LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Garbage Collection for Strongly-Typed Languages using Run-time Type Reconstruction

Computation Structures Group Memo 352
March 20, 1994

Shail Aditya
Christine H. Flood
MIT Laboratory for Computer Science
{shail,chf}@lcs.mit.edu

James E. Hicks
Motorola Cambridge Research Center
jamey@mcrc.mot.com

In Proceedings of the ACM Conference on Lisp and Functional Programming, Orlando, Florida, USA, June 27-29, 1994.

The research described in this paper was funded in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-89-J-1988.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Garbage Collection for Strongly-Typed Languages using Run-time Type Reconstruction

Shail Aditya Christine H. Flood
MIT Laboratory for Computer Science
{shail,chf}@lcs.mit.edu

James E. Hicks
Motorola Cambridge Research Center
jamey@mcrc.mot.com

Abstract

Garbage collectors perform two functions: *live-object detection* and *dead-object reclamation*. In this paper, we present a new technique for live-object detection based on run-time type reconstruction for a strongly-typed, polymorphic language. This scheme uses compile-time type information together with the run-time tree of activation frames to determine the exact type of every object participating in the computation. These reconstructed types are then used to identify and traverse the live heap objects during garbage collection.

We describe an implementation of our scheme for the Id parallel programming language compiled for the *T multiprocessor architecture. We present simulation studies that compare the performance of type-reconstructing garbage collection with conservative garbage collection and compiler-directed storage reclamation.

1 Introduction

Dynamic memory management is an integral component of modern programming languages such as C++, Common Lisp, SML, and Haskell that support the notion of a globally shared heap of objects. It is possible to manage the heap memory explicitly by means of explicit allocation and deallocation calls, but usually it is more convenient to use a separate garbage collector that reclaims storage once it is no longer in use. In such cases, the overall performance of the user program depends heavily on the choice of garbage collection technique and its run-time performance.

Abstractly, a garbage collector performs two functions: it distinguishes live objects from those that are garbage (*live-object detection*), and it reclaims the storage allocated to objects that are garbage (*dead-object reclamation*). For live-object detection, the garbage collector must be able to distinguish scalar objects from heap objects and determine their sizes (*object identification*).

Conventional techniques for object identification [3, 5] operate with a very simple memory model and make little or no use of language and compiler-specific information. In this paper, we present a new technique for object identification based on run-time "type reconstruction" for polymorphic languages [1]. Our scheme bridges the gap between the compiler and the garbage collector by using compiler-generated, polymorphic type information and the dynamic call tree of activation frames to compute the exact run-time types of all objects. These exact type are then used by the garbage collector to identify and traverse the live objects residing on the heap. We describe our implementation and compare its performance with other storage management schemes, including a conservative garbage collector that does not use any type information [5] and a compiler-directed storage reclamation scheme that explicitly deallocates objects based on static life-time analysis [9].

1.1 Motivation

A common technique for object identification is to tag every object: a few bits (usually one or two) in every word are used as a *tag* to distinguish scalar objects from pointers to heap objects. Also, heap objects have headers that identify their type and size. Keeping tag bits in every word reduces the range of representable scalars and pointers in conventional architectures, and the user application also pays the additional cost of tag maintenance. In some systems such as SML [3], scalar values (usually floating point numbers) are *boxed* in a heap data-structure to preserve their full range. This incurs the additional cost of allocating the box and accessing it indirectly.

If the semantics of a language necessitates a tagged or boxed representation for objects, or if special hardware support for tags is available, then run-time type reconstruction is probably not the right choice. For example, in the implementation of lazy languages such as Haskell [13], all objects need to be boxed into *closures* unless they are known to contribute towards the final result. These closures can easily identify themselves *via* their code pointers. Similarly, in a dynamically-typed language like Lisp, type reconstruction essentially reduces to continuous type maintenance because there are no static type restrictions on run-time objects. Again, this would be no cheaper than maintaining tags on every object.

The primary motivation for a type-reconstruction-based object identification scheme is to take advantage of the enormous compile-time type information available in a statically-typed language in optimizing its run-time performance. In particular, it is possible in such a system to use a tagless and unboxed representation for scalar objects and eliminate type headers for heap objects because all that type information can be reconstructed when necessary. This not only saves space and unnecessary heap allocations but also saves time spent in tag maintenance. Although the cost of type reconstruction may be significant, it needs to be paid only when the garbage collection is requested. Therefore, such a scheme may work very well for scientific applications where numerical performance is of prime concern and garbage collection is expected to happen infrequently and is used in conjunction with explicit storage management. Keeping tagless data also permits easy interoperability with conventional C and Fortran libraries that do not support tags.

Full run-time type reconstruction also offers some unique advantages that are not present in other untagged schemes for object identification. Having the exact run-time types of objects allows the garbage collector to examine and traverse objects selectively. For example, the collector need not search for heap pointers inside a large array of floating point numbers. Similarly, the scalar fields of a record may be safely skipped. For scientific applications manipulating large numeric arrays, this may constitute a substantial saving in identifying the set of all live objects.

It is also quite easy in this scheme to generate specialized traversal and marking functions for user-defined objects and function activation frames that understand their type and control structure. These functions selectively traverse the fields that point to heap objects as determined by their types, and mark those objects as live. Since these functions are specialized to the type of a particular object, they may be more efficient than interpreting the run-time reconstructed types of the objects.

Finally, the overall framework for type-reconstruction-based object traversal and marking easily extends to other applications such as displaying objects within a debugger, object-oriented I/O etc. This framework serves as a general paradigm for cooperation between the compiler and the run-time system. We have already used this mechanism successfully within the Id debugger for the Monsoon machine for displaying arbitrary Id objects [1].

1.2 Related Work

Conservative and tagged mechanisms have been used extensively in various garbage collection systems (see [15] for a recent survey), but object identification based on type reconstruction for strongly-typed, polymorphic languages is a relatively new idea. The basic concept for type reconstruction was suggested by Appel [2] and expanded by Goldberg and Gloger [6, 7]. These schemes were sometimes unable to compute the exact run-time type of objects embedded inside polymorphic functions. Traversal and marking of such live objects without complete type information remained a problem. On the other hand, our object identification scheme is based on a complete run-time type reconstruction mechanism which ensures that the exact types of all objects are reconstructed even if those objects are created or embedded within polymorphic functions [1].

Our work also relates to compiler-directed storage reclamation shown by Hicks [9] since that scheme also manages dynamic storage in the context of a strongly-typed, polymorphic language without any run-time type-tags. In that scheme, the compiler performs life-time analysis of objects and automatically inserts explicit deallocation calls at appropriate places in the program. The compile-time cost of this analysis can be substantial since it uses abstract interpretation over the whole program, although its run-time cost is minimal. Also, sometimes this scheme is unable to reclaim shared or cyclic data-structures. Thus, in general, this technique has to be used in conjunction with a regular garbage collector that picks up the remaining undetected garbage.

1.3 System Setup

In order to make a reasonable performance comparison of the type-reconstruction-based garbage collection with the conservative and explicit deallocation schemes, we have implemented them for the same source language, compiler, and the target architecture. Our source language is Id, which is a polymorphic, strongly-typed, implicitly parallel programming language [11]. We are compiling Id for the *T multiprocessor architecture [12] and executing it on an emulator for that machine.

We have chosen a very simple mark-and-sweep garbage collection algorithm so that the cost of object identification can be clearly identified during the mark phase. The wall clock performance of the garbage collection algorithm is not our major concern, we are primarily interested in the relative cost of type reconstruction and marking *vs.* the cost of conservative marking. Explicit deallocation scheme serves as a calibration point representing the essential cost of managing the storage.

1.4 Outline

The outline of the rest of the paper is as follows. Section 2 describes the language compilation framework and our run-time model. Section 3 describes our compiler-directed garbage collection scheme based on run-time type reconstruction. In Section 4, we briefly describe the *T multithreaded architecture and our implementation of the various storage management schemes on it. Section 5 discusses our benchmarks and presents the performance results. Finally, Section 6 presents the conclusions.

2 Compilation and Run-time Model

2.1 The Kernel Id Intermediate Language

We base our description on a functional, intermediate language called Kernel Id which is shown in Figure 1. This language supports a rich set of types including typical scalar basetypes, general algebraic (sum-of-products) datatypes, n-dimensional arrays, and curried function types. Records and tuples are a special case of algebraic datatypes with a single product disjunct. We also assume a rich set of primitive functions for basetypes and array construction/selection, as well as standard predefined algebraic datatypes such as *list* and *bool*. The operational semantics of this language has been given elsewhere in terms of graph rewriting rules [4].

The Id source language supports special syntactic constructs such as list and array comprehensions, complex pattern matching, and nested function and type declarations [11]. The Id source program is translated into a Kernel Id program using standard front-end analyses and transformations such as comprehension-desugaring, scope-analysis, type-checking, pattern-matching compilation, and lambda-lifting [4, 8, 14]. The lambda-lifting transformation is not essential for the purpose of this paper, but it helps to simplify our description of the Kernel Id language.

Expressions		
$F, x, y, z \ldots$	$\in$	Identifier
SE	$\in$	Simple Expression
PF^n	$\in$	Primitive Fn. with n arguments
$Case^m\_T$	$\in$	m-way Case Dispatch for type T
E	$\in$	Expression
Constant	::=	*Integer* | *Float*
SE	::=	Identifier | Constant
E	::=	$SE \mid PF^n\ SE_1 \cdots SE_n$
		$\mid Case^m\_T\ SE\ (E_1 \cdots E_m)$
		$\mid SE_0\ SE_1 \cdots SE_n$ | Block
Block	::=	{[Statement;]* in SE}
Statement	::=	Identifier = E

Types		
T^n	$\in$	Type-Identifier with n type arguments
α, β	$\in$	Type-Variable
τ	$\in$	Type
σ	$\in$	Type-Scheme
τ	::=	$\alpha \mid int \mid float \mid (\texttt{n}d\_array\ \tau)$
		$\mid (T^n\ \tau_1 \cdots \tau_n) \mid \tau_1 \rightarrow \tau_2$
σ	::=	$\forall \alpha_1 \cdots \alpha_n . \tau$

Program		
$C_m^{k_m}$	$\in$	m-th Constructor Identifier with k_m arguments
Declaration	::=	Fn-Decl | Type-Decl
Fn-Decl	::=	`def` $F\ x_1 \cdots x_n = E$
Type-Decl	::=	`type` $T\ \alpha_1 \cdots \alpha_n =$
		$C_1\ \tau_{11} \cdots \tau_{1k_1}$
		$\mid \cdots$
		$\mid C_m\ \tau_{m1} \cdots \tau_{mk_m}$
Program	::=	[Declaration;]* E

Figure 1: The Kernel Id Intermediate Language.

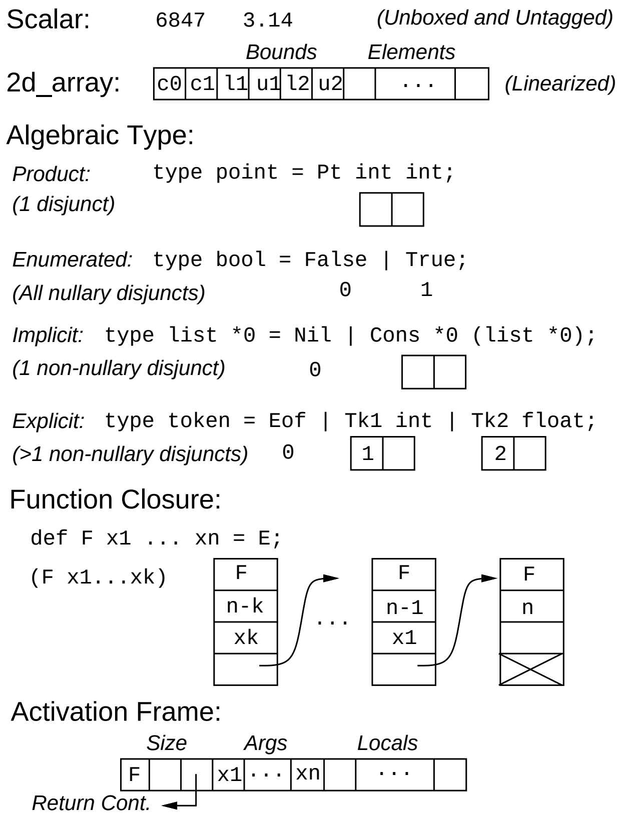

Figure 2: Run-time Object Representations for Id.

2.2 Object Representations and the Memory Model

Kernel Id is an abstract intermediate form that does not take a position on the underlying representation of objects. However, a concrete implementation of a language must specify a representation of objects, which to a large extent, determines its run-time performance and the garbage collection strategy. In this section, we describe the concrete representation of Id objects for our current implementation. This representation is still independent of the target architecture and only relies upon the assumption of a logically flat, shared, global address space. Since our ultimate aim is to get the best performance out of a compiler-directed object identification scheme, we avoid making any assumptions about boxing and explicit tagging of objects as much as possible. The only assumption necessary to support polymorphism is that all scalar objects and pointers to heap objects fit a single 64-bit word.

Examples of various Id object representations appear in Figure 2. Scalar objects are by definition untagged and unboxed in Id. n-dimensional arrays are linearized in row-major order into a flat data-structure that also keeps the bounds in each dimension $(l_1, u_1), \ldots, (l_n, u_n)$ and a set of linearization constants $c_0, \ldots, c_{n-1}$ that are used to compute the linear offset into the array given a n-dimensional index. For an algebraic datatype, depending on the total number m and the arity k_m of its various disjuncts, we may choose one of *product*, *enumerated*, *implicit*, or *explicit* representation. In all cases except when there are more than one non-nullary disjuncts present, we are able to choose an unboxed and untagged representation for the datatype. In particular, when there is exactly one non-nullary disjunct present, as in the case of the *list* datatype, we assume that heap pointers can be distinguished from a small fixed range of integers (say, 0-255), sufficient to represent all the nullary disjuncts of the datatype and no explicit tag is necessary. For some applications, this may save a lot of space and time.

There are two more kinds of objects that are created and manipulated indirectly at run-time by Id programs. These are *function closures* and *activation frames*. In an implementation without lambda-lifting and currying, function closures keep the values of the free identifiers of a function obtained from its lexical environment. In our implementation, all functions are already lambda-lifted, so the closures carry just the curried arguments accumulated under partial applications. We use the structure depicted in Figure 2 which permits sharing of intermediate closures.

An activation frame is a temporary storage area used by an executing function as a scratch pad keeping its input arguments and temporary intermediate values. In Kernel Id, the bound variables of a function constitute the intermediate values that need to be kept within its activation frame for future use.[1] The frame also keeps the return continuation,

[1] An intelligent compiler back-end may be able to share some frame

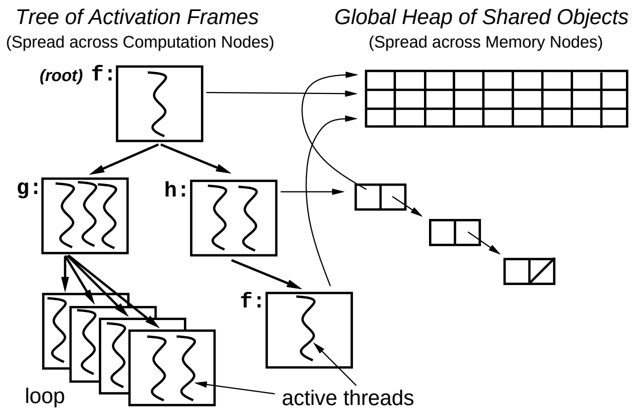

Figure 3: The Parallel Execution Model for Id.

consisting of the caller's activation frame and the return instruction pointer. In a sequential system, activation frames are usually allocated on a stack. In our parallel execution model, the linear stack of activation frames generalizes to a tree and is managed explicitly by the run-time system.

2.3 Id Execution Model

Id is a non-strict, parallel language with an eager evaluation strategy. This implies that an Id program must be compiled and executed in a multi-threaded fashion even on a sequential processor. Parallel, multi-threaded architectures like Monsoon or *T directly support this execution model in hardware, while software multi-threading may be used in sequential workstations or in more conventional parallel architectures like the CM-5.

A program in Kernel Id consists of an expression query to be evaluated within the scope of a set of top-level functions and type declarations. The set of functions is compiled and loaded into the program memory as code-blocks. The program starts by allocating a root activation frame to evaluate the top-level program query. The parallel execution model then unfolds the computation into a tree of function activation frames distributed across the parallel machine (see Figure 3). The frame memory on each processor is separate from the globally shared heap so it is easy to identify the dynamic tree of activation frames at any given point. When the garbage collector is invoked, we assume that all processor registers have been saved back into the activation frame. Therefore, the tree of activation frames constitutes the root set of objects that need to be traversed for the purpose of identifying all reachable heap objects.

3 Garbage Collection using Type Reconstruction

The overall strategy for type-reconstruction-based garbage collection is summarized below and described in the following sections:

1. At compile-time, we ensure that every object manipulated by the user program (including function closures and activation frames) is assigned a static, possibly polymorphic, datatype that accurately describes the structure of that object (Section 3.1).
2. When the garbage collector is invoked at run-time, first we reconstruct the type of every activation frame present within the current dynamic call tree. The reconstruction mechanism instantiates the compile-time type description of each activation frame to its exact run-time type using the algorithm described in [1] (Section 3.2).
3. During the mark phase of the garbage collector, the reconstructed frame-slot types are used to mark the reachable heap objects as live. This may be done in two ways: the reconstructed types may be directly interpreted to identify and traverse the heap objects, or the compiler may automatically generate specialized traversal and mark routines that are appropriately composed at run-time in order to mark the live objects (Section 3.3).
4. Finally, the unmarked objects are reclaimed as garbage by sweeping the entire heap.

slots based on live-variable analysis, but we are ignoring that issue here for simplicity.

3.1 Compiler Support for Object Identification

3.1.1 Visible and Invisible Datatypes

The scalar basetypes, algebraic datatypes, and array types in Kernel Id correspond to pure data-objects whose types are directly *visible* at the source language level. There is a direct, fixed mapping from the source types of these objects to their internal representations as described in Section 2.2. This mapping may be directly used in traversing these objects at run-time once their exact source type is determined.

On the other hand, arrow types ($\rightarrow$) correspond to two different run-time objects: function closures which behave like data-objects that must be garbage collected, and activation frames which are control-objects consisting of the live object root set. Neither of these is modeled completely by the source-level arrow type. This is because the visible type signature of a function does not provide any clue regarding the types of the arguments hidden inside its closure, nor does it provide any information about the local variables kept within the function's activation frame. To remedy this situation, we define *invisible* datatypes for these objects that provide an accurate description of their contents.

3.1.2 Modeling Function Closures

In order to simplify the type reconstruction analysis, we model the closures corresponding to partial applications of a function as disjuncts of an invisible algebraic datatype that is automatically derived at compile-time from the corresponding function signature. This derivation is shown in Figure 4. The various disjuncts of this hidden datatype represent successive partial applications of the function and identify the number and the types of the accumulated arguments. This indirect model captures all the necessary type information required to traverse the actual run-time representation of a function closure as shown in Figure 2. Given a run-time closure object, we can map it to an algebraic disjunct in this model by examining its function code-block pointer and the remaining arity slot. Then, given the exact algebraic type of the closure, the arguments contained within the closure can be traversed using the argument types of the mapped disjunct.

As an example, below we show a function `eqlen` that compares the length of two lists. We also show its Hindley/Milner visible source type and its automatically derived hidden closure datatype:

Example 1:

```
def eqlen l1 l2 =                % eqlen :: ∀αβ.(list α) →
  { len1 = length l1;            %          (list β) → bool
    len2 = length l2;
    p = len1 == len2;
   in p };

type eqlen_closure α β =    % Hidden Closure Type
  eqlen_ap0
| eqlen_ap1 (list α);

f = eqlen (1:nil);  % f :: ∀β.(list β) → bool
                    % f :: ∀β.(eqlen_closure int β)
```

The constructor $\texttt{eqlen\_ap}_0$ models the closure representation of the `eqlen` function itself, while $\texttt{eqlen\_ap}_1$ represents the closure formed by a partial application of the `eqlen` function to one argument. The example also shows the source type and the invisible type of a partial application of the `eqlen` function.[2] Note that the invisible type records the fact that the hidden first argument within the closure is a list of integers while this information is not present in the source type.

There is no need to make a closure for `eqlen` with two arguments since at that point its arity is fully satisfied and the application gives rise to an activation frame instead of a function closure.[3] Finally, note that the invisible closure datatype is parameterized by *all* the type-variables present in the source type of the function. This is necessary in order to model the exact run-time types of all the arguments contained within the closure.

3.1.3 Modeling Activation Frames

Function activation frames are modeled using an automatically derived, invisible datatype called the function *typemap*. One typemap datatype is defined for each function definition in the program as shown in Figure 4. The typemap of a function captures all the necessary compile-time information to reconstruct the complete type of its activation frame at run-time using the mechanism described in [1].

The typemap of a function is parameterized by the type-variables present in the function's type signature. The typemap records the function's signature, the types of its arguments and its bound identifiers, and the compile-time type-instances of all function application sites within the body of the given function. The argument and the bound identifier types model the corresponding frame slots of the function's activation frame, while the application site information is used for type reconstruction. Once all the parameter type-variables of a function's typemap are properly instantiated *via* type reconstruction, the typemap datatype provides the exact run-time type of all objects accessible through each argument and bound identifier slot of the corresponding activation frame.

As an example, we show the typemap datatype for the `eqlen` function given above:

Example 2:

```
type eqlen_typemap α β =
  {record
    (eqlen    :: (list α) → (list β) → bool)
    (l1       :: (list α))
    (l2       :: (list β))
    (len1     :: int)
    (len2     :: int)
    (p        :: bool)
    (length1  :: (list α) → int)
    (length2  :: (list β) → int) };
```

Note that the two application sites of the `length` function are separately recorded with their own compile-time type-instances.

INVISIBLE DATATYPES

Given a Function Declaration:
  def $F\ x_1 \cdots x_n = E$
  $F :: \forall\alpha_1 \cdots \alpha_m.\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau_{n+1}$

Define Function Closure Datatype:
  type F_closure $\alpha_1 \cdots \alpha_m$ =
      $F\_\mathtt{ap}_0$
    | $F\_\mathtt{ap}_1\ \tau_1$
    | ...
    | $F\_\mathtt{ap}_{n-1}\ \tau_1 \cdots \tau_{n-1}$;

Let $(z_1 :: \sigma_1) \cdots (z_k :: \sigma_k)$ be the Bound Identifiers of E.
Let $(f_1 :: \tau_{f_1}) \cdots (f_l :: \tau_{f_l})$ be the type-instances of all function application sites within E.

Define Function Typemap Datatype:
  type F_typemap $\alpha_1 \cdots \alpha_m$ =
   {record
     $(F :: \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau_{n+1})$ % Fn. Signature
     $(x_1 :: \tau_1)$ % Argument Types
     ...
     $(x_n :: \tau_n)$
     $(z_1 :: \sigma_1)$ % Bound Identifier Types
     ...
     $(z_k :: \sigma_k)$
     $(f_1 :: \tau_{f_1})$ % Application Site Types
     ...
     $(f_l :: \tau_{f_l})$ };

Figure 4: Automatic Derivation of Invisible Datatypes.

3.2 Compiler-directed Type Reconstruction

Compiler-directed type reconstruction requires some cooperation between the compiler and the run-time system. In the preceding section, we described the compiler support necessary to facilitate run-time type reconstruction. In this section, we describe the run-time data-structures and the reconstruction algorithm. A more detailed description of the algorithm may be found in [1].

3.2.1 Run-Time Type Encodings

Run-time type reconstruction requires an *encoding* of the datatypes that may be computed and instantiated during reconstruction. Each datatype T^n is encoded into a corresponding type descriptor $\overline{T^n}$ that contains all the necessary compiler information about its arity, internal field structure, and its representation. Conceptually, for each polymorphic

[2] ":" is the infix *cons* constructor for lists.

[3] However, under delayed or lazy evaluation, we may need to keep track of such *thunks*.

Encoding Schema $\mathcal{C}$
Given a Type T^n, define $\mathcal{C}[\![T^n]\!] = \texttt{encode\_}T$, where `def encode_`$T$ $(x_1, \ldots, x_n) = pack(\overline{T^n}, x_1, \ldots, x_n)$ Given a polymorphic type-variable α_i, define $\mathcal{C}[\![T^0_{\alpha_i}]\!] = \texttt{encode\_}T_{\alpha_i}$, where `def encode_`$T_{\alpha_i}$ $() = \overline{T^0_{\alpha_i}}$

Figure 5: Run-time Encodings for Datatypes.

datatype T^n with n type parameters, we define a corresponding encoding function `encode_`T that takes n encoded arguments and *pack*s them along with the encoded type descriptor $\overline{T^n}$ into a single type data-structure. Such an *encoding schema* $\mathcal{C}$ is shown in Figure 5. These functions may be used to create encodings of type-instances computed during type reconstruction. The bound type variables α_i of a polymorphic type-scheme $\forall\alpha_1 \cdots \alpha_n.\tau$ are encoded as special parameterless datatypes $\overline{T^0_{\alpha_i}}$ because they do not participate in type reconstruction.[4]

In practice, our compiler generates static type descriptors $\overline{T^n}$ for all the user-defined algebraic datatypes within the program and for the automatically derived closure and typemap datatypes corresponding to every function in the program. These static descriptors are linked together with the program and are used during type reconstruction. Run-time types are encoded as a flat array of static type descriptors using back-pointers to preserve sharing. This representation permits very efficient copying, unification, and instantiation operations on encoded types. The packing and unpacking of these encoded types is carried out on the fly within the run-time system, therefore no separate encoding functions are actually necessary.

3.2.2 Run-Time Type Reconstruction

When the type reconstruction of an activation frame is requested at run-time, we check whether the corresponding function is polymorphic and therefore may need some additional type context in order to properly instantiate its typemap. If not, then the compiler-generated encoding of the function's typemap already contains all the type information necessary to traverse the corresponding activation frame and no more work is necessary.

If the current function is polymorphic, then its parent activation frame is identified using the return continuation and is recursively type reconstructed. This recursive process continues towards the root of the dynamic call tree until a non-polymorphic function or the root frame is reached. At the root, the types of all the user-supplied arguments completely determine the type context for instantiating the root typemap. On unwinding the recursion back to the original activation frame, the return continuation in the current frame identifies its particular application site recorded within the parent's typemap. Now, the fully reconstructed application site type from the parent's typemap is matched against the current function's type signature to determine the exact type-instances of its polymorphic type parameters.

[4] This is because the exact run-time type of an object with a closed polymorphic type (*e.g.*, nil :: $\forall\alpha.(list\ \alpha)$) is that polymorphic type itself, see [1] for details.

This instantiates the typemap of the current function which can then be used to traverse its activation frame.

Note that the entire typemap for an activation frame is instantiated at once and is remembered subsequently. Therefore, no activation frame needs to be reconstructed more than once even if it has several branches below it. Thus, the overall complexity of reconstructing every frame in the dynamic call tree is proportional to the total number of activation frames present in the tree.[5]

The only problem with the above scheme is that sometimes the type-instance of an application site may not be sufficient to instantiate all the type parameters of a function's typemap or its hidden closure type. As an example, consider the following situation involving the `eqlen` function shown earlier:

Example 3:

`eqlen` :: $\forall\alpha\beta.(list\ \alpha) \rightarrow (list\ \beta) \rightarrow bool$
`f = eqlen (1:nil);` `% f ::` $\forall\beta.(list\ \beta) \rightarrow bool$
`%` Instantiation $\{\alpha \mapsto int\}$
`...`
`p = f ("foo":nil);` `%` Instantiation $\{\beta \mapsto string\}$

At the first application site, the function `eqlen` is partially applied to create a closure `f`. Neither the source type, nor the subsequent type-instance of this closure at the second application site reflect the hidden type instantiation $\{\alpha \mapsto int\}$. Without this information, the exact type of the elements of the list hidden inside the closure `f` can not be reconstructed. In such a situation, Goldberg and Gloger [7] argue that since those elements are never accessed by the function `eqlen`, they need not be traversed and marked as live. Only the spine of the list is marked as live. Unfortunately, if this list is shared among other activation frames its elements may still be live and may not be garbage collected.

Managing partially marked, shared objects is a difficult and unsolved problem [7]. Instead, our scheme guarantees full type reconstruction at every object reference so that an object can be marked completely the first time around. In the above example, our reconstruction mechanism encodes the hidden type instantiation explicitly at the first application site and passes it to the `eqlen` function as an additional parameter [1]. This encoded *type-hint* is directly used to reconstruct the invisible type of the closure `f` and to instantiate `eqlen`'s typemap. This is the only situation where an explicit, compiler-generated encoded type needs to be propagated at run-time to ensure full type reconstruction.

3.3 Object Traversal and Marking

In this section, we describe our scheme for object traversal and marking after complete type reconstruction has been performed. We present two mechanisms:

Interpreted Marking – In this mechanism, the encoded types generated by type reconstruction are directly used to guide the traversal and marking of the heap objects.

Compiled Marking – In this mechanism, the compiler automatically generates marking functions for each datatype in the program based solely on the static type information. These functions are appropriately composed at run-time using the reconstructed types and then directly applied to the corresponding objects.

[5] Each reconstruction step manipulates Hindley/Milner types of the original program which in the worst case could be very large [10], but such cases rarely occur in practice.

Marking Schema $\mathcal{M}$

Given a Type T^n, define
$\mathcal{M}[\![T^n]\!] = \texttt{mark\_}T$, where

1. T^0 is a BaseType (*int* | *float*):
   $\texttt{def mark\_}T\ () = \lambda x.()$

2. T^1 is an ArrayType ($\texttt{nd\_}array\ \alpha$):

$$
\begin{array}{l}
\texttt{def mark\_}nd\_array\ (z) = \\
\quad \lambda a.\{\ Mark(a); \\
\qquad (l_1, u_1), \ldots, (l_n, u_n) = bounds(a); \\
\qquad \texttt{for } i_1 \leftarrow l_1 \texttt{ to } u_1\ \cdots \\
\qquad \texttt{for } i_n \leftarrow l_n \texttt{ to } u_n \\
\qquad\quad \mathcal{I}nterpret[\![\mathcal{M}]\!] \\
\qquad\qquad (\mathcal{C}ompose[\![\mathcal{C}]\!]\ \{\alpha \mapsto z\}\ \alpha) \\
\qquad\qquad a[i_1, \ldots, i_n]; \}
\end{array}
$$

3. T^n is an Algebraic DataType ($T^n\ \alpha_1 \cdots \alpha_n$):

$$
\begin{array}{l}
\texttt{def mark\_}T\ (z_1, \ldots, z_n) = \\
\quad \lambda x.\{\ Mark(x); \\
\qquad \texttt{Case\_}T\ x \\
\qquad\quad C_1\ x_1 \cdots x_{k_1} = \\
\qquad\quad \{\ \mathcal{I}nterpret[\![\mathcal{M}]\!] \\
\qquad\qquad (\mathcal{C}ompose[\![\mathcal{C}]\!]\ \{\alpha_i \mapsto z_i\}\ \tau_{11})\ x_1; \\
\qquad\quad\ \cdots \\
\qquad\quad\ \mathcal{I}nterpret[\![\mathcal{M}]\!] \\
\qquad\qquad (\mathcal{C}ompose[\![\mathcal{C}]\!]\ \{\alpha_i \mapsto z_i\}\ \tau_{1k_1})\ x_{k_1}; \} \\
\qquad \mid \cdots \\
\qquad \mid C_m\ x_1 \cdots x_{k_m} = \cdots \}
\end{array}
$$

Given a polymorphic type-variable α_i, define
$\mathcal{M}[\![T^0_{\alpha_i}]\!] = \texttt{mark\_}T_{\alpha_i}$, where
$\texttt{def mark\_}T_{\alpha_i}\ () = \lambda x.()$

Figure 6: Generating Mark Functions for Datatypes.

Both mechanisms are specified as a set of mark functions, one for each basetype, array type, and algebraic datatype present in the program. The algebraic datatype could be a user-defined datatype (Figure 1) or an invisible datatype defined by the compiler for function closures and activation frames (Figure 4).

3.3.1 Interpreted Marking

The *Interpreted Marking Schema* $\mathcal{M}$ for a type T^n is shown in Figure 6. In this schema, for each type T^n with n type parameters $\alpha_1 \cdots \alpha_n$, we define a mark function $\texttt{mark\_}T$ that is parameterized by n corresponding encoded type arguments $z_1 \cdots z_n$. At run-time, this function is supplied with the exact encoded type instantiation of its type parameters, say $\overline{\tau_1} \cdots \overline{\tau_n}$, which produces an appropriate marking function for an object with type ($T^n\ \tau_1 \cdots \tau_n$).

The internal structure of the mark functions closely follows the structure of their corresponding datatypes. All our base types are scalars, so the mark functions for them do nothing. The mark function for arrays and algebraic datatypes first mark the object itself and then proceed to mark their internal components. This is achieved by first building their exact run-time type encoding by appropriately *composing* the encoding functions given by Figure 5, and then recursively *interpreting* that encoding. Finally, the polymorphic, bound type-variables of a type-scheme are also mapped to dummy mark functions because polymorphic objects contain no information.

Type-based Translation

Given a Compilation Schema $\mathcal{R}$,
and a Translation Environment ρ_r, define

$$
\begin{array}{lll}
\mathcal{C}ompose[\![\mathcal{R}]\!]\ \rho_r\quad \alpha & = & \rho_r(\alpha) \\
\mathcal{C}ompose[\![\mathcal{R}]\!]\ \rho_r\quad (T^n\ \tau_1 \cdots \tau_n) & = & \\
\quad (\mathcal{R}[\![T]\!])\ (\mathcal{C}ompose[\![\mathcal{R}]\!]\ \rho_r\ \tau_1, \ldots, \mathcal{C}ompose[\![\mathcal{R}]\!]\ \rho_r\ \tau_n) & & \\
\mathcal{C}ompose[\![\mathcal{R}]\!]\ \rho_r\quad \forall \alpha_1 \cdots \alpha_n.\tau & = & \\
\quad \mathcal{C}ompose[\![\mathcal{R}]\!]\ \rho_r\ (\tau[T^0_{\alpha_i}/\alpha_i]) & &
\end{array}
$$

Type-code Interpretation

Given a Compilation Schema $\mathcal{R}$, define

$$
\begin{array}{l}
\mathcal{I}nterpret[\![\mathcal{R}]\!]\ \overline{\tau} = \\
\quad \{\texttt{Case}\ head(\overline{\tau}) \\
\qquad \overline{T^n_1} = \{\ x_1, \ldots, x_n = arguments(\overline{\tau}); \\
\qquad\qquad \texttt{in}\ (\mathcal{R}[\![T^n_1]\!])\ (x_1, \ldots, x_n)\ \} \\
\quad \mid \overline{T^m_2} = \cdots \\
\quad \mid \cdots \}\}
\end{array}
$$

Figure 7: Type-based Translation and Interpretation.

The general mechanism of type-based function composition ($\mathcal{C}ompose[\![\mathcal{R}]\!]\ \rho_r\ \tau$) for an arbitrary schema $\mathcal{R}$ (such as the encoding schema $\mathcal{C}$ or the mark schema $\mathcal{M}$) is shown in Figure 7. This process translates a given type τ into a composition of schema functions specified by $\mathcal{R}$ under a translation environment ρ_r that maps free type variables of τ to schema-dependent values. In our present case, ($\mathcal{C}ompose[\![\mathcal{C}]\!]\ \{\alpha_i \mapsto z_i\}\ \tau$) generates a function composition that computes the type encoding of the exact run-time type instantiation of the static type τ.

Similarly, the mechanism of interpreting the type encodings ($\mathcal{I}nterpret[\![\mathcal{R}]\!]\ \overline{\tau}$) can also be generalized for an arbitrary schema $\mathcal{R}$ as shown in Figure 7. This process unpacks the encoded type and invokes the schema function for the appropriate type descriptor passing it the rest of the encoded type arguments. In our present case, ($\mathcal{I}nterpret[\![\mathcal{M}]\!]\ \overline{\tau}\ x$) traverses and marks the object x according to its exact run-time type encoding $\overline{\tau}$ by recursively instantiating and invoking the mark functions associated with the type descriptors in τ.

In our current implementation, the type-code interpretation mechanism of Figure 7 is built into the run-time system. It directly executes the marking specification of Figure 6 corresponding to each class of datatypes given a run-time object x and its reconstructed run-time type encoding $\overline{\tau}$. No separate marking functions are generated.

The interpretive marking process starts up by examining the frame-slots of every activation frame using the argument and the bound identifier types from its reconstructed type-map. This process is optimized based on the actual representation chosen for a particular class of datatypes as shown in Figure 2. For example, the marking function for linearized arrays computes the total size of the array and marks each of its elements in a single loop. In case of algebraic types, nullary disjuncts under enumerated or implicit representation are never marked, a product disjunct is always marked, and a tag dispatch is made for explicitly tagged disjuncts. Finally, the hidden arguments inside function closures are traversed and marked according to their reconstructed hidden closure types.

3.3.2 Compiled Marking

Rather than interpreting type encodings as in our interpreted marking schema, it is also possible to generate compiled marking functions for each datatype that know how to traverse the object directly without any type interpretation. In this *Compiled Marking Schema* $\mathcal{M}'$, for each datatype T^n the compiler automatically generates a mark function `mark'_T` that is parameterized by n mark function arguments $f_1 \cdots f_n$ instead of encoded type arguments.

This alternate marking schema $\mathcal{M}'$ can be directly obtained from our interpreted marking schema $\mathcal{M}$ shown in Figure 6 by replacing the recursive call for interpretation:

$$\mathcal{I}nterpret[\![\mathcal{M}]\!]\ (\mathcal{C}ompose[\![\mathcal{C}]\!]\ \{\alpha_i \mapsto z_i\}\ \tau)$$

by a type-based function composition:

$$(\mathcal{C}ompose[\![\mathcal{M}']\!]\ \{\alpha_i \mapsto f_i\}\ \tau)$$

This transformation expresses the fact that building the exact run-time type encoding of an object and then interpreting it to guide the traversal and marking is functionally equivalent to directly traversing it using a compiled marking function that knows the structure of that object. Note that marking functions so generated do not contain any type-code interpretation. Their execution directly results into the appropriate traversal and marking of the given object.

The compiled marking process is initiated by converting the reconstructed typemap of each activation frame into a composition of compiler-generated marking functions. This translation is similar to the type-based function composition shown in Figure 7 except that it operates on type encodings rather than static types. The resulting function composition may be directly executed to mark all objects reachable from the activation frame. The compiled marking schema is currently unimplemented.

4 Implementation of Id on *T

*T is a parallel, distributed-memory machine [12]. The *T architecture extends a basic RISC instruction set with low-overhead, user-mode communication and synchronization primitives. In this paper, we use a simulator for a machine based on the 88110MP processor. The 88110MP is Motorola's superscalar RISC processor extended with an on-chip message and synchronization unit (MSU). On this machine, messages consist of 4 to 24 32-bit words. A full-sized message may be composed from data present in general-purpose registers and sent in 6 instructions.

*T runs a Unix-like operating system. A parallel job running on *T consists of a separate process, or *player*, on each processor. These players have independent virtual address spaces, but may refer to a global 64-bit address space through the MSU by using split-phase transactions.

In the rest of this section, we present some details of the implementation of Id on *T and of performing distributed garbage collection on this machine.

4.1 Microthread Scheduling on *T

The 88110MP MSU provides hardware support for scheduling *microthreads*. A microthread is a compiler-defined thread, described by an instruction pointer (IP) and a frame pointer (FP). A microthread, by definition, executes to completion once it has been invoked. It may send messages or fork other microthreads that are deposited in a stack of ready-to-run microthreads.

Messages always contain a microthread descriptor as the first two words of payload. Normally, messages are handled by invoking the microthread described within the message, so these microthreads are termed *message handlers*.

A microthread's last operation is to schedule the next microthread of the highest priority. The MSU provides a `sched` instruction that returns the IP,FP of the next microthread to be run in a 64-bit register pair. The highest priority microthread is selected from a simple priority queue consisting of incoming message handlers, the microthread stack, and several microthread registers. Message handlers have higher priority than computation microthreads.

4.2 Layout of Memory

The memory of an Id process on *T is divided into six areas that are themselves distributed or replicated across the nodes:

Memory Area	Type
Code	replicated
Static Data	replicated
C Stack	distributed
Frame	distributed
Heap	distributed
Presence Bits	distributed

The code and static data areas are replicated — each node gets a copy of the whole program and all of its constants. Each node also has a stack that is used for calling into C procedures from Id. The Id run-time system is implemented in C and may also use the C stack.

The frame area contains the activation frames for each Id procedure that is invoked. When a procedure is invoked, the run-time system chooses a processor on which to allocate the frame, then sends a message to that processor, which allocates a frame in its own frame area.

The heap area contains all of the heap-allocated Id objects. In our implementation of Id on *T, all scalar objects and pointers to heap objects are 64 bits in size. Furthermore, these pointers are always aligned on 8-byte boundaries when stored in memory. Each 64-bit word in the heap has an associated 2-bit presence value in the presence-bit area. These presence bits are used to implement Id's non-strict array, I-structure, and M-structure operations.

Our compiler and run-time system never store a pointer to the interior of an object in a frame-slot or another Id object. Therefore, a pointer found within a frame or a heap object always points to the head of the *active area* of the object. The active area of the object is actually preceded in memory by some information managed by the run-time system including the object's size (used for deallocation) and the time when it was allocated (in instruction cycles — for statistics collection).

4.3 Garbage Collection on *T

Garbage collection on *T can be initiated either by request from the Id program or by the run-time system when one of the processors finds out that it is running out of heap storage. Our current policy is to initiate garbage collection when the allocated storage on a node reaches a specified fraction (say, 0.75) of its total storage.

Since the heap is shared globally, all processors must participate in a global garbage collection. Therefore, when one processor decides to do garbage collection, all other processors are informed about it. Currently, we have implemented a simple stop-and-collect garbage collection scheme.

First, the processors drain all messages out of the network because the messages may carry live pointers to heap objects. As messages are drained from the network, their handlers are invoked. Our message handlers can modify memory locations or fork other microthreads, but they cannot send messages. We can handle all messages and eventually reach quiescence, as long as we do not run any threads scheduled by the message handlers. Since we invoke message handlers as the network drains, there are no queues of messages to consider as part of the root-set during garbage collection.

Once the network is drained, all processors synchronize and then initiate the mark phase. In this phase, all live and reachable objects are marked according to one of the object identification techniques starting from the distributed root set of activation frames currently in use. This process requires global communication among processors to mark objects distributed across the machine. After global marking is completed on all nodes, the processors synchronize again and then each processor begins a local sweep phase. A final synchronization is performed after sweeping is completed on all nodes, and then the Id threads are allowed to resume computation.

4.3.1 Type-Reconstructed Garbage Collection

The mark phase of the Type-Reconstructed Garbage Collection (TRGC) follows the compiler-directed object identification scheme described in the last section. Currently, we have only implemented the interpreted marking scheme with full type reconstruction. During the mark phase, the frame memory of each processor is traversed locally to find the activation frames that belong to the current dynamic call tree. Each activation frame that is currently in use is type-reconstructed and then its contents are searched for heap objects to be marked using their reconstructed types. The type-reconstruction of a frame may be overlapped with marking of objects in another frame, so the overall cost of type-reconstructed marking is a combination of the two.

4.3.2 Conservative Garbage Collection

The mark phase of the Conservative Garbage Collection (CGC) [5] requires no source type information. Conservative garbage collectors use a simple, conservative test to determine whether a value in a frame or a heap object is a pointer to another object. Since pointers are identified conservatively, CGC may assume that there are live references to an object when there are none, therefore some objects may remain uncollected. Also, CGC cannot compact or copy objects because conservatively identified pointers cannot be updated. Finally, CGC has no knowledge of the source types, therefore it must examine every slot of every reachable object and no short-circuiting based on scalar-type information is possible.

In our system, the conservative pointer test is implemented as follows. CGC tests each 64-bit value to see if it is aligned to a 64-bit boundary and if it points within the heap area. Then, it checks to see if the value points to the head of a heap object. This is a very simple test because in our compilation model, actual pointers never point to the interior of objects. Furthermore, the run-time system marks the head of each allocated object with a special presence-bit pattern. Therefore, CGC simply checks for this pattern at the head of the current value. If this test succeeds then the value is considered to be a pointer and the object is marked. This test may mark some objects that are not actually reachable because a value in memory happens to look like a pointer to that object. However, the test is guaranteed to mark only actual heap objects.

4.3.3 Compiler-Directed Storage Reclamation

For comparison purposes, we have also implemented the explicit, compiler-directed storage reclamation scheme (CDSR) [9] within the same compiler and run-time system framework. In this scheme, no separate garbage collection needs to be performed: the compiler inserts code to deallocate an object when it can determine the object to be garbage. This analysis has a substantial compile-time cost and also a small run-time synchronization cost that is somewhat difficult to separate from the cost of the Id application. Also, the static analysis may not be able to reclaim all the garbage that is generated by the program. So, we present this scheme only to compare its relative storage management efficiency to that of the garbage collected schemes. It is also possible to simultaneously use the explicit storage management scheme to get most of the large objects along with a garbage collector that catches the smaller, harder to analyze objects. We believe that a mixed approach would yield better performance than either scheme on its own.

5 Performance

We are interested in two aspects of the performance of the type-reconstructed garbage collection (TRGC): how long it takes to garbage collect, and how much garbage it reclaims. We compared several programs running with TRGC, conservative garbage collection (CGC), and compiler-directed storage reclamation (CDSR).

In preparing a uniform execution platform, we naturally had to accommodate the requirements of each storage management scheme within the same run-time system. This resulted in a system that was not tuned to any particular storage management scheme. For instance, a copying garbage collector could not be used for TRGC since CGC would not work in that setup. Thus, the results we obtained cannot be treated as an absolute measure of performance for any particular scheme. On the other hand, they provide a good measure of relative performance of the object identification mechanisms studied and also characterize systems where more than one storage management strategy is used.

5.1 Benchmarking Setup

We used four different benchmarks. *Quicksort* is the standard recursive algorithm for sorting N list elements parameterized by a polymorphic comparison predicate. *Paraffins* generates and counts the number of distinct paraffin isomers of up to N carbon atoms. *Gamteb* is a Monte Carlo simulation of N photons impinging on a carbon rod divided into two cells. Finally, *Wavefront* consists of 10 iterations of a

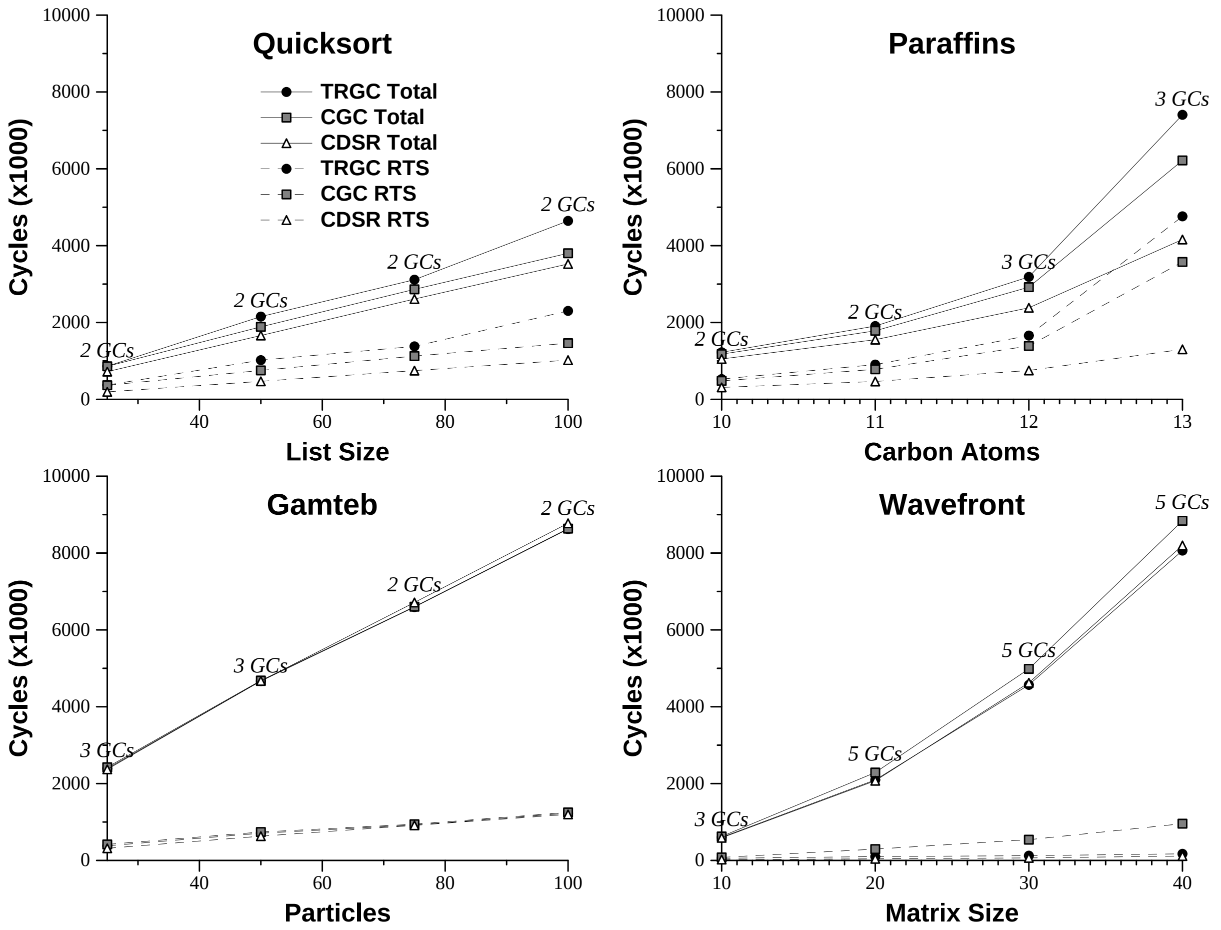

Figure 8: Total Cost and Run-time System Cost for the Benchmarks.

successive over-relaxation kernel of a $N \times N$ matrix containing floating-point data.

For each of the programs we tested, we ran three versions: TRGC, CGC, and CDSR. The TRGC version is the program running with type-reconstructing garbage collection. The CGC version is running with conservative garbage collection, and the CDSR is the automatically annotated version running with no garbage collection. Both garbage collectors use the mark and sweep algorithm, and use the same implementation of sweeping and inter-processor synchronization. Using a simple GC algorithm allowed us to separate the basic heap management cost (allocation and deallocation) from the overall cost of garbage collection. Thus, the cost of object traversal and marking of TRGC and CGC can be truly ascribed to their respective object identification strategies.

In all three cases, actual heap storage management and statistics collection is performed by the same Id run-time system. Although statistics gathering is mildly intrusive, it constitutes a tiny fraction of total cycles executed. Online statistics processing (resampling profiles) is not counted.

5.2 Benchmark Runs and Discussion

We simulated several problem sizes for a single processor with each program and storage management scheme. The TRGC and CGC runs were made with sufficient storage to avoid thrashing. Each garbage collected run also performed a final GC at the end of the run to reclaim all the uncollected garbage.

The total instruction cycles and the cycles spent in the run-time system and garbage collection for all the runs are shown in Figure 8.[6] The numbers appearing on top of the curves are the number of garbage collections performed by TRGC and CGC for that run. Both TRGC and CGC did the same number of garbage collections in all runs. Garbage collection was switched off for CDSR runs. These curves give an idea of the growth of run-time system cost of the various schemes as a function of problem size and as a fraction of the total cost.

5.2.1 Time Analysis

Several trends are apparent from Figure 8. The CDSR scheme consistently has the lowest run-time cost since it does not perform any garbage collection and only incurs the basic heap and frame management cost (allocation and deallocation). The fraction of time spent in the run-time system varies widely depending upon the nature of the ap-

[6] More detailed numbers are available from the authors. We could not present all the raw numbers here for lack of space.

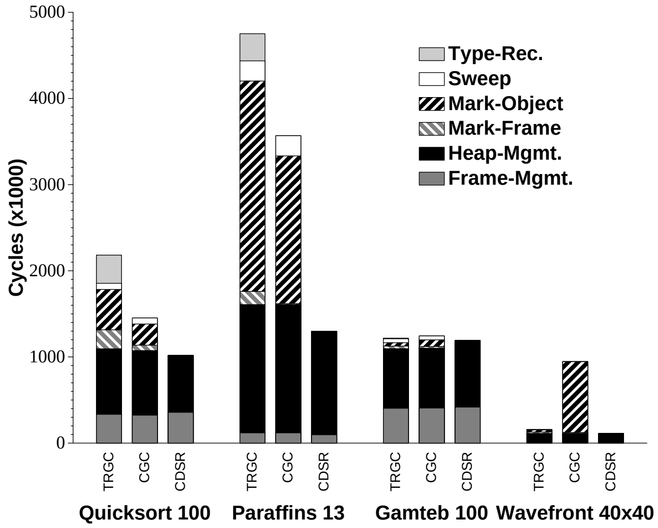

Figure 9: Run-time System Cost Breakup.

plication and the cost and the number of garbage collections performed. For example, Paraffins allocates a lot of small-sized data-structures keeping them live until the very end. Thus, each mark phase has to do a lot of work. Similarly, Quicksort rapidly unfolds into a tree of activation frames each of which holds onto a substantial amount of storage, so the cost of marking is high there as well. On the other hand, for Gamteb, the size of the live heap is quite small so the garbage collected schemes incur very little overhead.

Comparing the relative run-time costs of TRGC and the CGC, we find that for Quicksort and Paraffins TRGC does much worse than CGC while for Wavefront it performs much better. This wide variation can be explained by looking at the normalized run-time cost breakup shown in Figure 9 for the largest sized runs. We break the run-time system cost into basic frame and heap management cost, type reconstruction cost, the cost of marking frames and objects, and sweeping cost.

For both Quicksort and Paraffins, TRGC spends a significant amount of time in the type reconstruction phase. This is due to the fact that they both contain polymorphic functions so the type reconstruction mechanism has to generate and propagate the exact run-time type instantiation down from the root to each frame in the dynamic call tree. This reconstruction cost is much less visible in Gamteb and Wavefront that are not polymorphic and largely consist of first-order functions. Furthermore, the run-time types are represented as C data-structures and are currently managed using conventional `malloc` and `free` system calls. This cost can be substantially reduced by using a specialized version of `malloc`.

The marking cost of TRGC is also about 1.5-2.2 times higher than that of CGC in case of Quicksort of 10 elements and Paraffins of 13 carbon atoms. Our current implementation interprets the type structures at run-time in order to traverse and mark the corresponding run-time objects. This interpretation overhead could be eliminated by using the compiled marking schema as described in Section 3.3.2 where the compiler generates a specialized marking routine for each source type parameterized over its polymorphic variables. Furthermore, these routines can be inlined to produce highly optimized traversal and marking functions for each user-defined function activation frame.

In the case of Wavefront, TRGC takes much less time than CGC, and very little more time than CDSR, where no marking at all took place (see Figure 8). For Wavefront of 40×40, the marking cost of CGC is 25 times higher than that of TRGC. TRGC did so well because it could determine that the arrays contained only scalar data by inspecting their run-time type. Therefore, it only marked the arrays themselves and did not scan for pointers inside them, as CGC did. This scanning cost depends on the total size of the arrays and was responsible for the quadratic growth in run-time cost for CGC in Figure 8. However, sweeping took the same amount of time for both TRGC and CGC (see Figure 9).

The wavefront example shows that in an ideal situation, the time to mark the heap for TRGC is proportional to the total *number* of live object references, rather than the total *amount* of live storage as it is for CGC. TRGC can use the reconstructed type information to avoid scanning elements of scalar arrays and scalar fields within records and algebraic types.

5.2.2 Space Analysis

In terms of space usage, both TRGC and CGC perform identically. CGC is able to reclaim all the garbage because of our restrictive compilation model and support from the run-time system (see Section 4.3.2). The performance of CDSR varies with the application. For Gamteb and Wavefront, CDSR is able to insert deallocation commands to reclaim all the garbage automatically. Therefore, these benchmarks are able to run under CDSR without leaking any storage. The garbage collected versions for these benchmarks had to be given 2-10 times the storage used by CDSR to avoid thrashing. On the other hand, for Paraffins and Quicksort, CDSR is able to reclaim only 10-20% of the total garbage, therefore the TRGC and CGC versions are able to run in same or less storage than the CDSR version without thrashing.

6 Conclusions

In this paper, we have described a scheme for garbage collection of Id programs using run-time type reconstruction (TRGC). We described the compiler and run-time mechanisms required to reconstruct the exact types of all run-time objects. We also described an interpreted and a compiled marking schema for traversing and marking live run-time objects using the reconstructed type information. We have implemented the interpreted marking schema on a simulator for the *T architecture and compared its performance with conservative garbage collection (CGC) and compiler-directed storage reclamation (CDSR) on several benchmarks.

Our results show that in general, TRGC does more work in marking the live objects than CGC, unless it can avoid scanning large, scalar, array-like objects using type information. The type reconstruction overhead increases with the amount of polymorphism and higher-order functions (closures) used in the program, although the cost of reconstruction is small compared to the cost of marking live objects with type interpretation. The cost of interpreted marking itself should get reduced considerably using the compiled marking schema instead of type interpretation.

TRGC has the additional advantage that other storage reclamation schemes may be used, such as compaction or copying. These may not be used with CGC because they require the update of live pointers, and CGC cannot guarantee that what it uses as a pointer is not really a scalar value. On the other hand, TRGC requires initialization of polymorphic and pointer data with valid values and cannot cope with stale data as CGC can.

CDSR consistently does better than either of the garbage collection schemes in terms of time spent in the run-time system. This is as expected, although sometimes it is not able to collect all the garbage and therefore requires more memory than strictly necessary. CDSR also takes much longer to compile, sometimes increasing compile-time by a factor of 10.

On the whole, type reconstruction and type-reconstruction-based garbage collection seem to be a promising area of research with a lot of scope for compiler optimization and run-time performance improvement. This initial study has shown that type reconstruction based garbage collection is certainly feasible and can be competitive with other storage management strategies under the right mix of applications.

6.1 Future Work

There are several dimensions in which further investigation would be useful. We already plan to implement the compiled marking schema and compare its performance with our current interpreted marking schema. We expect to see a substantial improvement in performance using specialized marking functions. We also plan to investigate mixed storage management schemes that combine garbage collection with explicit storage reclamation within the same run-time environment.

Although our system has been designed and implemented for a multi-processor architecture, we have currently made a study for only a single processor. We would like to see how TRGC scales under a multi-processor environment and quantify the inter-processor communication overhead for type reconstruction.

It would be very interesting to compare the performance of TRGC with an explicitly tagged object identification scheme implemented within the same framework. It would be interesting to know if TRGC offers any concrete advantages over that technique.

Finally, it would be useful to implement a compacting garbage collector based on type reconstruction with a very simple allocation scheme (bumping a pointer) and compare its heap management overhead with that of the CGC and CDSR that require a more sophisticated storage management scheme (free lists).

7 Acknowledgments

The authors would like to thank Prof. Arvind, Mike Ernst, David Moon and the anonymous referees for their insightful comments and suggestions.

The research described in this paper was funded in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-89-J-1988.

References

[1] Shail Aditya and Alejandro Caro. Compiler-directed Type Reconstruction for Polymorphic Languages. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark*, pages 74–82, June 1993.

[2] Andrew W. Appel. Runtime tags aren't necessary. *Lisp and Symbolic Computation*, 2(2):153–163, June 1989.

[3] Andrew W. Appel. A runtime system. *Lisp and Symbolic Computation*, 3(4):343–380, November 1990.

[4] Zena M. Ariola and Arvind. A syntactic approach to program transformations. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*, June 1991.

[5] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software: Practice and Experience*, 18:807–820, September 1988.

[6] Benjamin Goldberg. Tag-free garbage collection for strongly typed programming languages. In *SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 165–176, June 1991.

[7] Benjamin Goldberg and Michael Gloger. Polymorphic Type Reconstruction for Garbage Collection without Tags. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 53–65, 1992.

[8] Shail Aditya Gupta. An Incremental Type Inference System for the Programming Language Id. Master's thesis, MIT, Laboratory for Computer Science, September 1990. Available as Technical Report MIT/LCS/TR-488.

[9] James E. Hicks. Experiences with compiler-directed storage reclamation. In *Conference on Functional Programming Languages and Computer Architecture*, 1993.

[10] Harry G. Mairson. Deciding ML Typability is Complete for Deterministic Exponential Time. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, pages 382–401, January 1990.

[11] Rishiyur S. Nikhil. Id 90.1 reference manual. CSG Memo 284-2, MIT Laboratory for Computer Science, Cambridge, MA 02139, September 1990.

[12] G. M. Papadopoulos, G. A. Boughton, R. Greiner, and M. J. Beckerle. *T: Integrated building blocks for parallel computing. In *Proceedings of Supercomputing '93*, 1993.

[13] Simon L. PeytonJones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.

[14] Kenneth R. Traub. A compiler for the MIT Tagged-Token Dataflow Architecture. Technical Report TR-370, MIT Laboratory for Computer Science, Cambridge, MA 02139, August 1986.

[15] 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.