  Integer                (* simple integer *)
| Float                  (* simple float; only legal within product list *)
| Static_pointer         (* pointer to non-gc'ed memory (e.g., code) *)
| Closure_pointer        (* pointer to self-describing closure *)
| Record of rtrd         (* pointer to heap record *)
| Type_var of int        (* index into type environment *)

and rtrd =
  Sum of rtrd vector     (* list of variant types; indexed by header tag *)
| Product of rttd vector (* list of record fields *)

datatype type_env =
  Tenv of {instance:rttd vector, mentioner:type_env}
| Empty_tenv
```

Figure 5: Contents of runtime type descriptions.

a *runtime record description* (rtrd), which details the contents of the record. A simple product record is described by listing the `rttd`s of its fields, which may include nested `Record`s. For sum types, it is necessary to describe the layout of each possible variant; the actual variant at hand is determined by consulting a tag in the record. Note that this mechanism can describe recursive types. Finally, function closure records are self-describing (see Section 5.2), so they receive a special `Closure_pointer` descriptor.

The key questions are how to represent type expressions that contain type variables, and how to instantiate such expressions at runtime, after the actual values of the type variables have become available. Polymorphic type expressions arise only inside generic functions, and their type variables become available when the function is passed its type environment vector. In principle, therefore, every type expression within the function body could be instantiated as soon as the type environment is seen. Perhaps the simplest approach to instantiation would be to have the compiler generate code to construct, at runtime, a completely fresh description of the instantiated type, using the polymorphic type as a template and filling in the actual type parameters in place of the type variables. Under this approach, there would actually be no need to represent polymorphic types at runtime, and the `rttd` fields already described would suffice.

But generating such instantiated descriptions at runtime would quite time-consuming; worse, these descriptions would have to be heap-allocated, costing additional space and time. Moreover, only the garbage collector needs to examine the entirety of a type description, and it only needs to do so for the types of variables that happen to be live at a collection point. Thus, much of this instantiation work would be completely wasted.

We therefore choose instead to implement a *lazy* form of instantiation. The key idea is to represent polymorphic types in *parametric* form and to interpret them relative to an *instantiation vector* of types. Concretely, we introduce an `rttd` constructor of the form `Type_var(i)`, where `i` is an integer index into an auxiliary vector `inst` of `rttd`s. To interpret a type description containing the `rttd` `Type_var(i)`, one simply fetches `inst[i]`. Instantiating a description now amounts simply to forming an association between an `rttd` and an appropriate instantiation vector. Of course, whenever we transmit an `rttd` containing type variables, e.g., to the garbage collector or as a type parameter to a generic function, we must take care to transmit the appropriate instantiation vector as well.

Appropriate instantiation vectors are already at hand. Recall that after the λ-lifting transform described in Section 3, each type variable mentioned in a function appears in that function's formal type-environment vector parameter. Thus we can easily assign the type variable a runtime description `Type_var(i)` where `i` is the variable's index in the current environment.

Figure 5 shows a concrete representation for type environments, called `tenv`s. There is a `tenv` corresponding to each mention of a generic function; it is constructed by the function that contains the point-of-mention (the *mentioner* of the generic function) and passed as an explicit argument to the mentioned function.[2] A `tenv` contains a `instance` vector of `rttd`s corresponding to the type variables referenced by the function, as described above.

An `rttd` listed in a `tenv instance` may be a `Record` description or any simple data type that occupies one word, i.e., `Integer` or `Static_pointer`. Moreover, it may itself be a type parameter of the form `Type_var(j)`. In this case, the parameter index `j` must be interpreted in the type environment of the mentioner. The simplest way to make this environment available to the generic function is for the the mentioner to incorporate a pointer to it into the `tenv`; the `mentioner` field contains exactly this pointer. Naturally, this environment may itself contain type variables, necessitating the consultation of a further type environment, and so forth. However, this recursion is limited by the static depth of nesting of generic functions, which in practice is surely quite small.

Figure 6 shows code from an earlier example re-expressed using these representations. Type annotations of variables show the `rttd` that would be passed to the garbage collector for that variable.

It is tempting to believe that the `mentioner` field is dispensable. Often, the mentioner and caller of a function will be one and the same, in which case the mentioner's `tenv` could, in principle, be obtained via the call stack. In general, however, the mentioner may no longer exist at the point of call, and no stack-based mechanism can connect mentioner and callee appropriately.

This representation of `rttd`s and `tenv`s was designed to minimize the amount of dynamic allocation required to support runtime type resolution. Note that `rttd`s and the `instance` vectors of `tenv`s are completely static. In fact, only the `tenv` records themselves, i.e., the *associations* of `instance` vector with mentioner's `tenv`, ever need to be allocated dynamically. Of course, care must be taken to scan `tenv`s during garbage collection![3]

Harper and Morrisett [10] give an abstract treatment of a system similar to ours, intended to support a wide range of applications. They characterize type instantiation as normalization in simply-typed λ-calculus of type expressions. They point out that such normal forms can be found using a variety of strategies, including call-by-need or call-by-value. Viewed in this framework, our "lazy" approach appears to be an implementation of call-by-need, while the "eager" instantiation approach we initially considered but rejected is call-by-value.

There are many other possible representations for type environments, which we have not fully explored. One alternative approach is based on the observation that a polymorphic program can be "unfolded" into a monomorphic one by making a separate copy of each generic function each time it is mentioned. If we generated, type-checked, and executed the unfolded program, each function code address would uniquely determine a (monomorphic) type for all the function's variables. The resulting environment descriptions could be stored in a table indexed by function code address. Of course, actually unfolding the program would probably increase code size unacceptably (although recent results of Jones [11] suggest otherwise). Moreover, doing so might be impractical in a separate compilation environment, where the "whole program" is never seen at once. But we could generate code in the ordinary, folded-up program to keep track of which "copy" of a function we *would* be executing, were we running the unfolded program [18]. This tracking could use essentially the same dynamic list technique as the one described above. It could also be done using nested lookup tables, or perhaps a single table with some form of hashing. Moreover, we can certainly generate an appropriate table containing fully instantiated type environments for each function and "copy." Conveniently, the type information can be digested to an appropriate form for different applications (e.g., full user-level descriptions for debugging, record layout information for garbage collection, etc.) without changing the copy-tracking code.

Some "two-finger" garbage collectors have been implemented by passing type-specific traversal functions rather than type descriptions that must be interpreted by a universal collector function [8]. Such approaches make elegant use of first-class functions, but we suspect that the costs of applying the general closure construction mechanism for these functions will outweigh the benefits of avoiding interpretation.

5 Implementation

5.1 Gallium

These ideas have been implemented by extending Xavier Leroy's Gallium compiler [12] for the Caml Light dialect of ML [13]. The compiler has two main parts: the front end generates an intermediate language called "C– –"; the back end generates native code for a MIPS processor, using a direct-style, stack-based compilation model. The front end was originally designed to support representation analysis based on compile-time type information. Gallium was chosen for this experiment primarily because C– – already annotates identifiers and temporaries with crude type descriptions, namely integer, float, and pointer (to anything). The back end uses this information to type registers and stack frame locations.

Ordinarily, Gallium uses a simple, non-generational, depth-first copying collector. Records are self-describing: at allocation, each record is given a header that points to a static record layout description, which is generated from the record constructor definition. The layout description classifies each record field as integer, float, or pointer; it also contains a tag to distinguish variants of concrete types. Values in polymorphic fields are always boxed, even if they are integers; the corresponding field description is thus always "pointer." This technique obviates the need for tag bits to distinguish integers from pointers, at the cost of storing integers inefficiently. Live roots are passed to the garbage collector via frame descriptors, which are embedded in the code stream at each function call site, and can be accessed by the collector via the function's return address.

[2] In some cases, the mentioner can reuse an existing tenv rather than constructing a new one; see Section 5.3.

[3] The rttd for a tenv is a constant known to the collector; it is similar to that of an integer list.

```
let f = λftenv.λx:Type_var(0).(x,x)
in let g = λgtenv.λ(y:Type_var(0),z:Type_var(1)).
           (f Tenv{instance=[|Type_var(0)|],mentioner=gtenv} y,
            f Tenv{instance=[|Type_var(1)|],mentioner=gtenv} z)
   in g Tenv{instance=[|Integer,Float|],mentioner=Empty_tenv} (3,2.0);
      g Tenv{instance=[|Float,Integer|],mentioner=Empty_tenv} (1.0,0)
```

Figure 6: Representing explicit type parameters using the example of Figure 3. The symbols `[|` and `|]` construct a vector from the expressions listed between them.

Each frame descriptor consists of a list of pointers (in registers or the stack frame) that are live at the given function invocation. To trace all local roots, the garbage collector must walk the call stack, tracing all pointers in each active frame descriptor. There is also a static list of global roots. The collector is coded in C.

5.2 Adding Type Descriptions

For these experiments, we made significant changes to the garbage collector's method for tracing data. Layout information is no longer obtained from record headers; indeed, record headers are no longer present except when necessary to distinguish variants in sum types.[4] Instead, the frame descriptors are expanded to include a `tenv` for the function and an `rttd` for each live register or frame slot that might contain a pointer. The collector traverses type information in parallel with data.

The C representation of an `rttd` fits in a single word, as follows: `Type_var(i)` is represented simply by `i`, which is certain to be small (say < `0x100`); `Record(rtrd)` is represented by the address of the `rtrd`, which is certain to be large (say >= `0x1000`); the remaining constructors are represented by constants in the intervening range. An `rtrd` is represented as a C union in a straightforward way.

Each `rttd` is statically allocated; if it corresponds to a polymorphic variable, it will contain one or more `Type_var`s that reference the function's `tenv`. The back end arranges to store the current `tenv` at a fixed offset in the stack frame before each procedure call or invocation of the allocator. The C representation of a `tenv` is just a two-word pair; each field of the pair points to a statically generated record, so the pairs themselves are the only components in the type description scheme that are dynamically allocated.

Closures introduce considerable complications. Even in monomorphic type system, it is impossible to determine the number and type of a function's free variables from the function's type, so closure records must be self-describing. Gallium uses simple flat closures; the function's code address is always in the first field.[5] To avoid a header field within the closure itself, we embed an `rtrd` describing the closure record in the code stream at a fixed offset before the function's starting address.

Polymorphism opens the possibility that a closure's `rtrd` contains type variable fields. If so, the closure record must contain the `tenv` current when the closure is built and its `rtrd` generated. In some cases this `tenv` will already be present in the closure as a free variable in its own right; in other cases, it must be forced into the closure explicitly. In either case, the offset of the `tenv` within the closure record varies from one closure to another, so this value is also embedded in the code stream at a (second) fixed offset before the function's starting address. Note that the closure may also contain other, different, `tenv` values needed as free variables, e.g., to set the current `tenv` value within the function itself. Indeed, a given free variable value in the closure may be traced at different times by the garbage collector using two completely different descriptions and `tenv`s.

5.3 Compiler Changes

The original C– – attaches crude type descriptions (integer, float, or pointer) to the results of loads, function arguments, and function return values. We refine the description of pointer-valued data by including the address of an `rttd` describing what is pointed *to*. The back end now picks up this address for use in frame descriptors; it required no other significant changes.

The front end required more substantial revisions. Type annotations for generic functions and their mentions are extracted during the type-checking phase and stored in the abstract syntax tree. A newly-added processing phase inserts formal and actual parameters representing type environments and code for building `tenv` pairs where needed. The compiler avoids constructing a new `tenv` when the environment of the mentioned and mentioning function are the same. This is quite common because recursive functions always call themselves with the same `tenv` they were passed by the initial, non-recursive call. As another optimization, if the `mentioner` field is empty, the `tenv` can be statically allocated; this circumstance arises naturally when the mentioning function's `tenv` is empty, and can be forced artificially if the `instance` vector contains no type variable fields.

This phase precedes the existing uncurrying optimization phase, so that the latter can remove type instantiation calls where possible; no changes to the uncurrying phase were required except minor modifications to keep more precise track of type information in the uncurried code.

The closure construction phase has been modified to treat free type variables like ordinary free variables, and

[4] They are also used to store length information for arrays, which we do not discuss further.

[5] Mutually recursive functions share a closure, as described in [4, Section 10.2].

to install tenvs in closures where needed and not already present. The free type variables of a function are calculated as the type variables that appear in the ordinary free (value) variables of that function. Since a tenv may itself be a source of free variables, processing order within this function is delicate.

The final front end phase, which emits C– –, is enhanced to generate data segments containing static rtrd records and tenv instance vectors; the addresses of these segments are associated with pointer-typed values in C– –. A memoization mechanism is used to avoid generating multiple identical rtrds from a single source file; a similar mechanism at link time would clearly be desirable.

5.4 Garbage Collector Changes

The garbage collector is substantially revised to cope without record headers. Some records, e.g., members of variant types, still have header fields, but these are treated as ordinary data fields by the collector. The principal change is that type information is now taken from frame descriptors and, on recursive traversals, from rtrds, rather than from headers. The collector's inner copy loop is now parameterized by an rttd and tenv in addition to source address and contents.

We make fundamental use of the collector's depth-first traversal strategy here, since there is no convenient place to queue typing information that a breadth-first collector would require. Of course, we could use the information available at record creation time to store a pointer to a monomorphic type description within the record. This approach would effectively require us to reintroduce headers, though it would still permit us to avoid integer vs. pointer tags. A slightly more attractive possibility is to avoid attaching a header until the record is actually copied at collection time[6] but even this option seems likely to cut significantly into the space savings we hope to achieve.

Ordinarily, Gallium uses headers to store forwarding addresses during copying collection. Since not all bits are needed for tagging, it is easy to distinguish forwarding pointers from unforwarded headers. In the new scheme, forwarding pointers must be placed in data fields. Since forwarding pointers cannot, in general, be distinguished from integers or real numbers, they can only be placed in pointer fields. The first field guaranteed to be a pointer (if any) is used for forwarding; its offset is stored in each rtrd. Either heap or static pointer fields can be used, as forwarding pointers can be distinguished by a range check or by setting otherwise unused low-order bits in the pointer. For simplicity, abstract fields, which might or might not contain pointers depending on how they are instantiated, are not used. Records containing only numbers (e.g., pairs of integers) are forwarded by recording them in an auxiliary hash table and using the first field to store the forwarding pointer.[7] Other possible approaches include using a bit map in place of a hash table, or forcing all-numeric records to have an extra field for forwarding purposes.

[6] Greg Morrisett suggested this idea.

[7] John Reppy suggested this technique.

6 Assessment

6.1 Performance

We have only preliminary performance information. Table 1 shows the relative performance of the new system vs. ordinary Gallium on a number of synthetic benchmarks, intended to exercise the allocator and the type representation mechanism. Benchmarks were conducted on a Decstation 5000/240. sieve is a prime number generator. sumlist sums a list of integers and sumlistf sums a list of floats. pair repeatedly applies a function similar to repair in Figure 7 to integers and pairf applies it to floats; each call to pair builds a fresh dynamic closure. pclist constructs and evaluates a list of polymorphic closures. Ratios of run times (total cpu times) are presented for three different heap sizes selected independently for each benchmark: Small Heap is close to the smallest heap in which the benchmark would run, with frequent collections; Large Heap is large enough that no collections occurred; Medium Heap is a size somewhere in between with a "comfortable" number of collections.

The figures presented are encouraging. Total allocation decreased for all benchmarks but pairf, because the new system's occasional need to heap-allocate tenvs was more than offset by its use of unboxed integers and lack of need for headers. In fact, both ordinary Gallium and our new system use a very naive representation for sum types which requires each list cell to have a tag field; if this were fixed, the improvement in the new system would be even more marked for list-based programs like sumlist. The poor behavior of pairf could be addressed by the hoisting optimization described in Section 7.1. Moreover, execution times in the absence of garbage collection (Large Heap) also decreased (except for pairf) in line with decreased allocation, suggesting that passing type parameters need not be a significant cost.

Comparing performance when collections occurred is difficult because the percentage of live data at a collection can vary enormously depending on the precise point where the collection occurs. However, we can conclude that the new system behaves about as well as the old at moderate collection rates (Medium Heap), but that some benchmarks behave significantly worse under the new system when garbage collection is very frequent (Small Heap). We believe that this behavior is due in part to increased amounts of live data (holding tenv parameters and values), but also to poor engineering of the collector code, which can be improved. For example, nearly 25% of sumlistf's execution time was devoted to managing the hash table for forwarding floats, which suggests that other forwarding mechanisms should be investigated. We also expect the recursion optimization described in Section 7.1 to be useful.

6.2 Experience

Implementing support for explicit type parameterization for a full-featured language turned out to be somewhat more complex than we had expected, particularly in the front end. All told, at least 20% of the

```
let pair:∀α.α → (α × α) = Λα.λx:α.(x,x) in
    let repair:∀β.β → ((β × β) × (β × β)) = Λβ.λy.pair {(β × β)} (y,y) in
        let doit n = if n = 0 then () else repair {int} n; doit (n-1)
```

Figure 7: Worst-case example for dynamic `tenv` creation.

Benchmark	Ratios (New Collector/Ordinary Gallium)				
	Records Allocated	Bytes Allocated	Execution Time		
			Small Heap	Medium Heap	Large Heap
`sieve`	0.8	0.8	1.0	1.0	1.0
`sumlist`	0.5	0.6	0.4	0.6	0.6
`sumlistf`	1.0	0.8	2.6	1.0	0.9
`pair`	1.0	1.0	1.1	1.0	1.0
`pairf`	1.2	1.1	0.7	0.8	1.2
`pclist`	1.0	0.9	1.9	1.1	1.0

Table 1: Benchmark Results

front-end code (which totals about 10,000 lines of ML) required modification, and even more radical changes would probably improve the reliability of the code.

The collector itself was not too difficult to implement. One essential aid was was provision of a debugging mode, in which the allocator stores a fully instantiated type description for each record in a separate table. Whenever the collector scans a record it can compare the stored type with the parametric type it believes to be correct. A formal proof that the collector views record types correctly at all times would certainly strengthen our work.

7 Future Work

We will continue to explore our collector's performance on a range of more realistic programs. Since Gallium is a stack-based system, many programs will allocate much less rapidly than our benchmarks, so we expect performance differences in heap management will be less significant.

7.1 Implementation Optimizations

Analysis of our preliminary performance results suggests a number of useful optimizations that we have not yet implemented.

It is fairly inexpensive to pass `tenv`s as parameters or storing them in the stack frame, but constructing new `tenv`s dynamically is expensive because it requires heap allocation. We already avoid generating a new `tenv` in certain common cases, e.g., when the `tenv`s of mentioned and mentioning function are the same, but other optimizations are possible. For example, if the `tenv` of the mentioned function is a subset of the mentioning function, possibly permuted, the type variables of the mentioned function could be renumbered to refer directly to slots in the mentioning function's `tenv`. Of course, in general this cannot be done for all mentions of a function, but constructing a `tenv` can be avoided at one mention at least. This optimization is analogous with choosing argument registers for calls to known functions so as to avoid register moves [4, p. 159].

If a function's mentioner and caller are the same, as is usually the case, it is possible for the garbage collector to obtain the mentioner's `tenv` via the stack frame rather than from the called function's `tenv mentioner` field. The `mentioner` field could take on a special constant value to signal this situation, allowing the `tenv` to be built statically. The resulting scheme might resemble Aditya's use of "hints" [1, 2].

As noted in Section 5.3, the `tenv` passed to the first, non-recursive call of a recursive function is retransmitted to each recursive call of the function. If this `tenv` is heap-allocated, it would be desirable to avoid passing it repeatedly in this manner. This is because, in addition to the usual costs of passing and storing a parameter, each copy of the `tenv` must itself be scanned by the garbage collector, so it will appear as an extra local root in the function's frame. If a collection occurs during a deep recursion, the addition of a single extra root to each recursive frame can considerably increase its cost, even if the `tenv` structure itself is trivial to scan.

A solution would be to maintain the *current* `tenv` in a global location rather than in the stack frame. Ordinarily, the code to call a function would store the current `tenv` into the stack frame before the call and restore it after the return; the start-up code within the function would set a new current `tenv`. Calls to recursive functions (and any other calls statically known not to alter the `tenv`) would simply leave the current `tenv` unaltered, storing an appropriate mark into the stack frame to advise the garbage collector to continue using the current `tenv` when processing this frame.

Consider again the code in Figure 7, in which the recursive function `doit` repeatedly invokes the generic function `pair` indirectly via `repair`. The optimization just described won't prevent `repair` from building `n` identical dynamic `tenv`s for `pair`, one on each call. This problem could be solved by taking advantage of the fact that the explicitly typed form of `repair` is curried. We could arrange to construct `pair`'s `tenv` just once, by hoisting the code to generate it above `repair`'s inner abstraction, and hoisting the partial application of `repair` (to {`int`}) out of the recursion in `doit`. Evidently, this optimization would need to be applied in lieu of uncurrying.

7.2 Alternative GC Methods

For completeness, we could also compare performance of our collector with a reconstruction-based collector in the style of Goldberg [7, 8] or Aditya [2], possibly using specialized collector functions. It would also be interesting to try producing completely specialized versions of polymorphic functions, as Jones does for overloaded functions in Gofer [11]. Finally, we should compare our approach with that of conservative collection [6].

7.3 Other Applications

We also hope to extend our results beyond garbage collection. If manipulating explicit type information is sufficiently cheap, there are many other potential applications. These include providing better type information for debugging; providing an `eval`-like capability; use of more efficient and specialized data formats; and implementing more sophisticated forms of representation analysis, e.g., coding parametric polymorphic functions to behave differently according to the size of their arguments.

Acknowledgements

Xavier Leroy graciously provided access to the code for his Gallium system, and permission to modify it for these experiments. Robert Harper provided useful comments on an earlier version of this paper.

References

[1] S. Aditya and A. Caro. Compiler-directed type reconstruction for polymorphic languages. In *FPCA '93 Conference on Functional Programming Languages and Computer Architecture*, pages 74–82, June 1993.

[2] S. Aditya and C. H. Flood. Garbage collection for strongly-typed languages using run-time type reconstruction. In *Proc. 1994 ACM Conference on Lisp and Functional Programming*, June 1994.

[3] A. W. Appel. Runtime tags aren't necessary. *Lisp and Symbolic Computation*, 2:153–62, 1989.

[4] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[5] L. Augustsson. Implementing haskell overloading. In *FPCA '93 Conference on Functional Programming Languages and Computer Architecture*, pages 65–73, June 1993.

[6] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software—Practice and Experience*, 18(9):807–820, Sept. 1988.

[7] B. Goldberg. Tag-free garbage collection for strongly typed polymorphic languages. In *Proc ACM SIGPLAN '91 Conf. on Prog. Lang. Design and Implementation*, pages 165–176. ACM Press, 1991.

[8] B. Goldberg and M. Gloger. Polymorphic type reconstruction for garbage collection without tags. In *Proc. 1992 ACM Converence on Lisp and Functional Programming*, pages 53–65, June 1992.

[9] R. Harper and J. C. Mitchell. On the type structure of standard ml. *ACM Trans. Prog. Lang. Syst.*, 15:211–252, Apr. 1993.

[10] R. Harper and G. Morrisett. Compiling with non-paramteric polymorphism (preliminary report). Technical Report CMU-CS-94-122, Carnegie Mellon University School of Computer Science, Feb. 1994.

[11] M. P. Jones. Partial evaluation for dictionary-free overloading. Technical Report YALEU/DCS/RR-959, Yale University Dept. of Computer Scinece, Apr. 1993.

[12] X. Leroy. Unboxed objects and polymorphic typing. In *Proc. Nineteenth Annual ACM Symp. on Principles of Programming Languages*, pages 177–188, New York, January 1992. ACM Press.

[13] X. Leroy and M. Mauny. *The Caml Light system, Release 0.6, Documentation and User's Manual*, 1993.

[14] R. Morrison, A. Dearle, R. Connor, and A. L. Brown. An ad hoc approach to the implementation of polymorphism. *ACM Trans. Prog. Lang. Syst.*, 13(3):342–371, July 1991.

[15] J. Peterson and M. Jones. Implementing type classes. In *Proc. ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 227–236, June 1993.

[16] S. L. Peyton Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference, Keele*, 1993.

[17] M. Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, Edinburgh University, 1988. CST-52-88.

[18] A. P. Tolmach. *Debugging Standard ML*. PhD thesis, Princeton University, Oct. 1992. Also Princeton Univ. Dept. of Computer Science Tech. Rep. CS-TR-378-92.

[19] P. Wadler and S. Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *Proc. Sixteenth Annual ACM Symp. on Principles of Programming Languages*, pages 60–76, January 1989.

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