Practical Aspects for a working Compile Time Garbage Collection System for Mercury

Nancy Mazur
Peter Ross
Gerda Janssens
Maurice Bruynooghe

Report CW310, May 2001

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Practical Aspects for a working Compile Time Garbage Collection System for Mercury

Nancy Mazur
Peter Ross
Gerda Janssens
Maurice Bruynooghe

Report CW 310, May 2001

Department of Computer Science, K.U.Leuven

Abstract

Compile-time garbage collection is still a very uncommon feature within compilers. In previous work we have developed a compile-time structure reuse system for Mercury, a logic programming language. This system derives so called reuse information. This information indicates which datastructures can safely be reused at run-time.

As preliminary experiments were promising, we have continued this work and have now a working and well performing near-to-ship compile-time garbage collection system built into the Melbourne Mercury Compiler.

In this paper we present the multiple design decisions leading to this system, we report the results of using compile time garbage collection for a set of benchmarks, including real-world programs, and finally we discuss further possible improvements. Benchmarks show substantial memory savings and a noticeable reduction in execution time.

Keywords : Program Analysis, Mercury, Compile-time Garbage Collection
CR Subject Classification : D.3.4, I.2.3

Practical Aspects for a working Compile Time Garbage Collection System for Mercury

Nancy Mazur, Peter Ross[⋆], Gerda Janssens, and Maurice Bruynooghe

Department of Computer Science, K.U.Leuven
Celestijnenlaan, 200A, B–3001 Heverlee, Belgium
{nancy,gerda,maurice}@cs.kuleuven.ac.be
petdr@miscrit.be

Abstract. Compile-time garbage collection is still a very uncommon feature within compilers. In previous work we have developed a compile-time structure reuse system for Mercury, a logic programming language. This system derives so called reuse information. This information indicates which datastructures can safely be reused at run-time.
As preliminary experiments were promising, we have continued this work and have now a working and well performing near-to-ship compile-time garbage collection system built into the Melbourne Mercury Compiler.
In this paper we present the multiple design decisions leading to this system, we report the results of using compile time garbage collection for a set of benchmarks, including real-world programs, and finally we discuss further possible improvements. Benchmarks show substantial memory savings and a noticeable reduction in execution time.

1 Introduction

Modern programming languages typically limit the possibilities of the programmer to access and manage memory directly. Allocation and deallocation is delegated to the run-time system and its garbage collector. Declarative languages go further by prohibiting destructive updates of datastructures. The price to pay is a considerable loss of performance due to the run-time overhead of garbage collection and the cost of creating new datastructures instead of updating existing ones.

Special techniques have been developed to overcome this handicap and to improve the memory usage, both for logic programming languages [4, 13, 20] as for functional languages [11, 23, 17]. Some of the approaches depend on a combination of special language constructs and analysis using unique objects [21, 1, 24], some are solely based on compiler analyses [7, 12, 18, 10], and others combine it with special memory layout techniques [23]. In this work we develop a purely analysis based memory management system.

Mercury, a modern logic programming language with declarations [21] profiles itself as a general purpose programming language for large industrial projects.

[⋆] Mission Critical, Avenue Claire, 27 - B 1410 Waterloo - Belgium

Memory requirements are therefore high. Hence we believe it is a useful research goal to develop a compile-time garbage collection (CTGC) system for this language. In addition, mastering it for Mercury should be a useful stepping stone for systems such as Ciao Prolog [9] (which has optional declarations and includes the impurities of Prolog) and HAL [5] (a Mercury-based constraint language). The intention of the CTGC system is to discover at compile-time when data is not referenced to anymore, and could be reused for new data.

Mulkers et al. [20] have developed an analysis for Prolog which detects when memory cells become available for reuse. In [2] this analysis was adapted for logic languages with declarations. As Mercury supports programming in the large through the use of modules, the analysis was further refined for modular logic languages with declarations [16]. A first prototype implementation was made to measure the potential of the analysis for detecting dead memory cells.

As the results of the prototype were promising, we have continued this work and implemented a full CTGC-system for the Melbourne Mercury Compiler (MMC). Once a program is annotated with the derived reuse information, the CTGC-system has to decide which reuses to actually perform, and how. In this paper we present different possible decisions that can be made and that we have implemented. A series of benchmarks are given. These benchmarks measure not only the global effect of CTGC, but also the effect of the different decisions.

In this work we mainly focus on techniques of reuse-selection that were easy to implement in the existing compiler. More theoretical approaches are described in [7], where the concept of reuse maps is used for representing the one-one mappings between dead cells and their reuses, and in [4] where a general formulation of the optimization problem at hand is given.

After presenting the necessary background terminology in Section 2, we describe the overall structure of the CTGC-system in Section 3. Section 4 presents the different reuse decisions that were made in orde to obtain a first working reuse system. In order to increase the performance, low level additions were made as described in Section 5. The results for the benchmarks are reported in Section 6. Using a cell cache (Section 7) even better results are obtained. Finally some further improvements are suggested (Section 8) and a conclusion formulated (Section 9).

2 Preliminaries

2.1 Mercury

Mercury [8] is a logic programming language with types, modes and determinism declarations. Its type system is based on a polymorphic many-sorted logic and its mode-system does not allow the use of partially instantiated datastructures.

The analysis performed by our CTGC-system is at the level of the *High Level Data Structure* (HLDS) constructed by the MMC. Within this structure, predicates are *normalized*, i.e. all atoms appearing in the program have distinct variables as arguments, and all unifications $X = Y$ are explicited as one of (1) a test $X == Y$, (2) an assignment $X := Y$, (3) a construction $X \Leftarrow f(Y_1, \ldots, Y_n)$, or (4) a deconstruction $X \Rightarrow f(Y_1, \ldots, Y_n)$ [8]. Within the HLDS, the atoms of a clause body are ordered such that the body is well moded.

Just like in the HLDS we will use the notion of a *procedure*, i.e. a combination of one predicate with *one* mode, and thus talk about the analysis of a procedure.

2.2 Types, selectors, datastructures and aliases

Using a simple example, we recall some basics about types. The polymorphic type `list(T)` is defined as: `list(T)--->[] ; [T|list(T)]`. The right hand side of this expression defines the different *type functors* of `list(T)`. A *type tree* is a possibly infinite graphical representation of a type definition. A finite *type graph* is obtained by imposing that two type nodes on the same branch from the root are the same when they are labelled with the same type. The type `list(T)` has two type nodes (`list(T)` and `T`) and two functor nodes (`[]` and `[.]`).

Selectors are used to select type nodes in type graphs. The empty selector is denoted by ϵ, it selects the root node of the type graph of the type to which it belongs. In general a selector is a tuple (f, i). It selects the i^{th} child of the functor node labeled f. Selectors can be combined in such a way that they form a legal path down a type graph. With recursive types, several selectors can be equivalent – selecting the same type node – and the shortest one can be used as a representative of the equivalence class.

Terms also have a tree representation and nodes of the term tree are mapped to the corresponding type tree. With X a variable of type t and s a selector for t, X^s denotes the nodes in the term tree which are mapped to t^s. We refer to the memory cells implementing these nodes as the *datastructure* or *data cell* of X^s. X^ϵ selects the root node of the term tree.

Part of the CTGC-system consists of deriving alias information [2]: in order to decide whether a datastructure d can be reused at some point, we have to be sure that there are no datastructures which are still needed during further execution of the program and which share memory with d. Alias information is expressed as a tuple (X^{s_x}, Y^{s_y}) with X^{s_x} and Y^{s_y} identifying type nodes (of the same type) in the type graphs of resp. X (of type t_X) and Y (of type t_Y). Its meaning is that the term trees for X and Y might share memory at nodes mapped to resp. $t_X^{s_X}$ and $t_X^{s_Y}$.

2.3 Reuse information

In [16] we developed a reuse analysis which derives which datastructure might become *available for reuse* at some program-point within a procedure. The datastructure is then said to be *dead.* This is detected during a so called *default analysis* which assumes that no aliases exist between the inputs of a procedure when called, and only the outputs will be accessed after returning from the call.

If a suitable candidate for reusing the available dead datastructure is encountered we say that *direct reuse* is possible. In some cases this reuse can be independent of the calling environment of the procedure. This is a case of *unconditional direct reuse.* Yet, in general, whether a datastructure can be reused or not will depend on the caller. We express this as a *condition* that has to be met by the calling environment of the procedure in order for the reuse to be safe. This is *conditional direct reuse.*

The reuse analysis is not limited to detecting direct reuse. Part of it's responsibility is to verify whether a call to a procedure can be replaced by a call to the reuse-equivalent of the procedure. This is done by verifying the reuse conditions. Consider a procedure p, with reuse-version p^r, which is called within the definition of the body of a procedure q. In analogy to direct reuse, calling the reuse-version p^r of p might be independent of the calling environment of q. This is *unconditional indirect reuse.* If it is not independent, then again, conditions can be derived which express the circumstances in which this substitution is safe. This is called *conditional indirect reuse.*

For each analysed module, an interface-file can be generated which records whether some of its exported predicates allow reuse, and under what conditions. This interface-file is used during the analysis of modules depending on that module. For more details we refer to [16].

2.4 Low level representation of typed terms

The CTGC-system aims at reusing datastructures. In the MMC, these datastructurese are represented using different representations for the terms of each type. Each term is represented by either a single machine word or a tagged pointer to an object on the heap. We only reuse terms of the latter kind.

We illustrate the representation with the following types:

```
:- type dir ---> north ; south ; east ; west.
:- type example ---> a(int, dir) ; b(example).
```

Primitive types such as integers, chars, floats[1] and pointers to strings are represented as single machine words. Types such as `dir`, in which every alternative is a constant are equivalent to enumerated types in other languages. Mercury represents them as small consecutive integers starting from zero. This representation is stored directly in a machine word. Terms of types such as `example` are stored on the heap. The pointer pointing to the actual term is tagged [6]. This (primary) tag allows the distinction between different function symbols of a type. Terms of types having more function symbols than a primary tag can distinguish use secondary tags.

Figure 1 illustrates the internal representation of some terms after the construction unifications: `X <= a(3, east), Y <= b(X)`. Now suppose that it is known that a datastructure such as `X` becomes available for reuse, and that the deconstruction in which this is decided is followed by a construction `Z <= a(12, west)`. The memory words pointed to by `X` can be reused for constructing `Z`: 3

[1] Depending on the word-size, these might have a boxed representation.

will be set to 12, and 2 (east) to 3 (west). The pointer (at `X`) can be reused too, but its effect is not as important as reusing structures on the heap[2].

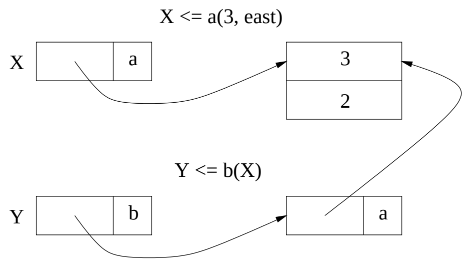

Fig. 1. Representation of the memory after `X <= a(3, east), Y <= b(X)`.

3 General structure of the CTGC-system

The CTGC-system consists of three parts (see Figure 2): first annotating the code, then deciding which reuses to perform and finally generating the low-level code which does the actual reuse. The input consists of a mode-, type- and determinism-correct HLDS-representation of a Mercury program.

Part one of the CTGC-system consists of two major annotation-phases based on program analysis. We first annotate each procedure with the possible aliasing between the head variables. The aliasing is derived in a *goal-independent analysis* and requires a fixpoint computation to cope with recursive calls. Each module is analysed separately. The analysis relies on alias-information available in interface-files to deal with procedures defined in other modules than the one being analysed. The second annotation step derives liveness information. Each deconstructed datastructure is marked as either available for reuse or not. It also records the condition for which the reuse is safe. The liveness annotations are derived by the *default liveness analysis.* No fixpoint-computation is needed. For more details about these analyses we refer to previous work [2, 14–16].

The second part of the CTGC-system contains all the passes which decide which reuses will be allowed. The reuse decisions occur at two levels. First decide which construction unifications in a procedure should reuse available dead cells (direct reuse). Once the direct reuses are sorted out, and accompanying reuse conditions are expressed, a pass is needed to verify and decide which procedure calls can be replaced by their reuse counterparts (indirect reuse). This pass requires a fixpoint computation as allowing indirect reuse can in itself introduce new reuse conditions. Finally each procedure is split into different versions: a

[2] Reusing the pointer may require that the pointer be kept on the stack, increasing stack usage in the program. Currently we do reuse the pointer as well.

basic version having no reuse (or only the unconditional ones), and versions with conditional reuse. While the underlying concepts were already developed in [16], the pragmatics of our implementation are discussed in the next section.

Finally, the last part generates the low-level code corresponding to the reuses decided in the previous pass. This pass consists of generating the apropriate backend-code (e.g. C) which performs the discovered reuse. We will not go into the low level details of this modification to the Mercury compiler.

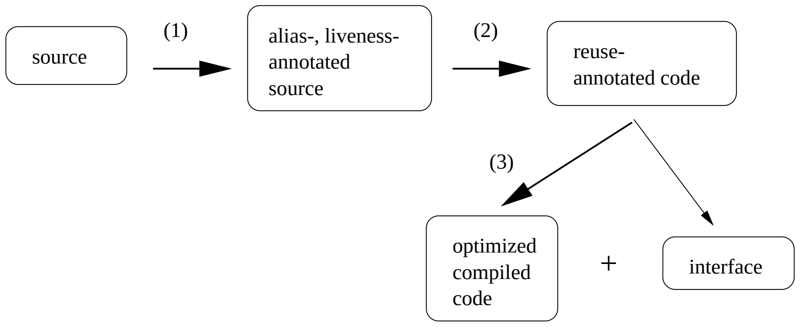

Fig. 2. Structure of the CTGC-system. Annotating the code (1), deciding the reuses (2), and finally, generating the optimised compiled code (3) and extra interface-files.

4 A working reuse decision approach

Consider a predicate which converts a given list of data-elements to another list of data-elements as shown in Figure 3. After the annotation-pass the procedure will be said to create no new aliases between the arguments (integers being an unsharable primitive type), and the datastructures at each of the deconstructions (`List0` and `Field1`) will be marked as conditionally dead (i.e. if reused, it will be a conditional direct reuse). The purpose of this pass is to select the reuses that yield the most interesting saving w.r.t. memory usage and execution time.

4.1 Deciding direct reuse

A first restriction we imposed from the very beginning was to limit reuses to local reuses, i.e. a dead cell can only be reused within the same procedure. It cannot be reused within another procedure. In order to allow otherwise could complicate this decision phase considerably (see also Section 7).

Another restriction we assume is that when executing a construction statement, at most one dead datastructure is reused (not a combination of different dead structures), and that one dead structure is reused by at most one construction. This assumption was also taken for simplicity's sake.

```
:- type field1 ---> field1(int, int, int).
:- type field2 ---> field2(int, int).
:- type list(T) ---> [] ; [T | list(T)].
:- pred convert(list(field1), list(field2)).
:- mode convert(in, out) is det.

convert(List0, List):-
    ( % switch on List0
        List0 => [],
        List  <= []
    ;
        List0 => [Field1 | Rest0],      % (d1)
        Field1 => field1(A, B, _C),     % (d2)
        Field2 <= field2(A, B),         % (c1)
        convert(Rest0, Rest),
        List <= [Field2 | Rest]         % (c2)
    ).
```

Fig. 3. Annotated Mercury code for converting a list of `field1`-elements to a list of `field2`-elements. In this code, the modes of the unifications are made explicit.

The default liveness analysis of the example identifies the deconstructed datastructures (at `d1`, resp. `d2`) as available for reuse. The procedure also contains two constructions (`c1` and `c2`) requiring the allocation of memory. Each of them can potentially reuse the available dead datastructures.

Each of the combinations yields an acceptable reuse-scheme. Yet, which one yields the most interesting memory reuse? It has been shown that this problem [4] can be reformulated as an instance of the maximum weight matching problem for a weighted bipartite graph. However for simplicity of implementation we have reduced this general matching problem to two orthogonal decisions: imposing constraints on the allowed reuses, and using simple strategies to select amongst different candidates for reuse. We will discuss each of these.

Constraints on allowed reuses. Constraints allow to express desirable properties between the dead cell and the newly constructed cell. Here we have used constraints on the differences in arity or constructor that we allow between these cells. For example, reusing a cell of arity 15 for a cell of arity 2 might not be desirable if the garbage collector is not able to recover the 13 remaining memory-words. Or if changing the type of a data cell is impossible[3], than we can only allow reuses for matching constructors.

These are the constraints we have implemented:

– Matching constructors. Only allow reuse of a dead cell which has exactly the same constructor as the new cell. In the example of figure 3 this means that `c1` cannot reuse anything, and `c2` can reuse the available memory of `d1`.

[3] This is the case when using .NET as the backend for Mercury.

- Matching arities. This is more flexible as it allows reuse even if the constructors are different, yet in the example no extra reuses would be allowed.
- A limit on the difference between the arities. This constraint expresses the intuition that it might be interesting to reuse a dead cell, even if some memory-words cannot be reused. In our example, allowing a difference of size one would already make the memory of `d2` available for reuse in the constructions `c1` or `c2`.

Selection strategies The above constraints are insufficient to obtain a clear one-to-one match between deconstructed cells available for reuse, and constructions where they can be reused (e.g. `c1` could reuse either the cell available from `d1` or `d2`) We have experimented with two simple strategies:

- lifo. Traverse the body of the procedure and assign the reuses using a last-in-first-out selection strategy. This means that when a choice is left for a given construction, always choose the dead cell which has been deconstructed most recently. The intuition here is that after deconstructing a variable, chances are that the first structure one constructs will be fairly similar. This similarity can have a positive effect on the number of data-fields that have to be updated when reusing the dead cell.
  e.g. If `c1` is allowed to reuse the cells of `d1` or `d2`, then according to this strategy, `Field1` will be reused for constructing `Field2` and `List0` for `List`. In this example, this corresponds to the best choice one can make as the reuse of `Field1` corresponds to a simple tag change on the pointer[4].
- random. The intuition behind the lifo-strategy might not always be true. Therefore we have added a simple second strategy which randomly selects the dead cell amongst all the candidates.

With matching size between dead and reused memory, the selection strategy only influences the execution time; when allowing size differences, also the total amount of reuse can be affected.

The configuration combining the constraint of matching arities with lifo as selection strategy is called the *default configuration*, or *default CTGC*.

4.2 Deciding indirect reuse

In order to decide whether a call to a procedure can be substituted by a call to a reuse version of that procedure, we must be sure that such substitution is safe. This is tested by checking the reuse-conditions (under the assumption of a default call pattern). If it is safe to call the reuse-version we have to decide whether we will do so or not.

Here we have decided for simplicity by always choosing the reuse-version of a procedure if it is safe to do so. In Section 8 we discuss the drawbacks and suggest a possible better solution.

[4] All the positions of the new cell have the same value as the corresponding positions of the reused cell.

In our example, at least one local direct reuse is detected and a reuse condition can be expressed. Under the assumption of a default call pattern, it can be shown that the recursive call satisfies the reuse conditions. Hence, the call to `convert` within a reuse version of this procedure can be substituted by a recursive call to this reuse version.

4.3 Splitting into different versions

Once the possible direct and indirect reuses have been decided, there is one remaining decision left: how many versions of a given procedure should be created? In our example, we might have detected three reuses: we might decide that `List0` should be reused by `List`, `Field1` by `Field2`, and that the recursive call can be replaced by the reuse version. Given these decisions, we can generate 4 interesting versions of the initial procedure: a version without any reuse, a version reusing only `List0` (including the recursive reuse call), a version reusing `Field1` (and recursive reuse call), and a version reusing both. In general, for a procedure with n possible direct reuses, 2^n interesting versions can be created. In our current implementation we have limited the number of versions we generate to at most two: one version which imposes no conditions on the caller (containing all possible unconditional reuses), and one version containing all possible reuses that have been detected.

5 Low level additions

Given the previous decisions and strategies, a first working CTGC-system using the three phases was implemented. Although good results were obtained for small programs (e.g. naive-reverse), the quality of the results and general behaviour of the CTGC-system dramatically worsened on more real-life examples. The reasons for this change in behaviour were:

- Rapid decrease in precision of the aliasing-information, hence more possibilities of sharing data and thus less cells detected for reuse.
- The number of aliases collected within a procedure can become huge. This makes the operations manipulating them slow and the CTGC process becomes too time consuming.

5.1 Enhancing the aliasing precision

The underlying analysis for deriving alias-information uses the concept of `top` which expresses that all data parts might be aliased. This is a safe abstraction in case of total lack of knowledge about the possible existing aliases at some program point. Once generated, this lack of information will propagate rapidly as all primitive operations manipulating it will yield `top` as well.

Such `top` is generated in the presence of language constructions with which the analysis cannot cope yet. These are procedures defined in terms of foreign code (c, C++), higher-order calls and typeclasses. It is also generated for procedures which are defined in other modules that have not yet been analysed and for which no interface files have been generated yet.

To obtain a usable CTGC-system, techniques had to be found to limit the creation and propagation of this lack of information. In our current implementation, three techniques are used:

1. *Using heuristics.* Based on the type- and mode- declaration of a procedure, one can derive whether it can create additional aliases or not, without looking at the procedure's body. This is the case when the procedure deals with unique objects (declared `di` or `uo` [8]), or has no non-unique output variables[5] or when the non-unique output arguments are of a type for which sharing is not possible (integers, enums, chars, etc.). In all these cases, it is safe to conclude that the procedure will not introduce new aliases. These heuristics are applied within the first phase of the system (for generating precise aliasing information), as well as in the second phase where the aliases are used for verifying the reuse conditions.
2. *Manual aliasing annotation for foreign code.* Important parts of the Mercury standard library consist of procedures which are defined in terms of foreign code. Hence the effect of `top` would be big. With the intention to be used mainly within this standard library, we have extended the Mercury language such that foreign code can be manually annotated with aliasing-information. Such annotations are interpreted by the compiler as a promise which the user makes about the foreign-code and are not verified by the compiler.
3. *Manual iteration for mutual dependent modules.* The current compilation-scheme of Mercury is not yet able to cope with mutual dependent modules. Consider a module A in which some procedures are expressed in terms of procedures declared in a module B, and vice versa. The normal compilation scheme is to compile one of the files, and then the other one. In the presence of an optimizing compiler this is not enough. At the moment the first module is compiled, nothing is known from the second one, yielding bad precision for the first one. This bad precision will propagate further to the second file as the second file relies on the first one. Bueno et al. [3] propose a new compilation scheme which is able to handle these cases. As this requires quite some work, we make a work around by allowing manually controlled incremental compilation.

5.2 Making compilation faster: widening the aliasing

While it is interesting to have more precise aliasing information than simply `top`, introducing more aliases slows down the system. It is not unusual to deal with thousands of possible aliases at some program points. Now one can argue that speed is not a major requirement of a CTGC-system as it is not intended

[5] A procedure call cannot create additional aliases between input variables which must be ground at the moment when the procedure is called.

to be used at each compilation, only at the final compilation. But even for our benchmarks we were not ready to wait hours for one single module to compile. Therefore, in order to produce a usable CTGC-system we have added a special widening operator which acts onto the aliases produced. This widening operator can be enabled on a per-module base, such that widening is applied to only a specific subset of the modules dealt with. The user can also specify the threshold at which widening should be performed: e.g. only widen the set of aliases if the size of this set exceeds 1000.

The widening we use consists of replacing a full path of normal selectors by one selector, a so called *type selector*. Such a selector consists of a type instead of a precise tuple containing a type functor and an index. The meaning of a type selector t applied to a variable V is as follows: V^t denotes all the datastructures within the type tree of the term of variable V which are of type t. We call this operation *type widening*. The following example illustrates the effect of type widening.

Example 1. Consider the following type-definition:

```
:- type t1(T) ---> empty; cons1(T);
                   cons2(t2(T)); cons3(T, t1(T)).
:- type t2(T) ---> cons4(T, T).
```

Consider a procedure creating all possible kinds of aliases between an input of type `t1`, say V_{in} and an output of type `t1`, say V_{out}. Using the full standard selectors, a large alias-set would be generated. If recursive types are always simplified (the selector $(V_{in})^{s(cons3,2)}$ would be mapped to V_{in}^{ϵ}), then at most 4^2 aliases are created. If other variables are involved too, this will have a great impact on the total number of aliases. Moreover, during the analysis of the body of a procedure, recursive types are not immediately folded (for reasons of precision), and thus an even larger set of aliases can be produced. The result of widening the alias-set will yield only one alias: $(V_{in}^{type:T}, V_{out}^{type:T})$, expressing that nodes of type `T` of V_{in} might be aliased to nodes of type `T` of V_{out}.

This widening leads to a considerable speed-up of the CTGC-system (compilation of some modules taking almost an hour was now reduced to less than 5 minutes), yet has the advantage of not loosing too much of the overall precision. The expected reuses could still be detected for our benchmarks.

6 First results

We have measured the effectiveness of our CTGC-system on some small toy benchmarks as well as one major real-life program. All the experiments were run on an Intel-Pentium III (600Mhz) with 256MB RAM, using Debian Linux 2.3.99, under a usual workload. Version 0.9.1 of the Mercury compiler was used as a basis to incorporate the CTGC-system into it. The reported memory information is obtained using the Mercury memory profiler provided by the compiler. This profiler computes the sum of all memory allocations on the heap.

The small benchmarks are *nrev* (naive reverse of a large list of integers, this operation is repeated 100000 times for a list of 30 integers) and *qsort* (quick sort applied on a list of 30 integers, repeated 100000 times). We also include the *argo_cnters* program (also used in [16]), a benchmark counting various properties in a file. Figure 1 shows the obtained results. All the CTGC-compiled versions were compiled with default configuration.

module	No Reuse			Reuse			
	C (sec)	M (kWord)	R (sec)	C (sec)	M (kWord)	rel (%)	R (sec)
nrev	1.49	99001.20	6.87	14.36	6000.27	-94	1.76
qsort	2.05	65000.13	6.15	16.16	10000.13	-84	2.45
argo_cnters	8.41	3241.61	1.45	38.27	2187.17	-32	1.41

Table 1. Results for the small benchmarks. C is the total compilation time (in seconds). M represents the amount of memory used by the program when run (in kilo Words). R is the time needed to run the program (in seconds). The column labeled "rel" shows the relative reduction in memory usage.

Within *nrev* the CTGC-system is able to recover every list cell deconstructed. The partitioning procedure used within *qsort* does not need to allocate any new memory as everything can be reused locally. For the *argo_cnters* benchmarks, reuse is also performed succesfully: the datastructure recording the properties of the file being updated in place. The difference in timing for this benchmark is statistically insignificant as most of the execution time is due to I/O.

Next to small benchmarks, we found it important to evaluate the system on a large real-life program. The large real-life program we used is a ray tracer program developed for the ICFP'2000 programming contest [19] where it ended up fourth. This program transforms a given scene description into a rendered image. It is a quite CPU- and memory-intensive process, and therefore an ideal candidate for our CTGC-system to be tested on. A complete description of this program can be found at [22].

The program consists of 20 modules (total of 5700 LOC). All modules could be compiled without widening, except for one: *peephole*. This module manipulates large instruction sets and generates up to 11K aliases. Without type-widening, the compilation of *peephole* takes 160 minutes. With type-widening (at 500 aliases), it only takes 40 seconds. The compilation of the program with CTGC (and widening) takes 5 minutes, compared to 1 minute for a normal compilation. As some of these modules depend on each other, the technique of manually iterating the compilation was used to obtain better results. For this benchmark, the compilation had to be repeated 3 times to reach a fixpoint (for a total time of 15 minutes). Each time every module was recompiled. In a smart compilation environment, most of the recompilations could be avoided.

To measure the effects of the different constraints and strategies we have compiled the ray tracer using different CTGC-configurations. As the program relies on the Mercury standard library, we have repeated the experiments with and without CTGC in the library.

Table 2 shows the memory-usage and timings of the different versions of the ray tracer for different scene descriptions. Absolute values are given for the ray tracer without any CTGC. Relative values are given w.r.t. these absolute values for the CTGC-configurations. Subscript m is used for memory usage, subscript t for execution time. The CTGC-configurations use a version of the standard library with default CTGC, unless explicitly stated otherwise.

input	nr_m	nr_t	1_m	1_t	2_m	2_t	3_m	3_t	4_m	4_t
	(kWord)	(sec)	%	%	%	%	%	%	%	%
ch-cylinder	7695.82	1.61	-10.55	-3.73	-9.52	-3.11	-8.15	-4.35	-13.77	-6.83
cylinder	24502.26	6.88	-25.61	-1.60	-24.82	-7.27	-23.59	-4.36	-29.24	-6.54
dice	537487.63	209.96	-28.47	-9.61	-28.43	-4.55	-27.81	-9.35	-4.80	11.31
fib	40276.59	11.85	-29.64	-6.41	-29.61	-5.49	-28.81	-6.08	-7.90	10.38
golf	42043.25	12.38	-9.29	-5.57	-9.20	-3.23	-8.11	-6.06	-13.70	-6.38
mtest10	27152.04	7.81	-22.77	-12.93	-21.48	-10.12	-20.90	-12.68	-18.58	-7.17
mtest11	24714.49	7.07	-7.42	-6.08	-7.14	-2.97	-6.06	-6.93	-15.95	-9.76
mtest4	7249.93	1.92	-24.37	-5.21	-24.36	-3.65	-21.13	-7.29	-16.10	2.08
mtest5	9166.16	2.36	-24.21	-6.78	-24.20	-3.39	-20.84	-6.36	-17.51	0.85
mtest6	11473.07	3.26	-25.56	-10.12	-25.55	-8.28	-22.90	-9.20	-14.18	0.00
mtest7	121501.36	34.78	-19.91	-10.47	-19.90	-8.42	-17.04	-10.26	-17.01	-6.33
mtest9	5814.33	1.53	-25.95	-5.88	-25.93	-5.88	-22.90	-8.50	-16.54	0.00
munion	5515.38	1.57	-19.57	-5.10	-19.36	-5.73	-16.70	-5.73	-17.69	-1.91
reflect	40264.38	10.25	-19.13	-2.63	-19.13	-0.49	-15.73	-0.20	-15.34	1.56
reflect2	41105.24	10.54	-19.11	-2.85	-19.11	0.47	-15.71	-3.32	-15.37	1.52
spheres	13609.06	3.55	-21.73	-3.66	-21.72	-3.10	-19.65	-5.92	-13.39	-0.28
spheres2	14104.01	3.70	-21.22	-3.24	-21.22	-4.32	-18.96	-4.59	-13.52	0.27
spotlight	15636.29	4.72	-30.22	-12.92	-30.21	-10.81	-28.74	-13.98	-30.22	-14.41
total	1022670.54	344.56	-27.11	-11.00	-27.01	-7.34	-25.72	-10.90	-13.03	3.20

Table 2. Results for the ICFP-ray tracer program comparing the memory usage and execution time to the plain non-optimised ray tracer (nr_m and nr_t).

input: Name of the scene description used as input to the ray tracer.
nr: Absolute memory usage (in kilo words) and execution time of the ray tracer without any CTGC.
1: Raytracer with default CTGC-configuration.
2: As 1, but using the standard library, hence no reuse in library predicates.
3: Reuse only for matching constructors (lifo strategy) in ray tracer.
4: Reuse allowed for size difference up to 2 (lifo strategy) in ray tracer.

Finally, the last row indicates the total amount of either absolute or relative memory usage and execution time.

Discussion of the results:

- Using the default CTGC configuration (1), up to 27% memory can be saved globally. For some individual scene descriptions, this can go up to 30%. There is also a noticeable gain on the execution time.
- Whether the Mercury standard libraries are compiled with CTGC (1) or without (2) has hardly any effect on the memory usage for the ray tracer. This can be explained by the fact that the ray tracer makes a limited use of these libraries. The difference in timings with (1) is not significant enough to draw any conclusions as this can be due to caching and other factors.
- CTGC under the constraint of only reusing matching constructors (3) gives a slightly worse memory saving.
- CTGC allowing reuse for constructors with arities within a distance of two (4) generates the worst results: the amount of memory that can be saved is less than for (1), but even worse is the execution time which is bigger than for the ray tracer without CTGC. The decrease in memory reuse is due here to an inappropriate selection strategy (lifo). Furthermore, the bad timings can be explained by the fact that with non-matching arities, reuse will leave space-leaks which cannot immediately be detected by the run-time garbage collector, with the effect that the garbage collector will be called more often.

We have also experimented with configurations using random as a selection rule. Combined with the constraints of (1) and (3), results similar to their lifo counterparts are obtained. The difference in amount of memory reused manifests itself mostly in the presence of disjunctions within the definition of the procedure[6]. Redoing (4) with random selection strategy improves the results of (4), going up to an average of 22% memory saving.

Finally, a default version of the raytracer was built without type-widening. Compared to the default version presented in Table 2 the overall memory usage difference is less than 1%. The execution times are comparable.

7 Non-local reuse: cell cache

Currently we have assumed that all dying datastructures must be reused locally, i.e. within the same procedure in which they die. This means that situations where a datastructure which dies in some procedure p cannot be reused within another procedure, say q, hence missing quite interesting possibilities of reuse:

```
p(..) :- ..., X => f(Y, Z), ...
q(..) :- ..., T <= f(A, B), ...
r(..) :- ..., p(..), q(..), ...
```

[6] e.g. `X => f(..), ( ... Y <= f(..) ; ... ), Z <= f(..)`: as the first branch of the disjunction might not always be executed, it is more interesting to allow `Z` to reuse `X` than `Y`.

There are three ways to achieve non-local reuses as well.

The first and the most difficult is to extend the analysis (i.e. part 1 of the CTGC) to handle non-local reuse. The analysis would have to propagate possible dead cells and thus become quite complex. It would also require intensive changes in the internal calling convention of procedures within the MMC as the address of the cells to be reused would have to be passed between procedures. The second approach is to combine reuse analysis with inlining in such a way that the cell death and subsequent reuse end up in the same procedure. The third approach, which is the one we implemented, is to *cache dead cells*. Whenever a cell dies *independently* of the exact call pattern of the procedure, and cannot be reused locally, we mark it as *cacheable*. At runtime the address of the cell as well as its size will be recorded in a cache. Before each memory allocation the runtime system will first check the cell cache to see if a cell of the correct size is available and use that cell instead of allocating a completely new cell. This operation is not less expensive as such, rather on the contrary. But by avoiding new allocations the overall cost of the runtime garbage collection system should go down due to smaller heap sizes, and less frequent need of garbage collection.

Table 3 compares the performance of the ray tracer using the default CTGC-configuration combined with the cell-cache technique, with the basic no-reuse version. Some scene descriptions allow to save up to 70% of memory usage and up to 12% gain of the execution time. Overal memory reuse increases from approx. 30% to approx. 50%, accompanied with a small speed-up.

8 Further improvements

In the near future, we intend to explore a number of improvements to our system. First, for some procedures, several reuse conditions are discovered, generating restrictive conditions for all reuse possibilities together. This turns out to be too ambitious with the effect that none of the reuse in it will ever be performed. A *top-down call-dependent version splitting pass* could aid in generating more useful reuse-versions of procedures, yet avoid a full code explosion when generating all the possible reuse versions.

A second problem is the too absorbant effect of the notion of `top` currently used in the alias information. Once `top` is encountered, it will propagate all throughout the remainder of the code. A certain increase in precision could be obtained by additionally recording the variables between which the possible aliases are unknown. So if a call to a predicate $p(X, Y)$ is encountered for which the aliases are unknown, we can conclude that there might be some aliases between X and Y (denoted as the set $\{X, Y\}$), instead of simply saying `top`. The effect is that the loss of information will only propagate itself through those variables. The alias information concerning variables that are not possibly aliased to the so called *top-variables* will remain unaffected, hence a possible global increase in precision. In combination with type-selectors, and knowing that sharing is only allowed between datastructers of the same type, specialised sets could be generated for each of the subtypes involved.

input	nr_m (kWord)	nr_t (sec)	cc_m %	cc_t %
ch-cylinder	7695.82	1.61	-18.62	1.86
cylinder	24502.26	6.88	-35.36	-2.18
dice	537487.63	209.96	**-57.60**	-12.96
fib	40276.59	11.85	**-72.66**	-9.54
golf	42043.25	12.38	-24.59	5.17
mtest10	27152.04	7.81	-43.72	-11.91
mtest11	24714.49	7.07	-23.05	-3.54
mtest4	7249.93	1.92	-43.01	-5.21
mtest5	9166.16	2.36	-42.72	-5.51
mtest6	11473.07	3.26	**-51.31**	-4.60
mtest7	121501.36	34.78	**-47.77**	-10.87
mtest9	5814.33	1.53	-44.21	-4.58
munion	5515.38	1.57	-36.84	-3.82
reflect	40264.38	10.25	-39.93	-1.07
reflect2	41105.24	10.54	-39.96	-1.04
spheres	13609.06	3.55	-40.35	-3.66
spheres2	14104.01	3.70	-39.45	-3.24
spotlight	15636.29	4.72	-33.41	-12.08
total	1022670.54	344.56	**-52.27**	-12.53

Table 3. Results for the ICFP-ray tracer program with cell-caching (*cc*) compared to the normal non-optimized version of the ray tracer (*nr*).

Third, we will experiment with other selection strategies: selection strategies which take into account the cost of updating datastructures (and thus preferring those datastructures where the least fields have to be updated), strategies which in the case of differing arities will first allocate the larger dead cells to the largest new cells (so as to be sure not to waste large data cells on small new cells), and other more sophisticated strategies.

9 Conclusion

This paper describes a complete working compile-time garbage collection system for Mercury, a logic programming language with declarations. The system consists of three passes: program annotation, reuse decision, and finally low level code generation. The program annotation pass consists of aliasing and liveness annotations based on previous work [16]. Different restrictions, constraints and strategies for selecting reuses were presented. In order to obtain an implementation, low level improvements were introduced.

A major contribution of this work is the integration of the CTGC system in the Melbourne Mercury Compiler and its evaluation. Some small benchmarks were used, but also one real-life complex program, a ray tracer. Average global memory savings of up to 50% were obtained, while execution was reduced up to 12%. It would be interesting to compare these results with the total potential of reuse within the program. This total potential could be approximated using the techniques used in our first prototype [16] to predict the amount of reuse.

Beside relatively small improvements proposed in Section 8, the system still needs to be extended such that language constructs such as higher order code and type classes are handled properly. Currently the aliasing generated for both are `top` and no reuse versions of procedures can be used in such calls. This requires major changes in the underlying aliasing and liveness analysis systems.

References

1. Yves Bekkers and Paul Tarau. Monadic constructs for logic programming. In John Lloyd, editor, *Proceedings of the International Symposium on Logic Programming*, pages 51–65, Cambridge, December 4–7 1995. MIT Press.
2. Maurice Bruynooghe, Gerda Janssens, and Andreas Kågedal. Live-structure analysis for logic programming languages with declarations. In L. Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming (ICLP'97)*, pages 33–47, Leuven, Belgium, 1997. MIT Press.
3. Francisco Bueno, Maria García de la Banda, Manuel Hermenegildo, Kim Marriott, Germán Puebla, and Peter J. Stuckey. A model for inter-module analysis and optimizing compilation. In *Tenth International Workshop on Logic-based Program Synthesis and Transformation*, London, UK, 2000. to appear.
4. Saumya K. Debray. On copy avoidance in single assignment languages. In David S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 393–407, Budapest, Hungary, 1993. The MIT Press.
5. Bart Demoen, Maria J. Garcia de la Banda, Warwick Harvey, Kim Marriott, and Peter J. Stuckey. An overview of HAL. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, pages 174–188, Virginia, USA, October 1999. Springer Verlag.
6. Tyson Dowd, Zoltan Somogyi, Fergus Henderson, Thomas Conway, and David Jeffery. Run Time Type Information in Mercury. In *Principles and Practice of Declarative Programming*, pages 224–243, 1999.
7. G. Gudjonsson and W. Winsborough. Update in place: Overview of the Siva project. In D. Miller, editor, *Proceedings of the International Logic Programming Symposium*, pages 94–113, Vancouver, Canada, 1993. The MIT Press.
8. Fergus Henderson, Thomas Conway, Somogyi Zoltan, and Jeffery David. The Mercury language reference manual. Technical Report 96/10, Dept. of Computer Science, University of Melbourne, February 1996.
9. M. Hermenegildo, F. Bueno, G. Puebla, and P. López. Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor. In D. De Schreye, editor, *1999 International Conference on Logic Programming*, pages 52–66, Cambridge, MA, December 1999. MIT Press.
10. Simon Hughes. Compile-time garbage collection for higher-order functional languages. *Journal of Logic and Computation*, 2(4):483–509, 1992.
11. S. B. Jones and D. Le Métayer. Compile-time garbage collection by sharing analysis. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture '89, Imperial College, London*, pages 54–74, New York, NY, 1989. ACM.

12. Andreas Kågedal and Saumya Debray. A practical approach to structure reuse of arrays in single assignment languages. In Lee Naish, editor, *Proceedings of the 14th International Conference on Logic Programming*, pages 18–32, Cambridge, July 8–11 1997. MIT Press.
13. Feliks Kluźniak. Compile-time garbage collection for ground Prolog. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1490–1505, Seattle, 1988. MIT Press, Cambridge.
14. Nancy Mazur, Gerda Janssens, and Maurice Bruynooghe. Towards memory reuse for Mercury. Report CW278, Department of Computer Science, Katholieke Universiteit Leuven, June 1999. http://www.cs.kuleuven.ac.be/publicaties/rapporten/CW1999.html.
15. Nancy Mazur, Gerda Janssens, and Maurice Bruynooghe. Towards memory reuse for Mercury. In K. Sagonas and P. Tarau, editors, *Proceedings of the International Workshop on Implementation of Declarative Languages*, Paris, France, 1999.
16. Nancy Mazur, Gerda Janssens, and Maurice Bruynooghe. A module based analysis for memory reuse in Mercury. In John Lloyd, Veronica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luis Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Computational Logic - CL 2000, First International Conference, London, UK, July 2000, Proceedings*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 1255–1269. Springer-Verlag, 2000.
17. M. Mohnen. Efficient Compile-Time garbage Collection for Arbitrary Data Structures. Technical Report AIB-95-08, RWTH Aachen, 1995.
18. M. Mohnen. Optimising the Memory Management of Higher–Order Functional Programs. Technical Report AIB-97-13, RWTH Aachen, 1997. PhD Thesis.
19. Greg Morrisett and John Reppy. The third annual ICFP programming contest. In Conjunction with the 2000 International Conference on Functional Programming, http://www.cs.cornell.edu/icfp/, 2000.
20. Anne Mulkers, Will Winsborough, and Maurice Bruynooghe. Live-structure dataflow analysis for Prolog. *ACM Transactions on Programming Languages and Systems*, 16(2):205–258, March 1994.
21. Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *The Journal of Logic Programming*, 29(1–3):17–64, October-December 1996.
22. The Mercury Team. ICFP 2000: The merry mercurians. Description of the Mercury entry to the ICFP'2000 programming contest, http://www.mercury.cs.mu.oz.au/information/events/icfp2000.html.
23. Mads Tofte and Talpin Jean-Pierre. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.
24. Philip Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, Albequerque, New Mexico, January 1992.