Accurate Garbage Collection in an Uncooperative Environment

Fergus Henderson
fjh@cs.mu.oz.au
Department of Computer Science and Software Engineering
The University of Melbourne
Victoria 3010, Australia

ABSTRACT

Previous attempts at garbage collection in uncooperative environments have generally used conservative or mostly-conservative approaches. We describe a technique for doing fully type-accurate garbage collection in an uncooperative environment, using a "shadow stack" to link structs of pointer-containing variables, together with the data or code needed to trace them. We have implemented this in the Mercury compiler, which generates C code, and present preliminary performance data on the overheads of this technique. We also show how this technique can be extended to handle multithreaded applications.

1. INTRODUCTION

New programming language implementations usually need to support a variety of different hardware architectures, because programmers demand portability. And because programmers also demand efficiency, new programming language implementations often need to eventually generate native code, whether at compile time, as in a traditional compiler, or run time, for a JIT compiler.

However, implementing compiler back-ends or JITs that generate efficient native code for a variety of different hardware architectures is a difficult and time-consuming task. Furthermore, a considerable amount of ongoing maintenance is required to generate efficient code for each new chip with different performance characteristics or a new architecture.

Because of this, few programming language implementors try to implement their own native-code generators. Instead, many programming language implementors reuse one of the existing back-end frameworks, in one of several ways: by generating another more-or-less high-level language, such as Java or C; by generating an intermediate language such as C-- [16], Java byte-code, or MSIL (the intermediate language of the .NET Common Language Runtime); or by interfacing directly with a reusable back-end, such as GCC (the GNU Compiler Collection back-end) or ML-RISC.

ISMM'02, June 20-21, 2002, Berlin, Germany.

Unfortunately, however, most of these systems, and especially most of the more mature and popular of them, do not have any direct support for garbage collection, and the ones that do, such as Java and MSIL, have their own drawbacks, such as poor performance or being available only on a restricted set of platforms.

Implementing garbage collection in systems that do not have direct support for it poses some difficult challenges; in particular, it is hard for the garbage collector to trace the system stack.

One widely-used solution to the problem of garbage collection in uncooperative environments is the approach of using conservative [4] or mostly-conservative [1, 22] collection. This approach can often deliver good performance. However, conservative collection has some drawbacks; the most significant of these is that the probabilistic nature of conservative collection makes it unsuitable for very high-reliability applications, but it also requires a small degree of cooperation from the back-end, because certain compiler optimizations are not safe in the presence of conservative collection.

Another solution which has been used when compiling to C [20] is to entirely avoid storing data on the C stack. This can be done by implementing a virtual machine, with its own stack and registers (which can be implemented as e.g. C global variables). The source language is then compiled to code which manipulates the virtual machine state; procedure calls and parameter passing are handled by explicitly manipulating the virtual machine stack and registers, rather than using C function calls.

With this approach, the collector only needs to trace the virtual machine stack, not the C stack. Since the source language compiler has full control over the virtual machine stack, tracing that stack is relatively straight-forward, and so traditional techniques for accurate garbage collection can be used. This approach also has the advantage that it can overcome some of the other drawbacks of C, e.g. the lack of support for proper tail recursion optimization [5]. However, this approach discards many of the advantages of compiling to a high-level language [13]. The source language compiler must do its own stack slot and register allocation, and the generated C code is very low-level. To make the code efficient, non-portable features must be used [14], and the resulting system is complex and fragile. Furthermore, the use of a different calling convention makes interoperability more difficult.

We propose an alternative approach that allows fully type-accurate and liveness-accurate garbage collection, thus allowing the use of a normal copying collector, without requiring any support from the back-end target language, and while still generating code that uses the normal C function calling mechanism. We describe this approach in the context of compiling to C, although it would also work equally well when interfacing directly to a compiler back-end framework, such as the GCC back-end.

Our technique is formulated as a transformation on the generated C code, which modifies the C code in such a way as to insert calls to perform garbage collection when necessary, and to provide the garbage collector with sufficient information to trace and if necessary update any pointers on the C stack. (The transformation is not *entirely* independent of the front-end language, however; it requires information from the source language front-end compiler about how to trace each local variable.)

We have implemented this technique in the Mercury compiler, and we have run a number of benchmarks of this implementation to investigate the overheads and performance of our technique.

Section 2 describes our technique, showing how the code is transformed and explaining how this avoids the difficulties with compiler optimizations that can cause problems for conservative collection. Section 3 describes how the technique can be extended to support multithreaded applications. Section 4 evaluates the performance of our technique. Section 5 discusses related work.

2. THE GC TRANSFORMATION

The basic idea is to put all local variables that might contain pointers in structs, with one struct for each stack frame, and chain these structs together as a linked list. We keep a global variable that points to the start of this chain. At GC time, we traverse the chain of structs. This allows us to accurately scan the C stack.

For each function, we generate a struct for that function. Each such struct starts with a sub-struct containing a couple of fixed fields, which allow the GC to traverse the chain:

```
struct function_name_frame {
    struct StackChain fixed_fields;
    ...
};
```

The fixed fields are as follows:

```
struct StackChain {
    struct StackChain *prev;
    void (*trace)(void *locals);
};
```

The '`prev`' field holds a link to the entry for this function's caller. The '`trace`' field is the address of a function to trace everything pointed to by this stack frame.

To ensure that the garbage collector does not try to traverse uninitialized fields, we insert code to zero-initialize any uninitialized fields of each struct before inserting it into the chain.

We need to keep a link to the topmost frame on the stack. There are two possible ways that this could be handled. One way is to pass it down as a parameter. Each function would get an extra parameter '`stack_chain`' which points to the caller's struct. An alternative approach is to just have a global variable '`stack_chain`' that points to the top of the stack.

```
extern void *stack_chain;
```

We insert extra code to set this pointer when entering and returning from functions. To make this approach thread-safe, the variable would actually need to be thread-local rather than global. This approach would probably work best if the variable is a GNU C global register variable, which would make it both efficient and thread-safe. If GNU C extensions are not available, the function parameter approach is probably best.

In our implementation in the Mercury compiler, we are currently just using a global variable, for simplicity.

2.1 Example

If we have a function

```
RetType
foo(Arg1Type arg1, Arg2Type arg2, ... )
{
    Local1Type local1;
    Local2Type local2;
    ...
    local1 = new_object( ... );
    ...
    bar(arg1, arg2, local1, &local2);
    ...
}
```

where `new_object()` is the allocation primitive that allocates garbage collected objects, and where say `Arg1Type` and `Local1Type` might contain pointers, but `Arg2Type` and `Local2Type` don't, then we would transform it as follows:

```
struct foo_frame {
    StackChain fixed_fields;
    Arg1Type arg1;
    Local1Type local1;
    ...
};

static void
foo_trace(void *frame) {
    struct foo_frame *locals = frame;
    /* code to trace locals->arg1
       and locals->local1 */
    trace_Arg1Type(&locals->arg1);
    trace_Local1Type(&locals->local1);
}

RetType
foo(Arg1Type arg1, Arg2Type arg2, ... )
{
    struct foo_frame locals;
    Local2Type local2;

    locals.fixed_fields.prev = stack_chain;
    locals.fixed_fields.trace = foo_trace;
    locals.arg1 = arg1;
    locals.local1 = NULL;
    stack_chain = &locals;
```

```
    ...
    locals.local1 = new_object( ... );
    ...
    bar(locals.arg1, arg2,
        locals.local1, &local2);
    ...
    stack_chain = ((struct StackChain *)
            stack_chain)->prev;
}
```

Here we are following Goldberg's approach to tag-free garbage collection for strongly typed languages [11]; `trace_Arg1Type()` and `trace_Local1Type()` are type-specific garbage collection routines that are generated by the front-end compiler. However, unlike Goldberg, we are primarily interested in adapting this approach to an uncooperative environment; our technique would apply equally well if using a tagged representation (in that case, only a single `trace_object()` function would be needed, instead of one for each type) or if using a table-driven approach [7] rather than a function per frame (in that case, the '`trace`' field of the '`StackChain`' struct would be a data table rather than a function pointer).

Finally, the code in the runtime system to traverse the stack frames is as follows:

```
void
traverse_stack(struct StackChain *stack_chain)
{
    while (stack_chain != NULL) {
        (*stack_chain->trace)(stack_chain);
        stack_chain = stack_chain->prev;
    }
}
```

The details of `new_object()` will depend on the particular garbage collection algorithm chosen. In our implementation, `new_object()` first checks for heap exhaustion, and then allocates memory by incrementing a global heap pointer:

```
typedef char byte;
extern byte *heap_pointer;

#define new_object(type,size) \
        ( GC_check(size), \
          heap_pointer += (size), \
          (type *) (heap_pointer - (size)) )
```

Here `GC_check()` compares the global heap pointer to another global variable which points to the end of the heap, and calls `garbage_collect()` if the heap is near exhaustion.

```
extern byte *heap_gc_threshold;

#define GC_check(size) \
        ( heap_pointer + (size) \
                >= heap_gc_threshold \
        ? garbage_collect() : (void)0 )
```

As is well known, it is possible to the reduce the overhead of checking for heap exhaustion by combining multiple checks into a single check. However, our current implementation does not perform that optimization.

The `garbage_collect()` routine in our current implementation implements a very simple copying collector [21].

```
byte *from_heap;
byte *to_heap;

void
garbage_collect(void)
{
    /* swap the "to" heap
       with the "from" heap */
    byte *tmp;
    tmp = from_heap;
    from_heap = to_heap;
    to_heap = tmp;

    /* reset the "to" heap */
    heap_pointer = to_heap;
    heap_gc_threshold =
        to_heap + HEAP_SIZE;

    /* copy the live objects
       from the "from" heap
       to the "to" heap */
    traverse_global_roots();
    traverse_stack(stack_chain);
}
```

Note that we keep a separate list of global roots, which is used for global variables that might contain pointers, in addition to the '`stack_chain`' list.

2.2 Safety

Why, you might ask, doesn't this technique suffer from the same problems with back-end compiler optimization that cause trouble with conservative collection?

The reason that this technique works is that we are not going behind the back-end compiler's back; everything is done in strictly conforming C code. Conservative collectors use non-portable techniques to trace the stack, but with our approach the collector can trace the stack just by traversing our linked list of structs. Although the code contains some pointer casts, the behaviour of these casts is defined by the C standard.

The back-end compiler cannot do any unsafe optimizations on pointer variables such as `locals.arg1` and `locals.local1`, because we've stored the address of `locals` in a global variable, and so it must assume that any call to a function whose body is not known might update the fields of `locals`.

Of course, inhibiting such optimizations in this way is a two-edged sword. The advantage is that it ensures correctness. The disadvantage is that it hurts performance.

The back-end compiler is free to perform function inlining (or outlining, for that matter); our own "shadow stack" of linked structs need not have any direct relationship with the stack frames in the underlying machine code. Furthermore, the back-end compiler can do whatever fancy optimizations it wants on non-pointer variables, such as `arg2` and `local2`, and it can cache the values of pointer variables such as `locals.arg1` and `locals.local1` in registers between function calls (if it does appropriate alias analysis). But in general it cannot cache the values of pointer variables in registers across calls to non-inlined functions. This can have a significant impact on performance.

A key contribution of this work is to *measure* the impact on performance that this transformation has.

2.3 Improvements

The scheme described above is naive in certain respects. There are several ways in which this scheme can be optimized. Rather than putting all variables that might contain pointers in the shadow-stack structs, it is sufficient to do this only for such variables which are going to be live across a point where garbage collection could occur, i.e. live across an allocation or function call.

Another optimization is to not bother allocating a struct for leaf functions that do not contain any functions calls or memory allocations.

Another possible optimization would be to use local variables to cache the fields of the shadow-stack struct, if they are referenced multiple times in a sequence of code where no garbage collection could occur, so that the C compiler could then allocate the local variables in registers.

The only one of these optimizations implemented in our current implementation is not allocating structs for functions that do not have any pointer-containing variables.

2.4 Nested scopes and liveness-accuracy

An issue that we have not yet discussed is how to handle variables declared in nested scopes within a function.

One way to handle them is to ignore the nesting and just put all pointer-containing variables in the struct, regardless of their scope. This requires first ensuring that no function contains two declarations of the same variable name with different types in different scopes (e.g. by renaming apart if needed).

This approach is the one that we have used in our current implementation. But it has two drawbacks: first, by extending the lifetime of such variables to the whole function, we may increase stack usage; and second, since the collector will scan them, this may in turn also lead to unnecessary heap retention.

The second drawback could be solved by inserting code to zero out these variables when their scope is exited, or when they are otherwise statically known to be dead. This would ensure that the collector is "liveness-accurate" with respect to the liveness of the local variables in the C code before applying our transformation. Note that if such variables were represented just as local variables in the final C code, as would be the case when using a conservative collector, it might not help to add code to assign zero to them at the end of their lifetime, since the C compiler could just "optimize" away the assignments. But with our approach, where such variables are represented as fields of the '`locals`' struct, whose address has been taken and stored in a global variable, we're safe from such unwanted optimizations.

Another way to handle nested scopes would be to use unions to ensure that the storage for pointer-containing variables declared in non-overlapping scopes is shared. However, in order for the collector to be able to trace these unions, we would need to store a discriminant for the union, which recorded which scope (if any) was active, so that the `trace()` function would know which union element to trace. Code would need to be inserted to initialize these discriminants on function entry, and to set the corresponding discriminant on entry/exit from each nested scope.

A third way to handle nested scopes would be to treat each nested scope as a separate stack frame, with the pointer-containing variables allocated in a separate struct for each nested scope. This would require adding code at the entry point(s) of each nested scope to link the corresponding struct into the stack chain list, and adding code at the exit point(s) to remove it from the chain.

For both the second and third alternatives, the collector would be to some degree liveness-accurate, in the sense that variables would definitely not be scanned after their scope has exited. However, if proper static liveness-accuracy is desired, then for any pointer-containing variables which are statically known to die before their scope has exited, additional code would need to be inserted to zero out such variables at the point of their death. (If the last use of such a variable is as a function call argument, this implies copying the variable from the stack frame struct to a local temporary, zeroing out the stack frame struct field, and then using the local temporary in the call; zeroing out the stack frame struct field *after* the call would be too late.)

3. MULTITHREADING

This system can also be extended to support multi-threading (using a "stop the world" approach to collection) in a fairly straight-forward manner, with little additional overhead.

The "stop the world" approach means that when garbage collection occurs, every mutator thread must advance to a *safe point*, and stop executing the program; when all mutator threads are stopped at a safe point, garbage collection can begin. The collector itself can be either sequential or parallel (single- or multi-threaded). The important thing about this approach is that the collector never runs in parallel with the mutator.

In our case, the "safe points" are the calls to `garbage_collect()`.

To avoid the need for synchronization at allocations, there should be a separate area of free heap space for each thread. The '`stack_chain`', '`heap_pointer`', and '`heap_gc_threshold`' variables all need to be made thread-local. When one thread runs out of free heap space, it can schedule a garbage collection by setting the '`heap_gc_threshold`' variables for every other thread to point a sentinel value (such as the start of the heap) which will cause those threads to enter the `garbage_collection()` function when they next invoke `GC_check()`. The `garbage_collection()` function can handle the necessary synchronization with other threads.

To ensure that each thread will invoke `GC_check()` within a bounded amount of time, the compiler will need to insert an additional call to `GC_check()` in the body of any long-running loops that does not do any heap allocation. Operating systems calls that can block will also need special handling (space limits prevent us from elaborating on that point here).

As mentioned earlier, '`stack_chain`' and '`heap_pointer`' can be GCC global register variables; this remains the case even with multithreading. However, '`heap_gc_threshold`' cannot be put in a register; it needs to be addressable so that it can be assigned to by other threads. In addition, because it can be modified by other threads, it needs to be declared '`volatile`'.

To summarize, the changes required to support multi-threading are these: the extra synchronization code inside `garbage_collection()`, the '`volatile`' qualifier on the '`heap_gc_threshold`' variables, the extra calls to `GC_check()` for loops that don't do any heap allocation, and special han-

Program	Lines of code	Number of iterations	Execution time (in seconds) hlc	hlc.gc	hlc.agc	Ratios hlc.gc / hlc	hlc.agc / hlc	hlc.agc / hlc.gc
cqueens	91	35000	0.70	1.29	1.28	1.84	1.83	0.99
crypt	132	20000	2.72	5.06	5.00	1.86	1.84	0.99
deriv	126	70000	0.15	0.75	0.79	5.00	5.27	1.05
nrev	56	20000	0.18	0.80	0.86	4.44	4.78	1.07
poly	259	1200	0.57	2.89	3.37	5.07	5.91	1.17
primes	78	30000	1.12	2.03	2.58	1.81	2.30	1.27
qsort	64	100000	0.83	2.68	3.11	3.23	3.75	1.16
queens	85	100	2.31	4.78	4.55	2.07	1.97	0.95
query	96	30000	3.92	3.95	4.09	1.01	1.04	1.04
tak	32	3000	2.60	2.62	2.63	1.01	1.01	1.00
Harmonic mean						1.99	2.08	1.06

Figure 1: Benchmark results

dling for OS calls that may block. While the extra synchronization code in `garbage_collection()` may be somewhat costly, collections should not be too frequent, so the amortized cost is small. Loops that don't do any heap allocation are also likely to be rare, and in most cases the cost of an extra GC check per loop iteration will be relatively small; if the loop is small, the cost of checking can be amortized over multiple loop iterations by performing loop unrolling prior to inserting the extra GC checks. Finally, the cost of the special handling needed for blocking OS calls is likely to be small in comparison to the cost of a system call. Hence we expect that overall, this technique should have little additional overhead compared with the single-threaded version.

Since the C standard does not support multithreading, it is not possible to do all this in strictly conforming C. In addition to functions for creating and synchronizing threads, such as provided by Posix threads, the approach described above also requires that assignments to the '`volatile`', thread-local pointer variables '`heap_gc_threshold`' be atomic. Technically this is not guaranteed by the C or Posix standards, but most current platforms do make assignments to appropriately aligned pointers atomic, so it should be pretty portable in practice.

4. PERFORMANCE

The benchmark machine was a Gateway Select 1200 PC, with a 1200MHz AMD Athlon CPU, 64kb L1 instruction cache, 64kb L1 data cache, 256k L2 cache, and 256Mb RAM, running Debian GNU/Linux Woody (testing), GNU libc 2.1, gcc 2.95.4, and Mercury rotd-2002-04-19. Each benchmark was compiled with '`mmc -O5 --no-reclaim-heap-on-failure --no-deforestation`'. Times shown are each the best of three successive runs, on an unloaded machine.

We used a set of small benchmarks that has previously been used in benchmarking Mercury implementations [18]. Since the time for many of these benchmarks is very small, we ran each benchmark for many iterations, and measured the total time to execute all iterations.

(Ideally, it would be better to test with larger benchmarks. But our current implementation does not yet support the full Mercury language — higher-order code and type classes are not supported because tracing of closures is not yet implemented. This makes it difficult to find large benchmark programs that work with our implementation.)

We compared three variants ("grades") of the Mercury compiler. The '`hlc`' grade allocates memory by just incrementing a heap pointer, as described in this paper, but the code does not test for heap overflow, and instead of performing garbage collection, memory is reclaimed by just resetting the heap pointer after each iteration of the benchmark. This is not a realistic memory management strategy for most real applications, but it represents a useful baseline.

The '`hlc.gc`' grade uses the Boehm (et al) conservative collector [4].

The '`hlc.agc`' uses the accurate garbage collection technique described in this paper. The collector is a simple two-space copying collector. The heap size used was 128k per space (i.e. 256k in total).

The results are shown in Figure 1.

On most of these benchmarks, the conservative collector is a little faster than the accurate collector. On some, the accurate collector is slightly faster. When averaged over all the benchmarks, the conservative collector is 6% faster. Both the conservative collector and the accurate collector are much slower than the version which just resets the heap pointer after each benchmark iteration.

Profiling indicates that for the '`hlc.agc`' grade, very little time (less than 1%) is spent in the `garbage_collection()` function. The slow-down compared to the '`hlc`' grade probably results from decreased locality and from the overhead of storing local pointer variables on the stack rather than in registers.

Given that the Boehm collector has had years of tweaking and tuning, whereas our implementation is not yet well optimized (e.g. it does not yet use GCC global register variables), we consider this to be a reasonable result. We conjecture that with additional work on optimization, this approach can achieve performance as good as the Boehm collector on most benchmarks. But the increased portability and reliability of our approach make it desirable for some uses regardless of whether there is a performance advantage one way or the other.

In our current implementation, we have not yet obtained the main benefit for logic programming languages of a copying collector, namely that copying collectors can allow cheap heap reclamation on backtracking, by just saving and restoring the heap pointer. Doing this requires some additional care in the garbage collector, to update the saved heap pointers after garbage collection, which we have not yet implemented properly.

5. RELATED WORK

The accepted wisdom of the community is that conservative collection is the only approach that works in uncooperative environments. For example, the highly experienced and respected language implementor Robert Dewar wrote "you can't do any kind of type accurate GC without information from the compiler back end" (noting that tagged architectures such as the CDC 6000 were an exception) [6]. Similarly, in the Garbage Collection FAQ [10], David Chase states that when compiling to C or C++, relocation of objects is "not generally possible" even if active pointers are registered, "because compiler-generated temporaries may also reference objects". However, this is certainly not a problem for our technique, as explained in Section 2.2, so this section of the FAQ is at best misleading.

A big part of the contribution of this paper is to dispute that accepted wisdom, by demonstrating that it is possible to implement fully type-accurate garbage collection within such an environment.

A lot of earlier work in the literature addresses issues which are quite close to the issues that we address, but using different techniques, or uses very similar techniques, but for a different purpose.

Boehm [4], Bartlett [1], and Yip [22] address the issue of uncooperative environments, but use conservative or mostly-conservative collection, rather than accurate collection. Boehm and Chase [2, 3] address the issue of safety of conservative garbage collection in the presence of compiler optimizations.

Shadow stacks have been used for debugging in the Berkeley Sather [19] implementation, and in `cdb` [12]. But these systems do not use shadow stacks for garbage collection; Berkeley Sather uses the Boehm (et al) conservative collector.

Shadow stacks have been used for hand-coded garbage collection in several systems that we are aware of, in particular Emacs, GCC, and RT++ [8, 9, 17]. In these systems, unlike ours, the code to register local variables is inserted manually, and the shadow stack is a list or array of individual variables rather than a list of frames. These systems are perhaps closest to ours, but we are interested in automating this technique as part of a programming language implementation, rather than using it to implement a garbage collection library for a language with manual memory management.

As far as we know there has been no published work comparing the performance of these hand-coded shadow stack approaches with conservative collection. And as far as we are aware, none of the published work on these addresses multithreading.

Tarditi et al [20] describe an ML to C compiler that supports accurate garbage collection. However, this compiler works by emulating a virtual machine, rather than using the normal C calling convention; this has the drawbacks mentioned in the introduction.

The Glasgow Haskell compiler [15] has accurate garbage collection and can compile via GNU C; but it relies on highly non-portable techniques that involve munging the generated assembler file after the GNU C back-end has finished with it. The success of such techniques relies on continued cooperation from the back-end compiler.

C-- [16], a portable assembly language that supports garbage collection, is another approach that relies on cooperation from the back-end compiler.

Wilson's GC survey [21] discusses a variety of different garbage collection techniques, such as mark-sweep collection and copying collection, but does not address the issue of how to locate roots on the stack.

6. CONCLUSIONS

We have presented a scheme for performing accurate garbage collection in an uncooperative environment, by a simple transformation on the code passed to the uncooperative compiler back-end framework. We have implemented this scheme in the Mercury compiler, and measured its performance, which is similar to that of the Boehm (at al) conservative collector [4] on most of our benchmarks, even though there are several important optimizations that we have not yet implemented.

We have also described how this scheme can be extended to handle multithreaded applications.

The source code for our system is freely available on the web at `<http://www.cs.mu.oz.au/mercury/download/rotd.html>`.

Acknowledgments

I would like to thank Tom Lord, for his comments on the gcc mailing list about the advantages of precise collection; Zoltan Somogyi, Andreas Rossberg, Tyson Dowd, Ralph Becket, Jon Dell'Oro and the anonymous referees for their comments on earlier drafts of this paper; and Microsoft and the Australian Research Council, for their financial support.

7. REFERENCES

[1] Joel F. Bartlett. Mostly-Copying garbage collection picks up generations and C++. Technical Note TN–12, Digital, Western Research Laboratory, 1989.

[2] Hans Boehm. Simple garbage-collector-safety. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, 1996.

[3] Hans Boehm and David Chase. A proposal for garbage-collector-safe C compilation. *Journal of C Language Translation*, 4:126–141, December 1992.

[4] Hans Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software — Practice and Experience*, 18:807–820, 1988.

[5] William D. Clinger. Proper tail recursion and space efficiency. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 174–185, 1998.

[6] Robert Dewar. Mail to the GCC mailing list (`<gcc@gcc.gnu.org>`). URL: `<http://gcc.gnu.org/ml/gcc/2001-07/msg01990.html>`, July 2001.

[7] Tyson Dowd, Zoltan Somogyi, Fergus Henderson, Thomas Conway, and David Jeffery. Run time type information in Mercury. In *Proceedings of the 1999 International Conference on the Principles and Practice of Declarative Programming*, pages 224–243, Paris, France, September 1999.

[8] Daniel R. Edelson and Ira Pohl. A copying collector for C++. In *USENIX C++ Conference Proceedings*, pages 85–102, 1991.

[9] Daniel Ross Edelson. Dynamic storage reclamation in C++. Technical Report UCSC-CRL-90-19, UCSC, June 1990.

[10] David Chase (et al?). GC FAQ (garbage collection frequently asked questions). URL: `<http://www.iecc.com/gclist/GC-faq.html>`.

[11] Benjamin Goldberg. Tag-free garbage collection for strongly typed programming languages. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 165–176, Toronto, Ontario, Canada, June 26–28, 1991.

[12] David R. Hanson and Mukund Raghavachari. A machine-independent debugger. In *Software — Practice and Experience*, volume 26, pages 1277–1299, November 1996.

[13] Fergus Henderson and Zoltan Somogyi. Compiling Mercury to high-level C code. In Nigel Horspool, editor, *Proceedings of the 2002 International Conference on Compiler Construction*, Grenoble, France, April 2002. Springer-Verlag.

[14] Fergus Henderson, Zoltan Somogyi, and Thomas Conway. Compiling logic programs to C using GNU C as a portable assembler. In *Proceedings of the ILPS '95 Postconference Workshop on Sequential Implementation Technologies for Logic Programming Languages*, Portland, Oregon, December 1995.

[15] Simon L. Peyton Jones, Cordy Hall, Kevin Hammond, Will Partian, and Phil Wadler. The Glasgow Haskell compiler: a technical overview. In *Proceedings of the UK Joint Framework for Information Technology (JFIT) Technical Conference*, Keele, 1993.

[16] Simon Peyton Jones, Norman Ramsey, and Fermin Reig. C--: a portable assembly language that supports garbage collection. In *International Conference on Principles and Practice of Declarative Programming*, September 1999. Invited paper.

[17] Wolfgang Schreiner. RT++ — higher order threads for C++, tutorial and reference manual. Technical Report 96-9, RISC-Linz, 1996. URL: `<http://www.risc.uni-linz.ac.at/software/rt++/index_131.html>`.

[18] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 1997.

[19] David Stoutamire and Matt Kennel. Sather revisited: A high-performance free alternative to C++. *Computers in Physics*, 9:519–524, Sep/Oct 1995.

[20] David Tarditi, Anurag Acharya, and Peter Lee. No assembly required: Compiling Standard ML to C. Technical Report CMU-CS-90-187, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, November 1990.

[21] Paul R. Wilson. Uniprocessor garbage collection techniques. In Yves Bekkers and Jacques Cohen, editors, *Proceedings of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, pages 16–18, St Malo, France, September 1992. Springer-Verlag.

[22] G. May Yip. Incremental, generational mostly-copying garbage collection in uncooperative environments. Technical Report 91/8, Digital, Western Research Laboratory, June 1991. Masters Thesis – MIT, Cambridge, MA, 1991.