Department of Computing Science and Mathematics
University of Stirling

An Experiment in Compile Time Garbage Collection

Simon B Jones

Department of Computing Science and Mathematics, University of Stirling
Stirling FK9 4LA, Scotland

Telephone +44-786-467421, Facsimile +44-786-464551
Email sbj@uk.ac.stirling.compsci

Technical Report CSM-127

September 1994

Abstract

This report presents the details and conclusions of an experiment designed to assess the potential benefits to be obtained from a particular proposal for optimizing the execution of functional programs by *compile time garbage collection*. The optimizations proposed comprise the compile time insertion of code to directly re-use heap cells which are released and can be statically reallocated immediately. The optimizations are determined by the static *sharing analysis* of strict, first order functional programs using list (tree-like) data structures. The method is powerful enough to detect many places in programs where the optimization can be applied, but the effect on the performance of "typical" programs has not been practically assessed before. In this experiment a non-trivial Haskell program is adapted to run as a strict, first order LML program. Re-use optimization is performed (simple deallocation is not possible in LML as it has no free list in the heap), and the performance of the unoptimized and optimized versions are compared. The results turn out to be (surprisingly) disappointing: although the number of bytes allocated from the heap is reduced by about 8% and the time for garbage collections reduces by about 15%, the cell allocation time itself is not improved, and the garbage collection time amounts to only 10% of the total program execution time. There is a slight improvement in the locality of memory references, reducing the cache penalty overhead by about 3%. The total time reductions are of the order of 4.5% — which is not encouraging.

The work reported here was carried out whilst on sabbatical leave visiting the Programming Methodology Group, University of Göteborg and Chalmers University of Technology, Göteborg, Sweden, March-July 1992.

Acknowledgements

I would like to thank my colleagues in the Programming Methodology Group, University of Göteborg and Chalmers University of Technology, for many discussions, and for use of their software and hardware systems during my visit.

Thanks also to the University of Stirling for allowing me the period of sabbatical leave during which this research was carried out.

Contents

1 Introduction

COMPILE time garbage collection (*CTGC*) is an approach to improving the run time performance of a program by determining statically those points in a program where *dynamically* allocated heap storage cells will be released, and inserting special purpose code to deal with releasing the cells for subsequent re-use. The aim is to reduce the execution time of a program by

- recycling cells more cheaply than would be achieved by a general purpose garbage collector (but see [App87]),
- postponing garbage collections, hopefully making later ones completely unnecessary.

There are two ways of dealing with cells that have been released:

- if the heap management has a free list then a released cell can simply be added to the free list — the cost of run time checks on its accessibility (incurred by a garbage collector) are saved, and the next garbage collection is certainly postponed;
- if the release can be paired with a new allocation that will be executed in the subsequent code, then the cell need not be added to the free list: it can simply be re-used directly — the costs of adding to and extracting from the free list are avoided, and the next garbage collection is certainly postponed.

For example, consider executing the LML functions for naïve reverse and append *strictly*:

```
   reverse [] = []
|| reverse (x.xs) = append (reverse xs) (x.[])

   append [] ys = ys
|| append (x.xs) ys = x.(append xs ys)
```

With the functions defined in this multiple equation, pattern matching style, it is easy to see that in the second equation of both functions there is a cons cell matched on the left hand side to which there is *no reference* on the right hand side. Thus if either function is only ever called with an argument to which it has a unique reference, the cons cell will be released as soon as the pattern match and structure decomposition is complete, and code could be compiled into the pattern matching which returns the cell to the free list. Further, in both these equations, there is a requirement on the right hand side for a new cons cell — and so the released cell can be directly re-used rather than being returned to the free list. Whether or not these optimizations are possible depends on the context of call: if the 'main program' is a call of `reverse` with an unshared argument, then both equations can be optimized; but if `reverse` is called with a shared argument, then `append` is optimizable (because its first argument is still unshared), but `reverse` is not.

In [JLM89], Le Métayer and I proposed a method of CTGC for strict, first order functional programs using nested list data structures stored as trees in the heap. Both the optimizations mentioned above can be introduced by the method. The method is powerful enough to detect many places in programs where the optimizations can be applied. In some cases it allows the garbage collector to be dispensed with entirely (for example naïve reverse), but the effect on the performance of "typical" programs has not been assessed practically before.

This report presents the details and conclusions of an experiment designed to assess the potential practical benefits to be obtained from the CTGC proposed in [JLM89]. In this experiment

- a non-trivial Haskell program was adapted to run as a strict, first order LML program (see Section 5);
- sharing analysis was carried out — this involved an exploration of various possible abstract sharing pattern domains, since it was not initially obvious which would be the most beneficial (see Section 6);

- re-use optimizations were introduced — simple deallocation is not possible in LML as it has no free list in the heap (see Section 7);
- and the performance of the unoptimized and optimized versions were assessed and compared (see Sections 8, 9).

Since LML is continuously being improved, it is necessary to note that the experiments reported here were carried out during Spring/Summer 1992. In particular, most of the optimizations were performed on the M-code produced using the "`-fstrict`" compiler flag.

2 Summary of the [JLM89] method

THE reader should refer to [JLM89] for a full description of the method. The following summary gives the main points of the techniques applied in this report:

- The method is based on a *sharing analysis*: abstract interpretations of the program based on *sharing patterns* which represent the degree of sharing of the data structures of which they are abstractions.
- For each function in a program an abstract *sharing function* is calculated which captures information about the way that the sharing of the result of the function depends on the sharing of the arguments (the *transmitted* sharing) and any sharing introduced within the function (the *created* sharing).
- The arguments of each function are augmented with sharing information about the data carried by the parameters, and explicit heap management operations are introduced which act on, and propagate, this sharing information.
- Each augmented function is then 'compiled' by partially evaluating it with respect to what is actually known about the sharing of its arguments (for example, that the main inputs to the program are 'unshared'). In this way, some of the heap management decisions may be completely determined at compile time (and the partial evaluation introduces the optimized code), and others must be delegated to the usual run time heap manager. Note that this might lead to the introduction of multiple versions of a function — if it is specialized to several different sharing contexts.

The method followed in the work reported here differs from [JLM89] slightly:

- The functions are not augmented with additional sharing arguments and then partially evaluated.
- Instead, a *collecting interpretation* approach was adopted, much like that described in [Hud87].
- A global collecting interpretation of the program was performed which calculates for each call of each function in the program the degree of sharing of each of its arguments, and hence, by a least upper bound calculation, a safe assumption about the degree of sharing of each argument of each defined function.
- Functions were specialized with respect to these l.u.b.s of the sharing of their arguments. This leads to single versions of each function — though the approach could be adapted to creating multiple versions too.

For various practical reasons, other details of the calculations differ from [JLM89] as well. These will be mentioned where relevant in the following sections.

3 Heap management in LML

LML is continuously being improved. In particular, when the experiments reported here were carried out (Spring/Summer 1992) the heap management was of an 'adaptive 2-space copying' type, but a generational scheme was under development.

The memory allocated is divided logically into two areas: cell allocation is performed linearly from one until it is exhausted, and then a garbage collection phase copies the active data structures into the other, and the rôles are switched. For the general principles see [Coh81]. In this implementation the size of the heap is adjusted after a garbage collection: quoting from the LML manual[AJ]:

> How large a part [of the allocated memory is used] is determined after each garbage collection. The amount used (i.e. available for allocation) is the amount that was copied when the collection occurred multiplied by eight and 200 kilobytes added to this. In this way the working set is adapted to the amount of heap that is actually in use.

Since the garbage collection (GC) is the copying type, the time taken for a single GC is (roughly) proportional to current data structure size and not to the heap size. On the machine used for the experiments (a Sun 4 SPARC, with 32 Mbytes of main memory with a 64 Kbyte write through cache) LML copies at about 4 Mbytes per second [1]. For programs with small heap occupancy this is relatively cheap, and the GC time is a small proportion of total time (5-10% maybe). For programs with a larger occupancy it may be a large proportion (for strict reverse of a long list it is more like 30%). [2]

Cell allocation is very efficiently implemented at the machine code level: basic blocks often contain several allocations, and the heap manipulation is optimized at the basic block level. The heap pointer is tested at the start of each basic block that contains at least one allocation to ensure that there is enough space for all the allocations, and the heap pointer is incremented at the end of such basic blocks; these operations are fast since the heap pointer is held in a CPU register, and if the basic block contains several cell allocations then the cost per cell is very low.

An unfortunate consequence of the linear memory allocation used in the 2-space heap manager is that there is no free list. This implies that only cell re-use optimizations are possible — nothing can be done with cells that can be determined statically to be releaseable unless the release can be paired with a (re-)allocation. Thus it is possible that many optimization opportunities cannot be exploited. However, [App87, Tya93] show that recycling a cell in a mark/sweep garbage collector (with a free list) with deallocation optimization in effect, might not be as efficient in general as a copying collector with no optimization in effect — so perhaps the lost opportunities are not serious. This is especially true if a generous amount of memory is available: [App87, Tya93] estimate that it will be true if the heap is at least roughly 15 times the data structure size.

The current consensus seems to be that multi-space heap managers are the appropriate choice for good implementations, which would make assessing the effectiveness of simple deallocation optimization a rather academic affair! Thus, the experiments carried out were of the effectiveness of the re-use optimization.

4 A problem with GC performance, and the effect of optimization

At first sight, the optimizations suggested above in Section 1 are quite likely to reduce the execution time of a program: the application specific parts of a program will remain unchanged (all those computations must still occur), but the garbage collections are postponed (since previously allocated cells are re-used rather than free cells being allocated), and in general the number of

[1] Determined from a test which built a large data structure and then carried out a single, timed GC

[2] For a variety of programs, with a variety of sizes of data set, the GC times as %-ages of total time were: 11%, 5%, 20%, 7%, 23%, 20%, 23%, 7%, 16%, 17%, 44%

garbage collections will decrease. If we do fewer GCs then the overall execution time will reduce by the time previously spent in performing those GCs.

Unfortunately, the reality is not that simple, as the actual cost of a postponed GC could well be *greater* than that of the GC of which it is a postponement! The reason for this is that:

- the cost of an individual GC is proportional to the heap occupancy during the collection (each cell of the active data structures in the heap must be copied);
- the heap occupancy can fluctuate considerably during execution;
- and hence a postponed GC may as easily occur at a moment of *higher* heap occupancy as at a moment of lower occupancy.

(A corollary of this is that GC costs may well be reduced if additional, though redundant, cell allocations are added to a program!)

Clearly, in the case of 'complete optimization', where every allocation can be satisfied by re-allocating a statically released cell (whether by direct re-use optimization, or indirectly via a free list), the GC time will reduce to zero. Examples such as naïve reverse have this property, but in general such complete optimization will not be possible. In the case of partial optimization, the GCs will certainly be postponed, and, if we are fortunate, some will disappear completely, but the total GC time may still increase (*even if* there is a reduced number of GCs), although this would be an unlucky outcome.

Optimization might sometimes increase both the number and the total time of GCs — if the GCs move to moments of *very* high occupancy, for example.

This phenomenon is a very serious problem: optimizations may be beneficial or disadvantageous in a way that is beyond the scope of current analytical techniques to predict. In the experiments reported here, the problem is simply ignored: the assumption is that, in a program execution with many GCs, the unpredictable increases and decreases of individual GC times will balance out, and improvements in execution speed will arise simply from a reduced number of GCs. However, the phenomenon is bound to be present, and it will probably cause confusing fluctuations in any trends in the data collected.

Further work

The problem perhaps suggests that a productive optimization technique may be to analyse a program to determine *when* to do GCs: the aim would be to trigger them explicitly at moments of low heap occupancy. (For example, in Prolog it may be productive to perform GCs after 'cuts' because a 'cut' throws away a potentially large backtracking record.) As to how a functional program could be analysed to determine this, especially under lazy evaluation, I have no suggestions at present.

5 The Conway number package

A PACKAGE of functions was available, from an earlier Haskell exercise, for operating on an unusual number representation: "Conway numbers". The mathematical details are not relevant here — they can be found in [Con76]. The arithmetic operations on Conway numbers are highly recursive, they make use of operations on *sets*, and in Haskell it was natural to exploit classes. The package was a non-trivial program, and hence seemed like a reasonable basis for an optimization experiment. Further, it did not depend in any essential way on laziness, and so could be adapted for the [JLM89] method. The Haskell package appears in Appendix A.

The Haskell Conway package was not useable as it was, and was adapted as follows:

- The Haskell was converted to LML: removing the classes and overloading, and collapsing the two modules into one because the modules became mutually recursive, which LML can't handle.

- Explicit copies of the LML library functions were added (because the pre-compiled ones are lazy code, and return results containing suspensions).
- The code was rearranged to avoid the need for laziness (this just meant splitting up some `let` blocks), and the need for higher order working (explicitly customized versions of `map`, `filter`, etc were added).
- Redundant/inessential functions were deleted from the modules.

During this process the functions were kept in a form which would make it easier to find potential optimizations. `hd` and `tl` were avoided; instead, each function was defined using multiple equations and pattern matching. In this form it is easy to see whether, for any argument of a function, the main constructor of a data structure is potentially re-usable: it will be so if it is not named by an "`as` pattern" (if it *is* named as an `as` pattern then some *necessity analysis* (see [JLM89]) will be needed; there turned out to be very few `as` patterns in the program). Of course such a constructor cell will only be actually re-useable if the function is never called with a shared argument — and this must be determined by analysis.

One further change to the package was the expansion of all strings to be explicit constructions of lists of characters. LML pre-allocates storage for strings, which means that the `cons` cells are certainly not reusable (see discussion of pre-allocated constants in Section 6), and `@` will certainly not be optimizable since it receives such strings as its first argument. Expanding the strings at least allowed the possibility of optimization, though the trade-off was not fully assessed.

The resultant LML package is given in Appendix B.

None of the translation, expansion, pruning, etc changed the nature of the computations carried out in any essential way. There were various other simplifications that could have been made that would have reduced the heap usage of the program and also simplified the CTGC analysis, but the program was retained in its "natural" state. For example, the key types are:

```
type SetCNum = MkSet (List CNum)
type CNum = MkCNum SetCNum SetCNum
```

The constructor `MkSet` is unnecessary (and each use implies a cell being allocated), but it was retained.

The LML Conway package contains 34 functions, most of them are called by one or more other functions in the package. There are several mutually recursive groups (the largest contain 4 and 5 functions); its complexity is apparent from the "calling diagram" given in Appendix B. 13 functions are exported. The types involved are the two mentioned above, plus lists, chars and bools. Altogether perhaps 150 lines of actual function definitions.

Three simple main programs were chosen for the performance experiments — just some simple arithmetic like "two multiplied by three" and so on [3]. These main programs are given in Appendix C. Since the optimizations being investigated necessitate a global program analysis, the Conway package can only analysed and optimized in the context of these main programs. However, fortuitously, each main program enables the same optimizations in the package.

6 The sharing analysis

THE intention in this experiment was to assess the performance improvements arising from the proposed optimizations. Thus the entire process of program analysis and optimization was carried out by hand; the aim was not to design and implement the necessary extensions to the LML compiler. Much of the hand calculation involved in abstract and collecting interpretations was rather tricky (for example, finding the fixpoints of the abstract sharing functions), and would have taken a very long time if carried out completely formally. They were carried out semi-formally.

[3] In fact, the Conway number representation, and the operation definitions, are *so* recursive that calculations even slightly more complex were simply unfeasible!

6.1 The sharing pattern domains

The sharing analysis must be carried out with respect to abstract domains of *sharing patterns*. In [JLM89] the abstract domains presented are for lists, with the assumption that all simple data values do not reside in their own heap cells.

Several aspects of the Conway package, and of LML, make that analysis not directly applicable:

- the additional `MkSet` and `MkCNum` constructors;
- simple constants like characters, `true`, `false` and `nil` are held in their own, tagged cells; (this applies to strings as well [4] — this can only be determined by inspection of the machine code produced);
- these constant cells are pre-allocated, and so evaluating such constant expressions returns a pointer to a pre-allocated cell, which thus must always be assumed to be shared [5].

Therefore, the sharing domains required are different in detail from those in [JLM89].

Further, the scheme outlined in [JLM89] allows the sharing domains to be customized to capture more or less detailed information about the degree of data structure sharing present. The domains must be selected to give an appropriate degree of precision in any particular application — the more approximate the domains are, the easier and quicker the analysis will be, but the less is the optimization that is likely to be found.

In the Conway package it is not immediately clear how much precision is required in the sharing domains in order to find all the optimization possible. Several iterations of domain refinement and sharing analysis were carried out before sufficient information had been extracted from the program.

> **Further work**
>
> If the optimizations under investigation do turn out to be effective, then, in a serious software engineering environment, an optimizing compiler will be faced with exactly this problem; the structures of the sharing domains are determined by the type definitions in a program, but the exact degree of detail is not immediate. There may be a case for including the engineer/programmer in the final optimization process — to guide the compiler in its choice of sharing domains.

The types in the Conway package for which we need sharing domains are:

```
type SetCNum = MkSet (List CNum)
type CNum = MkCNum (SetCNum) (SetCNum)
```

There are also booleans, characters and lists (of `CNum` and characters).

The sharing domain for each type `Foo` will be denoted by $\mathcal{S}$`Foo`. As a generalization, a domain element 0 will usually indicate "unshared", and 1 will indicate "shared". Pattern domains are built up by tupling: denoted by $< \_, \_ >$.

Thus we will always have:

$$\begin{array}{lcl} \mathcal{S}\texttt{Bool} & = & \{1\} \\ \mathcal{S}\texttt{Char} & = & \{1\} \end{array}$$

since booleans and characters are always shared.

[4] LML even allocates some explicitly constructed lists of characters as constants — for example, the `(')'.□)` at the end of the definition of `show CNum`

[5] The fact that `nil` must be treated as shared is a serious problem: *every* function that returns a list with `nil` as one of the options must have its result treated as a shared object. In particular, the tail of a list, whether arising from a pattern match or a use of `tl`, must be treated as shared.

First iteration

A very simple approximation: inspect the function definitions to see which patterns occur as arguments, then set up domains to discriminate between the shared and unshared cases of the head ('top') constructors, treating the sharing of subcomponents as 'unknown', i.e. 'shared' (for safety). The following patterns occur: `(MkSet _)`, `(MkCNum _)`, `(MkCNum (MkSet _) (MkSet _))` and `(x.xs)`. Thus:

- `SetCNums`: either the `MkSet` constructor (and hence the whole object) is shared (denoted by 1),

  or it is unshared but we know nothing about the embedded list (denoted by 0):

  $$\mathcal{S}\texttt{SetCNum} = \{0,1\} \qquad \text{with } 0 \sqsubset 1$$

- `CNums`: either the `MkCNum` constructor is shared (denoted by 1),

  or it is unshared and the two component sets are shared as given by $\mathcal{S}\texttt{SetCNum}$ (denoted by $< \ , \ >$):

  $$\mathcal{S}\texttt{CNum} = \{1\} \cup \mathcal{S}\texttt{SetCNum} \times \mathcal{S}\texttt{SetCNum}$$
  $$\text{with } <p_1,p_2> \sqsubset 1$$
  $$\text{and } <p_1,p_2> \sqsubseteq <p_3,p_4> \textit{ iff } p_1 \sqsubseteq p_3 \textit{ and } p_2 \sqsubseteq p_4$$
  $$\forall p_i \in \mathcal{S}\texttt{SetCNum}$$

- `Lists`: either the `cons` is shared (denoted by 1),

  or it is unshared but we know nothing about the head nor the tail of the list (denoted by 0):

  $$\mathcal{S}\texttt{List} = \{0,1\} \qquad \text{with } 0 \sqsubset 1$$

This looked quite promising, but it turned out not to be adequately discriminating. For example, consider the functions

```
filterset1 cn2 (MkSet xs) = MkSet (filter1 cn2 xs)

filter1 cn2 [] = []
|| filter1 cn2 (x.xs) = if cn2 lesseq x then x.filter1 cn2 xs
                                        else filter1 cn2 xs
```

The second argument of `filterset1` is a `SetCNum`, so, in the abstract interpretation of the functions, the sharing of this argument is represented by an element of $\mathcal{S}\texttt{SetCNum}$. If this element is 1 then the entire set is shared, so the sharing of the list subcomponent `xs` is 1; if the element is 0 then we do not know what the sharing of `xs` is, so we must assume that it is shared, 1. Thus, we predict that the second argument of `filter1` *must* be treated as shared, which implies that `filter1` cannot be optimized by deallocation nor reuse of the `cons` cell which is pattern matched in the second equation. However, it is quite clear that, if `filterset1` is called with a completely unshared second argument, then `filter1` *is* optimizable!

Final iteration

Several further iterations of sharing domain design and refinement yielded the following: (for simplicity domain injections and projections have been omitted, and the symbols 0 and 1 are heavily overloaded, but the meanings are intuitively clear from context)

- `Lists`: either some sharing (elements or spine) (1),

  or completely unshared (except that the final `nil` is always shared) (0):

  $$\mathcal{S}\texttt{List} = \{0,1\} \qquad \text{with } 0 \sqsubset 1$$

This gives us the following abstract functions:

$$\begin{array}{lll} \mathcal{S}\texttt{cons}\ 0\ \ 0 = 0 & \mathcal{S}\texttt{hd}\ 0 = 0 & \mathcal{S}\texttt{tl}\ 0 = 0 \\ \mathcal{S}\texttt{cons}\ \_\ \ 0 = 1 & \mathcal{S}\texttt{hd}\ 1 = 1 & \mathcal{S}\texttt{tl}\ 1 = 1 \\ \mathcal{S}\texttt{cons}\ 0\ \ 1 = 1 & & \\ \mathcal{S}\texttt{cons}\ \_\ \ 1 = 1 & & \end{array}$$

For example, the equations for $\mathcal{S}\texttt{cons}$ imply that $\mathcal{S}\texttt{cons}\ 1\ 0 = 1$, which means that if the head element of a list built with `cons` has some (non-0) sharing, and the tail is unshared, then the list built has some sharing. The equation $\mathcal{S}\texttt{hd}\ 1 = 1$ means that the head element of a list which has some sharing, itself may be shared.

This will apply to both lists of `CNum` and lists of `Char` (though the latter are *always* shared, for the reason above).

- `SetCNums`: either the `MkSet` constructor is shared (1),

  or the `MkSet` is unshared and the sharing of the embedded list is given by $\mathcal{S}\texttt{List}$. We build a discriminated union, denoting the tagged elements from $\mathcal{S}\texttt{List}$ by $<1>$ and $<0>$, with $<0>$ abbreviated to 0:

$$\begin{array}{rcll} \mathcal{S}\texttt{SetCNum} & = & \mathcal{S}\texttt{List}^{<>} \cup \{1\} & \\ & = & \{0, <1>, 1\} & \text{with} \quad 0 \sqsubset <1> \sqsubset 1 \end{array}$$

  This gives us the following abstract functions ($\downarrow$ is an abstract function that selects the embedded list from a `SetCNum`):

$$\begin{array}{ll} \mathcal{S}\texttt{MkSet}\ 0 = 0 & 0\downarrow = 0 \\ \mathcal{S}\texttt{MkSet}\ 1 = <1> & <1>\downarrow = 1 \\ & 1\downarrow = 1 \end{array}$$

- `CNums`: either the `MkCNum` constructor is shared (1),

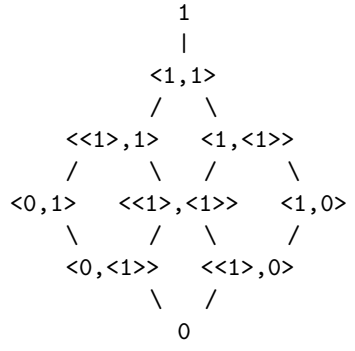
  or the `MkCNum` is unshared and the sharing of each component set is given by $\mathcal{S}\texttt{SetCNum}$ (denote the pair of components by $<\ ,\ >$, and abbreviate $<0,0>$ by 0):

$$\begin{array}{l} \mathcal{S}\texttt{CNum} = \mathcal{S}\texttt{SetCNum} \times \mathcal{S}\texttt{SetCNum} \cup \{1\} \\ \quad \text{with } <p_1, p_2> \sqsubset 1 \\ \quad \text{and } <p_1, p_2> \sqsubseteq <p_3, p_4> \textit{ iff } p_1 \sqsubseteq p_3 \textit{ and } p_2 \sqsubseteq p_4 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \forall p_i \in \mathcal{S}\texttt{SetCNum} \end{array}$$

  This gives us the following abstract functions ($\downarrow_l$ and $\downarrow_r$ are abstract functions that select the component sets from a `CNum`):

$$\begin{array}{ll} \mathcal{S}\texttt{MkCNum}\ s_1\ s_2 = <s_1, s_2> & \\ <s_1, s_2>\downarrow_l = s_1 & <s_1, s_2>\downarrow_r = s_2 \\ 1\downarrow_l = 1 & 1\downarrow_r = 1 \end{array}$$

  A pictorial view of the elements of $\mathcal{S}\texttt{CNum}$ and their ordering:



With less elaborate domains than this there was certainly optimization that had not been found. There is a strong possibility that if the $\mathcal{S}$`List` domain were expanded to discriminate three cases (shared spine, unshared spine but shared elements, and completely unshared (except the `nil`)) then some additional optimization would have been possible in `@`. This is because `@` is used only to combine lists of characters and, using the $\mathcal{S}$`List` above, lists of characters must be treated as shared due to the pre-allocated characters, and so unshared spines cannot be reused. However, in the test main programs chosen there is little output, so this improvement would have had little impact on the performance.

6.2 The sharing functions

The sharing functions are the abstractions of the functions present in the program. In general they are determined by constructing the least fixpoints over the sharing domains. Many of them are immediately obvious from the definitions in Appendix B, and they are all surprisingly simple. The sharing function corresponding to function `fn` is denoted by $\mathcal{S}$`fn`; it has the same arity as `fn`, and describes how the sharing of the result of `fn` depends on the sharing of its arguments (if at all).

Working (roughly) upwards from the leaves in the calling diagram in Appendix B, writing s, s_i for elements of the appropriate sharing domains:

$\mathcal{S}$`isempty` $s = 1$ $\quad$ $\mathcal{S}$`sing` $s = \mathcal{S}$`MkSet`($\mathcal{S}$`cons` $s\ 0$)
$\mathcal{S}$`zero` $s = 0$ $\quad$ $\mathcal{S}$`lesseq` $s_1\ s_2 = 1$
$\mathcal{S}$`filter1` $s_1\ s_2 = s_2$ $\quad$ $\mathcal{S}$`filterset1` $\_s = s \sqcap <1>$
$\mathcal{S}$`filter2` $s_1\ s_2 = s_2$ $\quad$ $\mathcal{S}$`filterset2` $\_s = s \sqcap <1>$
$\mathcal{S}$`less` $s_1\ s_2 = 1$ $\quad$ $\mathcal{S}$`eq` $s_1\ s_2 = 1$
$\mathcal{S}$`max` $s_1\ s_2 = s_1 \sqcup s_2$ $\quad$ $\mathcal{S}$`min` $s_1\ s_2 = s_1 \sqcup s_2$
$\mathcal{S}$`maxinset` $s = s\downarrow$ $\quad$ $\mathcal{S}$`mininset` $s = s\downarrow$
$\mathcal{S}$`member` $s1\ s2 = 1$
$\mathcal{S}$`addelement` $s_x\ s_s = \mathcal{S}$`MkSet`($\mathcal{S}$`cons` $s_x\ 0) \sqcup s_s$
$\mathcal{S}$`quickset` $s = \mathcal{S}$`MkSet` s $\quad$ $\mathcal{S}$`union` $s_1\ s_2 = \mathcal{S}$`MkSet`$(s_1\downarrow) \sqcup s_2$
$\mathcal{S}$`@` $s_1\ s_2 = s_1 \sqcup s_2$ $\quad$ $\mathcal{S}$`convertCNums` $s = 1$
$\mathcal{S}$`show_CNum` $s = 1$ $\quad$ $\mathcal{S}$`show_setCNum` $s = 1$
$\mathcal{S}$`conway_error` $s_1\ s_2 = 1$ $\quad$ $\mathcal{S}$`buildCNum` $s_1\ s_2 =< s_1, s_2 >$
$\mathcal{S}$`mapsetadd` $s\ s_s = 0$ $\quad$ $\mathcal{S}$`add` $s_1\ s_2 = 0$
$\mathcal{S}$`mapsetneg` $s = 0$ $\quad$ $\mathcal{S}$`neg` $s = 0$
$\mathcal{S}$`addone` $s = 0$ $\quad$ $\mathcal{S}$`sub` $s_1\ s_2 = 0$
$\mathcal{S}$`mapsetf` $s_1\ s_2\ s_3\ s_4 = 0$ $\quad$ $\mathcal{S}$`unionmapmapsetf` $s_1\ s_2\ s_3\ s_4 = 0$
$\mathcal{S}$`mult` $s_1\ s_2 = 0$ $\quad$ $\mathcal{S}$`f` $s_1\ s_2\ s_3\ s_4 = 0$

Further, there is one call of the special LML function, `fail`, for which we assume:

$\mathcal{S}$`fail` $s = 0$

Here is an example of how to interpret these equations: The function application `filter1 cn xs` filters the list `xs` to retain those elements related to `cn`. Thus the sharing of `cn` has no effect on the result. The spine of the result list is built of new cells, and the elements are a subset of the elements of `xs`. Therefore the sharing of the result is the same as the sharing of the list `xs` (given the meanings of the members of the list sharing domain $\mathcal{S}$`List` given above), and so $\mathcal{S}$`filter1` $s_1\ s_2 = s_2$.

6.3 Now, what can be optimized?

The potentially optimizable functions are easy to identify: equations with

- a data constructor pattern (with no '`as`') on the left,
- and an allocation requirement somewhere on the right,
- such that the sizes of the constructor and allocation are the same.

Some of the equations have *several* potentially reuseable constructors!

Fortunately the LML compiler allocates each of the constructors `cons`, `MkSet` and `MkCNum` the same amount of heap (3 words), so each can be reused as any other.

17 of the 34 functions in the package are potentially optimizable, by this criterion. They are: `union`, `member`, `addelement`, `maxinset`, `mininset`, `show CNum`, `show setCNum`, `convertCNums`, `filterset1`, `filter1`, `filterset2`, `filter2`, `mapsetadd`, `mapsetneg`, `mapsetf`, `unionmapmapsetf`, `@`.

For example in the definition

```
show_SetCNum ( MkSet s) = ('{' , []) @ convertCNums s @ ('}'.[])
```

the `MkSet` could be reused as the `cons` `,`.

In order to know whether a function *is* optimizable we need to know whether all calls of that function (for any particular main program) will have an unshared argument. In fact, we only need to know that the argument is unshared to an appropriate degree. For example, if there is only the opportunity to re-use the `MkSet` constructor, then we only need to be sure that *that* constructor is unshared; the degree of sharing of the embedded list is irrelevant.

6.4 Collecting interpretation

For each main program a "collecting intepretation" was carried out using sharing domain values instead of actual values, to find the l.u.b. of sharing values for the relevant arguments of the potentially optimizable functions in the Conway package. Again, this was by hand.

Finally seven of the functions turned out to be actually optimizable. In fact they were the same seven for each of the example main programs, and no main program could allow more optimizable functions (the remaining, un-optimizable functions were intrinsically un-optimizable, due to the interdependence and sharing within the package). The particular main programs were rather simple, so other examples might give less optimization opportunities.

The *context* of a given function application during the execution of a program is (in general) a tuple comprising the sharing values associated with the actual arguments of the call. The *least upper bound* context of a given function application in the text of a program is the lub of all the contexts that occur for that function application during execution of the program. We need to know the lub context as this guarantees safety in our assumptions about the sharing of any given argument — we will not optimize assuming it to be unshared when it in fact might be shared.

The collecting interpretation is a global analysis of a program: it derives its results from the sharing in the initial expression that is being evaluated, and any further sharing that is created. Therefore it is convenient to look at the analysis in a top down fashion.

For example, considering the main program Test 1 in Appendix C and the function calls it makes on the Conway package, we can reason as follows:

- `add_one` is called with contexts: 0 since it is applied to the result of `zero`, and $\mathcal{S}$`zero` $s = 0$; 0 since it is applied to the single use of variable `one`, which is the result of `add one`, and $\mathcal{S}$`add_one` $s = 0$; 1 since it is applied to the first of two uses of variable `two`.

  The lub of these immediately visible contexts is 1, and, since this is the top element in the sharing domain, we need not consider any other occurrences of `add one` in order to find the lub context.

- `show_CNum` is considerably harder: its lub context is in fact 0, but that fact is far from immediate. (Since it has a re-useable constructor in its definition, it is therefore optimizable.)

  In the test main program `show CNum` is called with context 0 (applied to the result of `mult`). In the Conway package `show CNum` and `convertCNums` are mutually recursive: `show CNum` decomposes its `CNum` argument and applies `convertCNums` to both the component lists of `CNums`, `convertCNums` applies `show CNum` to each of the `CNums` in the lists, and itself recursively to the tails of the lists. Inductively, all these data structures are unshared if the

original argument of `show CNum` is unshared; therefore this contribution to the lub context of `show_CNum` is 0.

However, `convertCNums` is also called from `show SetCNum`, itself called from `conway error`, and that from `buildCNum`. Fortunately it is easily determined that each of the three calls of `buildCNum` have unshared arguments (i.e. because they are values produced by functions whose sharing interpretations (see section 6.2) guarantee unshared results), and this unshared property is propagated to `convertCNums`.

So the true overall lub context of `convertCNums` is 0 and also that of `show CNum`.

- `add` is not called directly from the main program. However, we do not need to know its original call context, because its mutual recursion with `mapsetadd` is of such an inconvenient kind that its lub context must be 1. `add` calls `mapsetadd` four times, with an entirely unavoidable sharing of both arguments (whatever the sharing of `add`'s arguments is). `mapsetadd` then propagates this sharing to the recursive calls of `add`, thus guaranteeing the lub context of 1.

Applying this analysis to all the functions, and combining the results for tests 1, 2 and 3, yields the following lub contexts:

`isempty`	1	`sing`	1
`zero`	0	`lesseq`	$(1,1)$
`filter1` †	$(1,1)$	`filterset1` †	$(1,1)$
`filter2` †	$(1,1)$	`filterset2` †	$(1,1)$
`less`	$\star$	`eq`	$(0,1)$
`max`	$(1,1)$	`min`	$(1,1)$
`maxinset` †	1	`mininset` †	1
`member` †	$(0,1)$	`addelement` †	$(1,0)$
`quickset`	$\star$	`union` †	$(0,0)$
`@` †	$(1,1)$	`convertCNums` †	0
`show_CNum` †	0	`show setCNum` †	0
`conway_error`	$(0,0)$	`buildCNum`	$(0,0)$
`mapsetadd` †	$(1,1)$	`add`	$(1,1)$
`mapsetneg` †	0	`neg`	0
`add_one`	1	`sub`	$(0,0)$
`mapsetf` †	$(1,1,1,1)$	`unionmapmapsetf` †	$(1,1,1,1)$
`mult`	$(1,1)$	`f`	$(1,1,1,1)$

Note: $\star$s indicate functions not called by the test programs. † indicates functions potentially optimizable by the pattern/allocation criterion above.

At last we can now see fairly easily which functions can actually be optimized: those above marked with †, and which have lub contexts with 0 corresponding to an argument with a constructor that can be re-used.

The optimizable functions are thus: `member`, `addelement`, `union`, `convertCNums`, `show CNum` (three re-usable constructors, although `show CNum` is not called often enough for this to have a serious impact on performance), `show setCNum`, and `mapsetneg`.

7 Introducing the Optimizations

The LML *eager* code generator is not very good (since it is not really intended to be used): apply nodes are allocated immediately before each function call which are overwritten at the end of the call, and are then thrown away. The code was altered manually to

- remove the code at the end of each function which overwrites the apply node with the result,
- remove the allocation of the redundant apply nodes allocated immediately before each function call.

Those changes provided basic, eager, unoptimized code whose performance could be measured.

Seven of the Conway functions are optimizable. Each re-use optimization opportunity means a constructor cell belonging to an argument of a function, which would otherwise become garbage following pattern matching at function entry (extraction of the pointers to its components), can be re-used in the body of the function. During pattern matching a pointer to the constructor cell is pushed onto the stack. Thus, the code for the heap cell allocation in the body of the function can be replaced by a simple fetch of the pointer from the stack.

The optimization was easily achieved by manually altering the generated code. Care had to be taken over the remaining code to keep it consistent with the optimizations; for example: "end of heap" tests can be removed at the start of basic blocks (see section 3) in which all new cell allocations have been removed; incrementing of heap pointers at basic block exit must be reduced where fewer new cells have been allocated.

8 Assessing the effectiveness of the optimizations

THE aim is to assess the increase in performance of the Conway programs when the re-use optimizations are inserted. The optimizations only affect the memory management and not the primary computation carried out by the programs. Therefore, the time spent in memory management must be measured, with and without the optimizations. Unfortunately, this information was very hard to obtain, for a number of reasons.

8.1 Timing problems

The experiments were carried out on a Sun 4 SPARC under 'Unix'. Times noted were 'user' rather than 'system' or 'elapsed' times. The resolution of the reported times was 0.01 seconds.

LML provides some timing statistics: principally useful here are the total run time and the total time spent in GC. Unfortunately, the times produced are very variable. This due to the normal, unpredictable operating system overheads (even when no other user tasks are running). Some trials indicated that times may easily vary by up to 5%.

Approximate *average total run times* can be measured fairly well (provided they are at least a few seconds). However, detecting whether there was a *change* in total run time when changing to the optimized version was a very difficult problem: the variability of the measured run times was typically of the same order of magnitude as the change due to optimization. For example, see the line labelled ΔT in Table 3 (page 15). This is therefore a very unreliable way to assess the performance improvement.

Further, the times for *individual* garbage collections, although collected and reported by LML, are of the same order of magnitude as the resolution of the Unix clock—so collecting them and adding them up makes no sense. For example, see the lines labelled *G (secs)* in Table 1 (page 14) and *G' (secs)* in Table 2 (page 15), dividing in each case by the number of GCs.

One more factor which complicates the interpretation of any measured timings is the possibility that improved program performance could be due to better locality in memory use. Lennart Augustsson expressed the opinion that locality effects may be very significant with modern memory/cache architectures, since the time taken for memory accesses dominates the activity. It is very hard to determine the existence or magnitude of this effect in these experiments, and so the possibility was noted and simplifying assumptions made.

Fortunately, LML also collects various *counted* statistics, which are therefore precise. Of principal interest here are

- the number of bytes allocated from the heap,
- the number of bytes copied during GC,
- and the number of GCs.

Given the number of bytes copied during GC, and a reliable average copying rate of 4 Mbytes/sec (see section 3) (plus the roughly realistic assumption that GC time is proportional to the number of cells copied), the total time for garbage collection is easily calculated.

[Better would be to repeat these experiments on a machine with instruction level profiling.]

8.2 A Performance Model

Given the difficulties outlined above, and the various components of the total execution time of the test programs, it was necessary to base the assessment of the optimizations on a detailed performance model.

The model is based on the following parameters:

T Total program execution time.

B Basic program execution time (the essential computations involved in the program) assuming that all memory is accessible at cache speeds (stack manipulation, calculations, filling the fields of allocated cells, etc).

C Time penalty for requiring access to memory not currently in the cache.

A Total time to *allocate* cells.

G Total time for garbage collection.

These are related in the model by the equation:

$$T = B + C + A + G$$

When a program has CTGC optimization applied:

B will remain unaltered.

C may reduce (due to improved locality).

A will remain roughly the same (see below).

G will change in some way to be determined: possible reducing since there may be fewer GCs, possibly increasing if GCs occur at more expensive moments (see section 4).

T will therefore reduce by the reductions in C and G.

The reasoning behind the assertion that A will remain roughly the same depends on the details of the allocation code, and is as follows:

- The number of allocations, unoptimized and optimized, remains the same.
- For each cell in the *unoptimized* program the allocation cost is
  the time to take a copy of the heap pointer register and to add an offset
  + some *fraction* of the cost of testing and incrementing the heap pointer (because these are done *once* per basic block)
- For each cell in the *optimized* program the allocation cost is
  either *unoptimized:* the same as the original cost,
  or *optimized:* the time to take a copy of a pointer found by *indexed addressing* access to the stack memory ("not cheap")

We are interested in assessing various aspects of the improved performance:

- The change in the time required for GC, ΔG. This is indirectly measurable (from the number of bytes *copied*).

- The overall reduction in execution time, ΔT. The magnitude of T is directly measurable, though not reliably, for the reasons above.
- The reduction in cache penalty, ΔC, due to improved locality. This cannot be measured directly, but, on the assumption that the measured total execution times are accurate, ΔC can be calculated indirectly:

$$\begin{array}{rcll} C & = & T - B - A - G & \\ C' & = & T' - B - A - G' & (B \text{ and } A \text{ do not alter}) \\ \Delta C & = & C - C' = (T - G) - (T' - G') & \end{array}$$

- Although we are assuming that the total time for cell allocation, A, remains the same, it is of considerable interest to know the reduction in the *number* of cell allocations *from the heap*, ΔH; i.e. how many have been replaced by re-use optimizations. This is directly measurable (from the number of *bytes* allocated from the heap, H).

Thus the parameters to be measured are: the number of bytes allocated from the heap (H), the number of bytes copied (G) and the total execution time (T).

9 Performance results and analysis

9.1 Measurements

PERFORMANCE measurements were carried out on the three test programs in Appendix C, with the code unoptimized and optimized for re-use of cells. The results are summarized in Tables 1, 2 and 3. The measurements are expressed in various units, as appropriate. The time values, T and T', are averages of clock timings with a resolution of 0.01 second; the averages given here are rounded to the same resolution, but the full accuracy was used in calculating G, G', $\Delta G, \Delta C$ before rounding them, too, to that resolution or to two significant digits, as appropriate. On the other hand, H and G (in bytes) are precise values, and values calculated from them alone are given to 3 significant digits.

		Test 1	Test 2	Test 3
T	(secs)*	0.63	0.75	9.27
H	(bytes)	1315284	1725120	17615964
G	(bytes)	283556	88680	5855444
	(secs)	0.0709	0.0222	1.46
	(% of T)	11	3.0	16
No. of GCs		7	9	61

Table 1: Unoptimized performance parameters
(* These times are averages of 100 repetitions, over a period of 8 hours.)

9.2 Analysis

9.2.1 Allocations from the heap

The reduction in the number of bytes allocated from the heap, ΔH, is reasonably consistent across the tests: 7–9%. This gives a direct indication of the effectiveness of the optimizations carried out—they were clearly at reasonably significant points in the code. These results are moderately encouraging, but there is no particular reason to expect this reduction for other programs.

		Test 1	Test 2	Test 3
T'	(secs)*	0.59	0.71	8.9
H'	(bytes)	1220532	1562544	16464960
G'	(bytes)	236060	69712	5329788
	(secs)	0.0590	0.0174	1.33
	(% of T')	9.9	2.4	15
No. of GCs		7	8	57

Table 2: Optimized performance parameters
(* These times are averages of 100 repetitions, over a period of 8 hours.)

		Test 1	Test 2	Test 3
ΔT	(%)	5.3	4.7	3.6
ΔH	(%)	7.20	9.42	6.53
ΔG	(%)	16.8	21.4	8.98
	(% of T)	1.9	0.63	1.4
ΔC	(secs)	0.02	0.03	0.20
	(% of T)	3.4	4.1	2.1

Table 3: Reductions in performance parameters

Unfortunately this reduction in allocated bytes does not contribute directly to reducing the overall program execution time. It may, however, contribute indirectly, by reducing the garbage collection overhead.

9.2.2 Garbage collection time

GC time, G, as a % of T is rather variable between the tests, ranging from 3–16%. These are on the low side of average for G as a % of T (see section 3).

Encouragingly, G as a % of T reduces in each optimization, but only by a small amount (about 1%).

The GC time reduction, ΔG, is arguably uniform at 9–21% (say 15%); that is quite respectable. However, since G is only, say, 10% of T, ΔG is a disappointingly small fraction of the total program execution time, *about 1.5%*.

9.2.3 Total execution time

If the cell allocation and cache penalty overheads did not change as a result of optimization, then $\Delta T(\%)$ would be identical to $\Delta G(\%$ of $T)$. We have assumed that the cell allocation costs do not change, and that the cache costs might.

ΔT is reasonably uniform, at roughly 4.5%; that is not unrespectable, but, equally, is not dramatic. On the other hand, $\Delta G(\%$ of $T)$ is reasonably uniform at about 1.5%. Therefore, the reduction in total execution is due only in part to improved storage management.
(Although the garbage collection times themselves are calculated from average copying rates, rather than being measured, and thus may not be reliable, it is unlikely that there is an error of a factor of 3 in the estimation of the copying rate.)

9.2.4 Cache penalty reduction

According to the performance model in section 8, the difference between ΔT and ΔG is a change in the cache penalty overhead.

Thus, in the final line of Table 3, we see an estimated reduction in cache penalty of about 3% (of T). Once again, this is respectable but not dramatic: locality effects do not appear to be significant in these tests.

9.2.5 Discussion

Table 3 shows a reasonable degree of uniformity over the three tests. There are fluctuations in the benefits derived from the optimizations. We cannot tell with such a small population of tests whether or not there is a trend. If there is a trend then it is that the longer an execution is, the less benefit is derived from optimization: test 3 has smaller improvements in most respects than the other tests.

As described in section 3, the LML heap adapts its size at each GC to be a multiple of the number of bytes collected. Test 1 required a small heap extension, test 2 required no extension, and test 3 required a large extension. The fact that test 3 needed a heap 4 times as large as the others may explain why it derived less cache penalty benefit from the (potentially) improved locality.

One would expect the largest benefits to show up when a program is running in a "tight" heap—since then the total GC time will be a larger fraction of the total time, and any GC reduction would thus be more significant. However, the LML adaptable heap prevents this effect from being observed.

There is an interesting lack of correlation between the reduction in GC overhead when recorded as bytes copied and as number of GCs. In test 1 the number of GCs before and after optimization is the same, and 17% fewer cells are copied (so the time taken is 17% less). In test 2 the GCs drop from 9 to 8, with 21% less copying. In test 3 the GCs drop from 61 to 57, but there is only 9% less copying. This is probably due to the effects discussed in section 4.

10 Conclusions

The results from the investigation reported here are easy to summarise. For the Conway package, and the given test programs, analysed using the given sharing domains, and executed using eager LML:

- The re-use optimization is applicable to 7 out of 34 functions.
- This produces about an 8% reduction in the number of cell allocations made *from the heap*,
- and a corresponding reduction in the GC (time) overhead of about 15%.
- GC time is a relatively small component of the total execution time, about 10%, and so the total time is reduced by only about 1.5% by optimized GC.
- There appears to be an additional reduction in total time of about 3%. It is hypothesized that this is due to improved cache performance as a result of improved locality of memory accesses.

What we should conclude from the results is less clear.

- It is disappointing that only 7 out of 34 functions are optimizable, and that, although they are reasonably key functions, they are not sufficiently central to the execution to produce more than an 8% reduction in the cell allocations from the heap.

- On the other hand, it was not clear, at the start of the investigation, that there would be *any* optimizations possible at all in a package with this much interdependence between functions. One purpose of the investigation was to discover whether re-use optimization is applicable to a 'real' package, so it is a welcome *positive* result that a reasonable reduction in heap usage is indeed possible.

- Perhaps the most important result is the negative observation that the total execution time is not reduced by much, despite a reasonable reduction in the GC time itself. Since we cannot escape this "multiplication of %ages", the only way to achieve a large reduction in total execution time is to have both GC time being a very large fraction of the total time, and a very large reduction in GC time. The former will be true if a program is executing in a small heap—this may be enforced by the environment, but it is not a situation that we would naturally choose to be in. The latter is determined purely by the program to be optimized, and is outside our control, except to the extent that we must use the most accurate (detailed) sharing domains and analysis that we can.

If the overall improvements discovered in this investigation are typical of most 'real' functional programs, then their optimization appears *not worthwhile* unless either the sharing analysis and optimization are very cheap to carry out (which they probably are not), or it is known that the program will have to run with a small heap.

There remains the open question of *whether* the degree of optimization possible with the Conway package is typical of most 'real' functional programs. This is perhaps the most important piece of further work in this particular area.

References

[AJ] L. Augustsson and T. Johnsson. *Lazy ML Users' Manual (Draft)*. Department of Computer Science, Chalmers University of Technology. (March 1992 version.).

[App87] A.W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, June 1987.

[Coh81] J. Cohen. Garbage collection of linked data structures. *Computing Surveys*, 13(3):341–367, September 1981.

[Con76] J.H. Conway. *On Numbers and Games*. Academic Press, 1976.

[Hud87] P. Hudak. A semantic model of reference counting and its abstraction. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 45–62. Ellis Horwood, 1987.

[JLM89] S.B. Jones and D. Le Métayer. Compile-time garbage collection by sharing analysis. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 54–74, 1989.

[Tya93] A.S. Tyas. An investigation into the optimization of garbage collection within functional languages. Final Year Dissertation, Department of Computing Science and Mathematics, University of Stirling, April 1993.

A The Conway number package in Haskell

The implementation of Conway numbers comprises two Haskell modules: a general purpose `Sets` module, and the `Conway` module itself.

```
module Sets(Set,

            empty,isempty,
            sing,quickset,
            union,inter,member,diff,
            bigunion,
            eqset,
            mapset,filterset,
            maxinset,mininset,

           ) where

data Eq a => Set a = MkSet [a]

empty :: Eq a => Set a
empty = MkSet []

isempty :: Eq a => Set a -> Bool
isempty (MkSet []) = True
isempty s          = False

sing :: Eq a => a -> Set a
sing x = MkSet [x]

quickset :: Eq a => [a] -> Set a
quickset = foldr addelement empty

union :: Eq a => Set a -> Set a -> Set a
union (MkSet []) s2 = s2
union (MkSet (x:xs)) s2 = addelement x (union (MkSet xs) s2)

inter :: Eq a => Set a -> Set a -> Set a
inter (MkSet []) s2 = empty
inter (MkSet (x:xs)) s2 | member s2 x
                                = addelement x (inter (MkSet xs) s2)
                        | True = inter (MkSet xs) s2

member :: Eq a => Set a -> a -> Bool
member (MkSet xs) x = x `elem` xs

diff :: Eq a => Set a -> Set a -> Set a
diff (MkSet []) s2 = empty
diff (MkSet (x:xs)) s2 | member s2 x
                               = diff (MkSet xs) s2
                       | True = addelement x (diff (MkSet xs) s2)

bigunion :: Eq a => (Set (Set a)) -> Set a
bigunion (MkSet ss) = foldr union empty ss

addelement :: Eq a => a -> Set a -> Set a
addelement x s@(MkSet xs) | x `elem` xs = s
                          | True        = MkSet (x:xs)

eqset :: Eq a => Set a -> Set a -> Bool
```

```
eqset s1 s2 = isempty (diff s1 s2) && isempty (diff s2 s1)

mapset :: (Eq a,Eq b) => (a->b) -> Set a -> Set b
mapset f (MkSet xs) = quickset (map f xs)

filterset :: Eq a => (a->Bool) -> Set a -> Set a
filterset p (MkSet xs) = MkSet (filter p xs)

maxinset :: Ord a => Set a -> a
maxinset (MkSet xs) = foldr1 max xs

mininset :: Ord a => Set a -> a
mininset (MkSet xs) = foldr1 min xs

instance Eq a => Eq (Set a) where

  (==) = eqset


instance (Text a, Eq a) => Text (Set a) where

  showsPrec d (MkSet xs) r = "{" ++ convert xs ++ "}" ++ r
                                          -- Ignore precedence level
      where
            convert [] = ""
            convert [x] = show x
            convert (x:xs) = shows x (", " ++ convert xs)


------------------------------------------------------------

module Conway(CNum,

              buildCNum,buildlitCNum,checkCNum,

              add,mult,neg,sub,
              lesseq,eq,less,

              zero,one,minus_one,two,minus_two

             ) where


import Sets

data CNum = MkCNum (Set CNum) (Set CNum)

two = one `add` one
one = MkCNum (sing zero) empty
zero = MkCNum empty empty
minus_one = zero `sub` one
minus_two = minus_one `sub` one

specials :: [CNum]
specials = [two,one,zero,minus_one,minus_two]    -- for really special names

names :: [String]
names = ["two","one","zero","-one","-two"]
```

```
specialnumber :: CNum -> Bool
specialnumber cn = cn `elem` specials

convertspecial :: CNum -> String
convertspecial cn | cn `elem` specials
                      = names !! (length (takeWhile (/= cn) specials))

buildCNum :: Set CNum -> Set CNum -> CNum   -- build 'simplest'
buildCNum s1 s2 | constraint_OK = MkCNum  l r
                | True          = error "Attempt to build invalid Conway number"
                  where constraint_OK = isempty s1 || isempty s2 || not (s2' `lesseq` s1')
                        s1' = maxinset s1
                        s2' = mininset s2
                        l | isempty s1 = s1
                          | True       = sing s1'
                        r | isempty s2 = s2
                          | True       = sing s2'

checkCNum :: Set CNum -> Set CNum -> Bool   -- Validity constraint check
checkCNum s1 s2 = isempty (filterset
                             (\x2 -> not (isempty (filterset (lesseq x2) s1)))
                             s2)

buildlitCNum :: Set CNum -> Set CNum -> CNum    -- build 'literally'
buildlitCNum s1 s2 | constraint_OK = MkCNum s1 s2
                   | True          = error "Attempt to build invalid Conway number"
                  where constraint_OK = checkCNum s1 s2


lesseq :: CNum -> CNum -> Bool
cn1@(MkCNum cn1l _) `lesseq` cn2@(MkCNum _ cn2r)
           = isempty (filterset (lesseq cn2) cn1l)
             &&
             isempty (filterset ((flip lesseq) cn1) cn2r)

eq :: CNum -> CNum -> Bool
cn1 `eq` cn2 = (cn1 `lesseq` cn2) && (cn2 `lesseq` cn1)

less :: CNum -> CNum -> Bool
cn1 `less` cn2 = not(cn2 `lesseq` cn1)

add :: CNum -> CNum -> CNum
cn1@(MkCNum cn1l cn1r) `add` cn2@(MkCNum cn2l cn2r)
      =    buildCNum (union (mapset ((flip add) cn2) cn1l)
                            (mapset (add cn1) cn2l))
                       (union (mapset ((flip add) cn2) cn1r)
                            (mapset (add cn1) cn2r))

neg :: CNum -> CNum
neg (MkCNum cnl cnr) = buildCNum (mapset neg cnr) (mapset neg cnl)

sub :: CNum -> CNum -> CNum
cn1 `sub` cn2 = cn1 `add` (neg cn2)

mult :: CNum -> CNum -> CNum
cn1@(MkCNum cn1l cn1r) `mult` cn2@(MkCNum cn2l cn2r)
       = buildCNum (union ll rr) (union lr rl)
```

```
        where ll = bigunion (mapset (\x1l -> mapset (\x2l -> f x1l x2l)
                                                    cn2l)
                                    cn1l)
              rr = bigunion (mapset (\x1r -> mapset (\x2r -> f x1r x2r)
                                                    cn2r)
                                    cn1r)
              lr = bigunion (mapset (\x1l -> mapset (\x2r -> f x1l x2r)
                                                    cn2r)
                                    cn1l)
              rl = bigunion (mapset (\x1r -> mapset (\x2l -> f x1r x2l)
                                                    cn2l)
                                    cn1r)
              f x1 x2 = (x1 `mult` cn2) `add` (cn1 `mult` x2) `sub` (x1 `mult` x2)


instance Text CNum where

  -- Ignore precedence d

  -- Numbers we have special names for:
  showsPrec d cn r | specialnumber cn = convertspecial cn

  -- Now the general case:
  showsPrec d (MkCNum ls rs) r
        = "{" ++ convert ls ++ " | " ++ convert rs ++ "}" ++ r
          where
                convert s = init (tail (show s))      -- strip off { and }

instance Eq CNum where

  (==) = eq

instance Ord CNum where

  (<=) = lesseq
```

B The converted Conway number package in LML

This LML module is a first order version of the Haskell `Conway` and `Set` modules. All laziness dependence has been removed; likewise dependence on the LML function library.

```
module

infixl "add";
infixl "sub";
infixl "lesseq";
infixl "less";
infixl "eq";
infixl "mult";
nonfix "addelement";
infixr "@";
nonfix "add_one";

export buildCNum,
       sing, quickset,
       zero,
       add_one, add, sub, mult, eq, neg, lesseq,
```

```
       show_CNum,
       @;

-------------------------------------------------------------
--------------- Set definitions -----------------------------

rec

    type SetCNum = MkSet (List CNum)

and isempty :: SetCNum -> Bool
and isempty (MkSet []) = true
 || isempty s          = false

and sing :: CNum -> SetCNum
and sing x = MkSet [x]

and quickset :: List CNum -> SetCNum
and quickset [] = MkSet []
 || quickset (x.xs) = addelement x (quickset xs)

and union :: SetCNum -> SetCNum -> SetCNum
and union (MkSet []) s2 = s2
 || union (MkSet (x.xs)) s2 = addelement x (union (MkSet xs) s2)

and member :: SetCNum -> CNum -> Bool
and member (MkSet []) cn = false
 || member (MkSet (x.xs)) cn = x eq cn | member (MkSet xs) cn

and addelement :: CNum -> SetCNum -> SetCNum
and addelement x (s as (MkSet xs)) & (member s x) = s
 || addelement x (MkSet xs)                       = MkSet (x.xs)

and maxinset :: SetCNum -> CNum
and maxinset (MkSet [x]) = x
 || maxinset (MkSet (x.xs)) =  max x (maxinset (MkSet xs))

and max x y & (x lesseq y) = y
 || max x y = x

and mininset :: SetCNum -> CNum
and mininset (MkSet [x]) = x
 || mininset (MkSet (x.xs)) = min x (mininset (MkSet xs))

and min x y & (x lesseq y) = x
 || min x y = y

-------------------------------------------------------------
--------------- Conway number definitions -------------------

and type CNum = MkCNum (SetCNum) (SetCNum)

and show_CNum :: CNum -> String
and show_CNum (MkCNum (MkSet ls) (MkSet rs))
            = ('{'.[]) @ convertCNums ls @ (' '.'|'.' '.[])
                       @ convertCNums rs @ ('}'.[])

and show_SetCNum :: SetCNum -> String
```

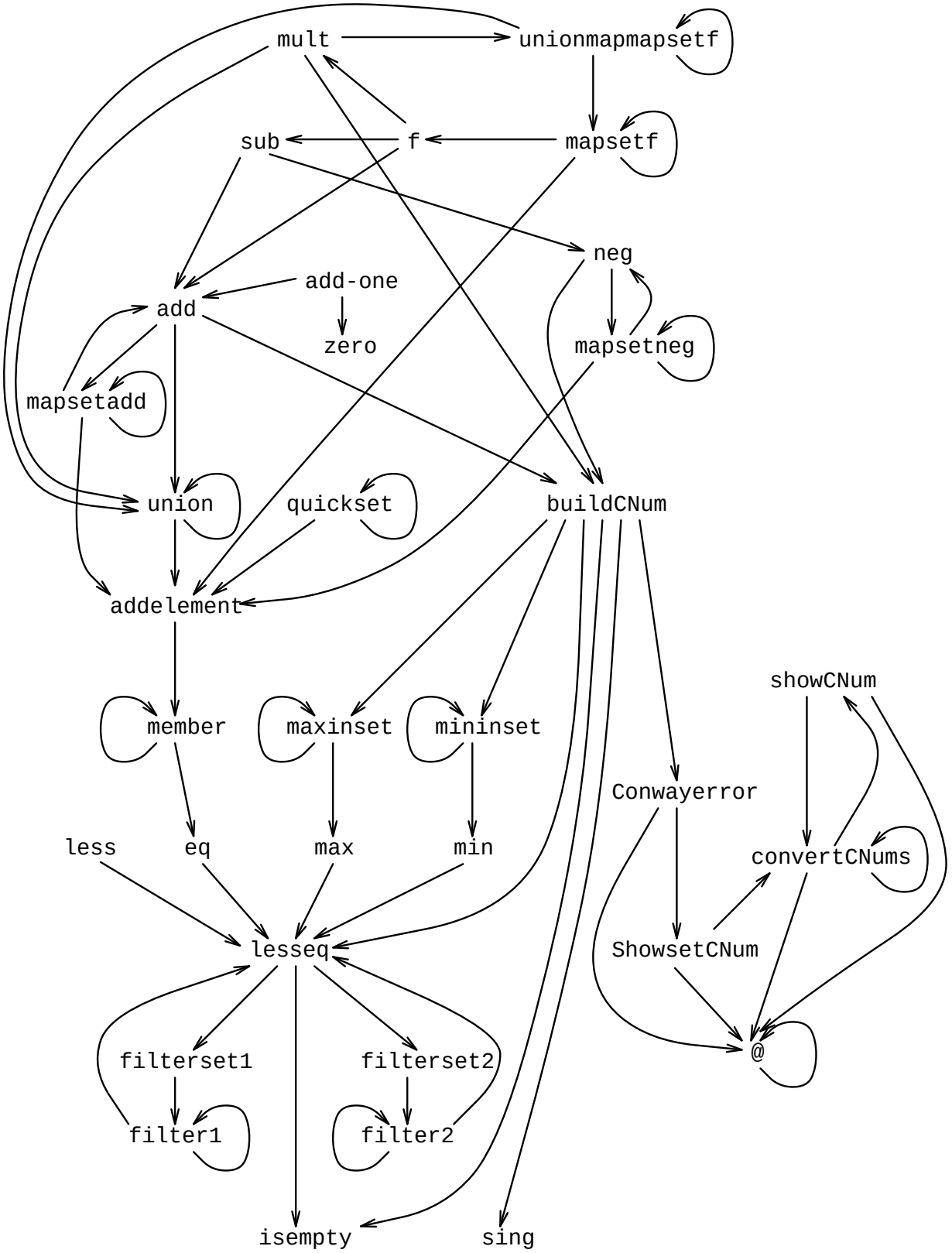
```
and show_SetCNum (MkSet s) = ('{'.[]) @ convertCNums s @ ('}'.[])

and convertCNums :: List CNum -> String
and convertCNums [] = []
 || convertCNums [x] = show_CNum x
 || convertCNums (x.xs) = (show_CNum x) @ (',' '.[]) @ convertCNums xs

-------------------------------------------------------

and conway_error :: SetCNum -> SetCNum -> String
and conway_error l r =
     ('A'.'t'.'t'.'e'.'m'.'p'.'t'.' '.'t'.'o'.' '.'b'.'u'.'i'.'l'.'d'.' '.
      'i'.'n'.'v'.'a'.'l'.'i'.'d'.' '.'C'.'o'.'n'.'w'.'a'.'y'.' '.
      'n'.'u'.'m'.'b'.'e'.'r'.'\n'.[])
      @ show_SetCNum l @ ('\n'.[])
      @ show_SetCNum r @ ('\n'.[])


and buildCNum :: SetCNum -> SetCNum -> CNum   -- build 'simplest'
and buildCNum s1 s2 = case (isempty s1,isempty s2) in
                          (true,true)   : MkCNum s1 s2
                       || (true,false)  : let r = sing (mininset s2)
                                          in MkCNum s1 r
                       || (false,true)  : let l = sing (maxinset s1)
                                          in MkCNum l s2
                       || (false,false) :
                            let s1' = maxinset s1
                            and s2' = mininset s2 in
                            let l = sing s1'
                            and r = sing s2'
                            in if ~(s2' lesseq s1') then MkCNum l r
                                                    else fail (conway_error s1 s2)
                     end

---------
and (lesseq) :: CNum -> CNum -> Bool
and (cn1 as (MkCNum cn1l _)) lesseq (cn2 as (MkCNum _ cn2r))
              = isempty (filterset1 cn2 cn1l)
                &
                isempty (filterset2 cn1 cn2r)

and filterset1 :: CNum -> SetCNum -> SetCNum
and filterset1 cn2 (MkSet xs) = MkSet (filter1 cn2 xs)
and filter1 :: CNum -> List CNum -> List CNum
and filter1 cn2 [] = []
 || filter1 cn2 (x.xs) = if cn2 lesseq x then x.filter1 cn2 xs
                                         else filter1 cn2 xs
and filterset2 :: CNum -> SetCNum -> SetCNum
and filterset2 cn1 (MkSet xs) = MkSet (filter2 cn1 xs)
and filter2 :: CNum -> List CNum -> List CNum
and filter2 cn1 [] = []
 || filter2 cn1 (x.xs) = if x lesseq cn1 then x.filter2 cn1 xs
                                         else filter2 cn1 xs

---------
and zero [] = MkCNum (MkSet []) (MkSet [])

and (eq) :: CNum -> CNum -> Bool
```

```
and cn1 eq cn2 = (cn1 lesseq cn2) & (cn2 lesseq cn1)

and (less) :: CNum -> CNum -> Bool
and cn1 less cn2 = ~(cn2 lesseq cn1)

---------
and (add) :: CNum -> CNum -> CNum
and (cn1 as (MkCNum cn1l cn1r)) add (cn2 as (MkCNum cn2l cn2r))
          =    buildCNum (union (mapsetadd cn2 cn1l)
                                (mapsetadd cn1 cn2l))
                         (union (mapsetadd cn2 cn1r)
                                (mapsetadd cn1 cn2r))

and mapsetadd :: CNum -> SetCNum -> SetCNum
and mapsetadd cn (MkSet []) = MkSet []
 || mapsetadd cn (MkSet (x.xs)) = addelement (cn add x) (mapsetadd cn (MkSet xs))

---------
and add_one :: CNum -> CNum
and add_one cn = cn add (MkCNum (MkSet [(zero [])]) (MkSet []))

---------
and neg :: CNum -> CNum
and neg (MkCNum cnl cnr) = buildCNum (mapsetneg cnr) (mapsetneg cnl)

and mapsetneg :: SetCNum -> SetCNum
and mapsetneg (MkSet []) = MkSet []
 || mapsetneg (MkSet (x.xs)) = addelement (neg x) (mapsetneg (MkSet xs))

---------
and (sub) :: CNum -> CNum -> CNum
and cn1 sub cn2 = cn1 add (neg cn2)

---------
and (mult) :: CNum -> CNum -> CNum
and (cn1 as (MkCNum cn1l cn1r)) mult (cn2 as (MkCNum cn2l cn2r))
          = let ll = unionmapmapsetf cn1 cn2 cn2l cn1l
            and rr = unionmapmapsetf cn1 cn2 cn2r cn1r
            and lr = unionmapmapsetf cn1 cn2 cn2r cn1l
            and rl = unionmapmapsetf cn1 cn2 cn2l cn1r
            in buildCNum (union ll rr) (union lr rl)

and f :: CNum -> CNum -> CNum -> CNum -> CNum
and f cn1 cn2 x1 x2 = (x1 mult cn2) add (cn1 mult x2) sub (x1 mult x2)

and mapsetf :: CNum -> CNum -> CNum -> SetCNum -> SetCNum
and mapsetf cn1 cn2 cn (MkSet []) = MkSet []
 || mapsetf cn1 cn2 cn (MkSet (x.xs))
          = addelement (f cn1 cn2 cn x) (mapsetf cn1 cn2 cn (MkSet xs))

and unionmapmapsetf :: CNum -> CNum -> SetCNum -> SetCNum -> SetCNum
and unionmapmapsetf cn1 cn2 scn (MkSet []) = MkSet []
 || unionmapmapsetf cn1 cn2 scn (MkSet (x.xs))
          = union (mapsetf cn1 cn2 x scn) (unionmapmapsetf cn1 cn2 scn (MkSet xs))

-------------------------------------------------
-------------- Basic functions ------------------------
```

```
and (@) :: List *a -> List *a -> List *a
and [] @ ys = ys
 || (x.xs) @ ys = x.xs @ ys

end
```

Calling diagram for Conway package

An arrow from `a` to `b` indicates that `a` contains one or more references to `b`.



C Test main programs

Test 1

```
#include "conway.t";

let one = add_one (zero []) in
let two = add_one one in
let three = add_one two in

(show_CNum (two mult three)) @ "\n"
```

Test 2

```
#include "conway.t";

let one = add_one (zero []) in
let half = buildCNum (sing (zero [])) (sing one) in
let oneandahalf = one add half in

(show_CNum (oneandahalf add oneandahalf add oneandahalf add oneandahalf)) @ "\n"
```

Test 3

```
#include "conway.t";

let one = add_one (zero []) in
let two = add_one one in
let oneandahalf = buildCNum (sing one)(sing two) in


(show_CNum ((two mult oneandahalf) add two)) @ "\n"
```