Region-Based Memory Management in Perspective

Mads Tofte (tofte@itu.dk) and Niels Hallenberg (nh@itu.dk)
The IT University of Copenhagen
Invited Talk presented at the Space 2001 Workshop, Imperial College, London, Jan. 2001

1 Introduction

Originally, Region-based Memory Management was conceived as a purely theoretical idea intended to solve a practical problem in the context of Standard ML, namely inventing a dynamic memory management discipline which is less space demanding and more predictable than generic garbage collection techniques, such as generational garbage collection.

Over the subsequent eight years, considerable effort has been devoted to refining the idea, proving it correct and finding out whether it worked in practice. The practical experiments revealed weaknesses, which led to new program analyses, new theory and yet more experimentation. In short, we have sought to work on memory management as an experimental science: the work should be scientific in the sense that it should rest on a solid mathematical foundation and it should be experimental in the sense that it should be tested against competing state-of-the-art implementation technology.

The purpose of this paper is to step back and consider the process as a whole. We first describe the main technical developments in the project, with an emphasis of what motivated the developments. We then summarise what we think has gone well and what has not gone so well leading to suggestions as to what one might do next and some thoughts on what we have learned about the interaction between theory and practice during the project.

2 First attempts

In the late eightees, Standard ML of New Jersey was becoming the most sophisticated Standard ML compiler. While it generated fast code, it seemed to require an inordinate amount of space. The first author had for many years been fascinated by the beauty of the Algol stack and somewhat unhappy about the explanations of why, in principle, something similar could not be done for the call-by-value lambda calculus. (These explanations typically had to do with "escaping functions" and other concepts that did not feel canonical.)

Surely, if an expression has type `int` (and there are no effects in the language) then all the memory that is allocated during the computation of the integer (except for the memory needed to hold the result) can be de-allocated once the result has been computed. This was in contract to the way memory was used with garbage collectors (at the time), namely linear allocation in memory until memory becomes too full. The beauty of the stack discipline is that it uses memory only proportional to the depth of the call stack, whereas with garbage collection, one allocates memory as if one is trying to put the entire call tree in memory. (The call stack may require only the logarithm of the space required to represent the call tree.)

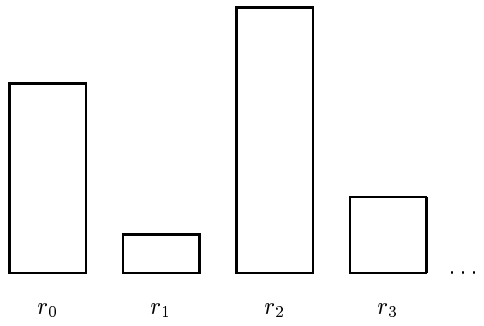

Figure 1: The store is a stack of regions; a region is a box in the picture.

In 1992, Talpin and Jouvelot published a type discipline for polymorphic references which used an effect system for controlling quantification of type variables [TJ92]. Like earlier work on effect systems, their system involved a notion of region (of references). Tofte suggested to Talpin that the approach may be generalised from dealing with references to accounting for allocation and de-allocation of all values in the call-by-value lambda calculus. Together, Tofte and Talpin developed a basic region inference system for a toy language based on ML [TT92]. It had no recursive functions.

At runtime, the store consists of a stack of regions, see Figure 1. All values, including function closures, are put into regions.

Every well-typed source language expression, e, can be translated into a region-annotated expression, e', which is identical with e, except for certain region annotations. The evaluation of e' corresponds, step for step, to the evaluation of e. Two forms of annotations are

$$e_1 \text{ at } \rho$$
$$\texttt{letregion } \rho \text{ in } e_2 \text{ end}$$

The first form is used whenever e_1 is an expression which directly produces a value. (Constant expressions, λ-abstractions

and tuple expressions fall into this category.) The ρ is a *region variable*; it indicates that the value of e_1 is to be put in the region bound to ρ.

The second form introduces a region variable ρ with local scope e_2. At runtime, first an unused region, r, is allocated and bound to ρ. Then e_2 is evaluated (probably using r). Finally, r is de-allocated. The `letregion` expression is the only way of introducing and eliminating regions. Hence regions are allocated and de-allocated in a stack-like manner.

The translation from the source language to the language of region-annotated terms was formalised by a set of formal inference rules, the *region inference rules*, that allowed one to infer conclusions of the form

$$TE \vdash e \Rightarrow e' : (\tau, \rho), \varphi$$

read: in the type environment TE, the source expression e may be translated into a region-annotated expression e' which has type τ, is placed in region ρ and has effect φ, where, slightly simplified, an effect is a finite set of region variables.

There was a proof of correctness with respect to a standard operational semantics, a region inference algorithm and a proof of existence of principal types and minimal effects.

Tofte built a prototype implementation of a slightly larger toy language with recursive functions, pairs and lists. The implementation contained a different region inference algorithm and an instrumented interpreter for region-annotated terms. Experimental results were terrible. There seemed to be two main causes:

1. When a function f, say, returns a result in a region, then all calls of f must return their result in the same region. Thus the region must be kept alive until no result of f was needed (which is very conservative)

2. In particular, when a function calls itself recursively, the result of the recursive call must be put in the same region as the result of the function (even in cases where the recursive call produces a result which is not part of the result of the function).

The solution to the first problem was straightforward: functions should be allowed to take region parameters at run time. Talpin observed a beautiful connection between region parameters and quantified region variables in type schemes: a function of type

$$\forall \rho_1, \ldots, \rho_k \alpha_1, \ldots, \alpha_n . \tau \rightarrow \tau'$$

should take $\rho_1, \ldots, \rho_k$ as formal region parameters. The actual parameters at a call of f are the result of instantiating the type scheme to region variables at the call site. A function is *region-polymorphic*, if it takes regions as parameters.

Thus two more forms of region annotations were introduced in region-annotated terms, one for declaring (recursive) region-polymorphic functions

$$\texttt{letrec}\ f[\rho_1, \ldots, \rho_k](x) = e_1\ \text{in}\ e_2$$

and one for referring to them:

$$f\,[\rho'_1, \ldots, \rho'_k]$$

3 Polymorphic recursion

Tofte noticed that what was required to solve the second problem mentioned above (recursive functions) was polymorphic recursion in regions. For example, consider the source expression:

```
letrec fac(n)=
  if n=0 then 1
  else n*fac(n-1)
in fac 100
```

Translating this without polymorphic recursion would give

```
letrec fac[r1](n)=
  if n=0 then 1 at r1
  else (n*fac[r1](n-1))at r1
in fac[r0]100
```

which causes 100 values to pile up in the region `r0`. With polymorphic recursion, however, one could translate the expression into

```
letrec fac[r1](n)=
    if n=0 then 1 at r1
    else
      letregion r2
      in (n*fac[r2](n-1))at r1
      end
in fac[r0]100
```

so that each recursive invocation uses its own (local) region.

However, there were still problems with recusion in the special case of tail recursion and iteration. For example, consider the following program, which is intended to sum the numbers from 1 to 100 (`fst` and `snd` stand for the first and second projection of pairs, respectively):

```
letrec sumit(p: int*int)=
  let acc= fst p
  in let n  = snd p
     in
       if n=0 then p
       else
         sumit(n+acc, n-1)
in
  fst(sumit(0,100))
```

At first sight, the recursive call of `sumit` in its own body is a tail call. Moreover, since the two branches of the conditional must put their results in the same region, `sumit` delivers its result pair in the same region as its argument resides. But as a result, 100 pairs will pile up in the region that contains the initial pair (0,100). (It would not improve matters to use polymorphic recursion: this would just give 100 pairs in 100 different regions on the stack.) What one would like to do is to have the pair (n+acc, n-1) *overwrite* the pair `p`, since - in this program - `p` is not used after (n+acc, n-1) has been created.

To achieve this, Birkedal and Tofte devised a so-called *storage mode analysis*, which allows the compiler to generate code to reset regions prior to an allocation, when the analysis can conclude, that the region contains no live value.

The proof of correctness of the region inference rules was extended to deal with region polymorphism and polymorhic region recursion. The region inference algorithm was extended to deal with polymorphic recursion by iterative region inference of the recursive function till a fixed point type

scheme was obtained. Ad-hoc methods were used in order to ensure termination. These ad-hoc methods also made it clear, that there was no guarantee that the algorithm would find most general type schemes for region-polymorphic functions. The results were presented at POPL '94 [TT94].

The experimental results for runtime space usage varied from the excellent to the poor. Excellent results were obtained for programs that were written iteratively or had a natural stack-like behaviour. Good results were obtained for quicksort and other small, classical algorithms. Poor results for programs where lifetimes just are not nested or where there is extensive use of higher-order functions.

Encouraging facts at this stage were:

1. There are programs for which the region scheme works extremely well (even without any other form of garbage collection)!
2. Soundness of memory use is guaranteed
3. Memory behaviour is explicated and can be studied if one cares sufficiently about memory as a resource

Worries at this stage:

1. What will happen in big programs? What is the "typical" ratio between parts of the program for which region inference works well and the parts for which it does not work well? How difficult and time consuming is it to rewrite programs to make them region friendly?
2. The good experimental results were based on an instrumented, inefficient interpreter. Actual runtime performance was nowhere near what compilers like SML/NJ could deliver. Could regions be managed efficiently at runtime, or would administrative overhead at runtime be prohibitive?
3. Unclarity about principal types: looked like a tough problem, even soundness of the algorithm was no longer easy!
4. Soundness of region inference rules was getting complicated, although do-able!
5. Experimental results depended on the storage mode analysis, which had not been described and studied independently of the implementation.

Of these questions, the two first seemed the most important to address. In the long run, who would care about principal types and the correctness of extra analyses, if the overall scheme did not work in practice?

There seemed to be only one way to find out whether this scheme could ever become a serious contender to the much more mature garbage collection techniques that were already present in many implementations. We would have to build a real language implementation based on region inference and compare it to other implementations.

We decided to aim at doing region inference for full Standard ML. There were pragmatic reasons for choosing Standard ML as the source language: we already knew the language in detail and we had a Standard ML front-end which was compliant with the language semantics, namely the ML Kit, originally developed at Edinburgh University [BRTT93]. Moreover, there existed several sophisticated SML compilers that could be used for comparison and plenty of SML programs that could be used as experimental data. But there was an other very important reason for choosing SML as a source language: it is one thing to propose a new way of implementing programming languages - if one also proposes a new language, there is a danger that the whole enterprise becomes obscure. It seemed much more promising to try to develop the ideas in the context of an existing research community.

4 Aiming for the SML Core Language

The work on extending the ML Kit with regions began in the fall of 1993. Lars Birkedal developed a region-based runtime system, written in C, and a code generator from region-annotated terms to C. Tofte extended the region inference algorithm to cover the SML core language, so it became possible to compile Core Standard ML programs to C-code, which was then compiled using a C-compiler and linked with the runtime system using an ordinary linker. Martin Elsman and Niels Hallenberg subsequently wrote a machine code generator for the HP PA-RISC architecture.

With this system it became possible to compile medium sized test programs (the largest being around 1000 lines of Standard ML) into machine code. The test programs were taken from the SML/NJ benchmark suite.

At first, results were disappointing. Target programs used more space and ran slower than when run under other systems.

However, inspection of the produced code revealed that slow running time likely had to do with unnecessary overhead in managing regions. It seemed that a lot of regions only ever contained one value. Placing such regions on the stack rather than allocating region pages for them might reduce executing times.

Birkedal, Tofte and Vejlstrup then developed and implemented a so-called *multiplicity inference analysis*, the theory of which was developed by Vejlstrup [Vej94]. The idea was to find out, for every region, an upper bound on the number of values that were written into the region.

Initially, the bound could be an integer or infinity (meaning that the analysis could not find any finite bound). Experiments on sample programs revealed that by far the most common case was that the upper bound was 1, the second most common case was that the upper bound was infinity while only very rarely did finite upper bounds other than one appear. Consequently, we simplified the analysis so that it distinguishes between *finite* regions, defined as regions that have a finite upper bound of one value, and *infinite* regions, meaning all other regions. Finite regions are part of the activation record, while an infinite region consists of a linked list of fixed-size region pages, allocated from a free list of region pages, see Figure 2.

Other optimisations included elimination of regions that contain word-sized values only. (Such regions can be kept in machine registers - or spilled onto the stack, in case of lack of registers.)

The effects of incorporating these improvements were astonishing: in many programs, 90% or more of all allocations at runtime were to finite regions. In the largest test program (simple, a 1000 lines program), more than 99% of the allocations were on the stack. Changing the runtime system and code generator to use finite regions led to an improvement in running times of roughly a factor 3.

After these optimisations, the test programs ran between 10 times faster and four times slower under the Kit than

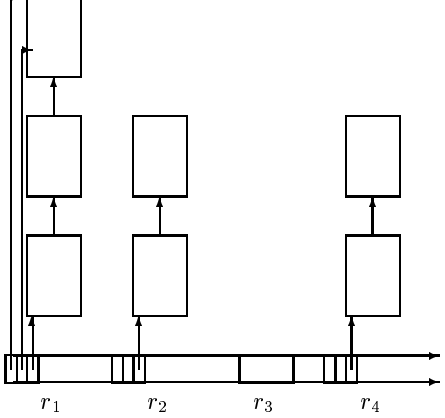

Figure 2: The runtime stack (bottom) contains three infinite regions (r_1, r_2,r_4) and a finite region (r_3). An infinite region is represented by a region descriptor on the stack and a linked list of fixed size region pages. A region descriptor contains a pointer to the last (i.e., most recently allocated) region page, a pointer to the first free position in this region page, and a pointer to the first region page. A finite region is just a number of words on the stack.

under SML/NJ v. 93, which was generally considered the state-of-the-art SML compiler at the time. Space usage for the test programs run under the Kit varied between 8% and over 3000% of the space usage for the same programs run under SML/NJ v. 93, with most of the test programs using considerably less space under the Kit than under SML/NJ, once the test programs had been modified to become "region friendly".

The process of making programs "region friendly" was time consuming and required intimate understanding of the analyses involved (although not much knowledge of the algorithms that implemented the analyses). Basically, the process involved peering at the region-annotated code to see whether the region annotations gave reasonable life-times. Deciding what was reasonable life-times involved understanding the various test programs in some detail. For example, since the Game of Life test program conceptually is an iterative computation of subsequent "generations" of a game board, a reasonable objective was to organize the use of regions so that no more than two generations of the game were live at the same time. Once this objective was achieved, space usage reduced to 376KB, or around one fourth of the space usage of SML/NJ on the same program.

For the largest of the test programs (1000 lines), it was estimated that making a detailed analysis and perhaps rewriting of the program would be too time consuming; fortunately the analyses worked well without any modification of the test program in this case: the program still used less memory under the Kit than under SML/NJ.

So, significant progress:

1. It was possible to extend the entire scheme to all of the Core Language of Standard ML
2. Programs of up to 1000 lines of SML could be executed with speed and space usage comparable to that of SML/NJ
3. Programs that were (re-)written with care could be made to run in significantly less space than under SML/NJ

So, from a purely technical point of view, we felt that region-based implementation had passed its first test with the competition in the practical world. However, concerns about the process of programming with regions were mounting. There were the following problems:

1. Region inference favours a particular discipline of programming. How would one explain this discipline to programmers?
2. Region inference generates a large number of regions and region parameters to region-polymorphic functions. Therefore, region-annotated programs are large and difficult to read.
3. As source programs change, the region annotations change as well. Thus the time invested in understanding the region annotations of one program may be lost, when the source program is modified slightly.
4. Almost all of the region annotations seemed to be fine. If one were faced with an apparent space leak, how would one locate it, other than by studying the entire program?
5. Could region inference be extended to ML Modules?

In short, there was a strong sense that here was a technology which could produce astonishing results, when it worked well, but it was too difficult to hit those precise points where this happens. Moreover, if it is difficult for the people who developed the technology, what would be the chances of success with the average programmer? We felt that we lacked instruments other than the source code and the intermediate forms produced by the compiler to understand the runtime memory behaviour of programs.

5 Profiling and the ML Kit

At this point (Summer 1995), we became aware of the work by Runciman and Wakelin on profiling of Haskell-programs [RW93]. Based on their system, Hallenberg developed a region profiler for the Kit. This was a break-through for our ability to program with regions in practice. Running some of the programs that had been hand-tuned using the profiler resulted in fascinating pictures of memory usage, see Figure 3 for an example.

Also, the profiler made it much easier to locate and eliminate space leaks, i.e., annotations which cause a program to use much more memory than one would expect.

A discipline of programming was emerging. From the peering at the region annotated programs, we had learned a great deal about what works well and what does not work well with regions. The profiler was the tool required to locate space leaks and, more generally, to verify that memory was used as planned.

Hence we decided to try to describe a discipline of programming with regions in a comprehensive report [TBE+97]. It gives a step-by-step introduction to programming using regions, moving from basic values and lists over first-order recursive functions to datatypes, references, exceptions and higher-order functions.

The report was released in april 1997 as part of "The ML Kit, Version 2".[1]

[1] http://www.it-c.dk/research/mlkit/kit2/readme.html

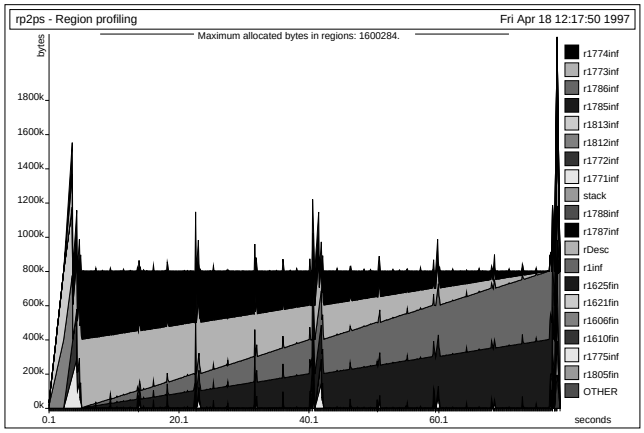

Figure 3: Region profiling of region-optimised mergesort. The two upper trianges contain unsorted elements, while the two lower triangles contain sorted elements.

We also held a summer school on programming with regions[2], consisting of lectures on the theory behind regions and practical programming exercises. Concerning the latter, it was interesting to see how some students became very excited about getting their programs to run in as little memory as they could possibly manage, showing that the technology really does give the programmer a handle on understanding space. It also became clear that programmers found that some of the analyses, especially the storage mode analysis, were unpleasantly complicated.

We felt that we had made good progress on the first and the fourth of the five problems listed at the end of Section 4. The second and the third problem seemed hard to do anything about, without changing the approach to, say, considering explicit region annotations in the source language (which we did not want to do, since this would mean departing from using SML as the source language).

Rather than delving into the design of a new programming language, it was more interesting to work on whether regions could be extended to ML Modules. There were two reasons for this. First, to be able to compile big ML programs, one would need to be able to handle ML Modules. Second, dealing with region inference in some modular fashion was an interesting challenge in itself. Region inference depends on a much finer level of description than the ML type system itself offers. Separate compilation of modules normally requires only type information. To what extent is it possible to compile modules separately using region-based implementation technology?

6 Modules and Separate Compilation

In his Ph.D.-thesis, Martin Elsman [Els99] presented his solution to the problem in the form of a general scheme for propagation of compile-time information across module boundaries, exemplified by a separate compilation system for the ML Kit with Regions. This scheme was used in Version 3 of the ML Kit[TBE+98].

Version 3 of the ML Kit made it possible for the first time to compile large ML programs for a region-based implementation. AnnoDomini, a 58,000 lines SML-program, took 1.5 hours to compile. Running it with the region profiler revealed a couple of space leaks. It was possible to fix these by rewriting around 10 of the 58.000 lines of ML code. Thereafter, AnnoDomini used less space under the Kit than under SML/NJ.

This is a very interesting result, since much of the code in AnnoDomini is written by programmers who do not know how to program with regions - in fact, of the 58.000 lines of SML, more than 10,000 lines were accounted for by a machine generated lexer and parser. On the other hand, it required a regions expert to locate and change the 10 lines. So, progress:

1. It is possible to extend region analyses to all of Standard ML, including modules
2. Proof of concept for large programs: a large ML program was compiled and run using the system
3. Making this large program region friendly required a (surprisingly) small amount of modification to the program.

That the compiler was slow was of course a problem that would require further work, for the technology to become attractive in practice; more of a concern were the things having to do with the way programmers would use regions in practice.

It was always known that there are programs that are just not well suited for region inference. Even if the AnnoDomini experiment suggests that one can get far without performing major revision of the code, it must be a concern for everybody who uses regions that there is no guarantee that one will be able to solve all problems that one encounters within the regions scheme. Clearly, it is one thing to invest time in tuning a program with a view to using regions. This is something one is likely to be willing to do, if one really cares about control over memory resources and the other benefits obtained by region inference (e.g., for real-time programming in embedded systems). It is another thing to invest time without knowing whether, at the very last moment, one will run into a problem which will force one to do major revisions to the code or, in the worst case, give up on regions altogether!

Summing up, the problems to do with how one programs with regions were:

1. Region inference generates a large number of regions and region parameters to region-polymorphic functions. Therefore, region-annotated programs are large and difficult to read.
2. As source programs change, the region annotations change as well. Thus the time invested in understanding the region annotations of one program may be lost, when the source program is modified slightly.
3. What is one to do, if one cannot see how to rewrite the program to use regions more efficiently (or if one can see it, but it would mean an inordinate amount of work)? What about algorithms that just are inherently not well suited for regions?

The solution of the first two problems still seemed to require change of source language, which we were not willing to do. But the third problem could perhaps be addressed by finding a combination of region inference and garbage collection.

[2] See http://www.it-c.dk/research/mlkit/kit2/summerschool.html

Program	t_{gt}	$\#GC_{gt}$	t_{rgt}	$\#GC_{rgt}$
kitlife35u	74.20	5254	39.84	0
kittmergesort	15.27	27	7.36	4
kitqsort	47.89	76	17.68	11
kitreynolds2	13.61	963	10.03	0
kitkbjul9	95.22	3976	47.32	31
kitlife_old	46.22	2832	38.83	43
kitkb_old	191.05	674	57.62	18
kitreynolds3	45.94	4540	24.75	553
professor_game	26.83	3621	15.53	78

Table 1: Using garbage collection without and with region inference.

If successful, a combination of region inference and garbage collection could perhaps even reduce the importance of the first two problems: one might conceivably not have to look at region-annotated programs at all, because garbage collection would handle the space leaks instead.

7 Garbage Collection and Regions

In his M.Sc. thesis [Hal99], Hallenberg developed a scheme for garbage collecting regions and implemented it in the ML Kit.[3]

The scheme consists of a generalisation of Cheney's stop-and-copy copying garbage collection algorithm to apply to regions. Very briefly, the idea is to perform a Cheney copying collection of all regions on the region stack but to do it in such a way that two live values are in the same region before the collection if and only if they are in the same region after the collection. The garbage collector is invoked whenever more than 2/3 of the region pages in the free list have been used.

In the case where there is just one region, the algorithm reduces to (essentially) Cheney's algorithm. Thus one can get a rough idea of the interaction between region inference and garbage collection by comparing what happens when one forces all values to be put in a global region to what happens when region inference is allowed to run its normal course.

Hallenberg conducted this experiment on a number of test programs. The results are shown in Table 1. The first column is the name of the benchmark program. The second and third columns show what happened when all values are put in global regions (so that region inference collects no values at runtime); the second column shows the running time (in seconds), while the third column shows the number of times the garbage collector had to run. The fourth and fifth columns show what happened when region inference is allowed to collect regions in the usual way; the fourth column shows the running times (in seconds) and the fifth column shows the number of times the garbage collector was invoked.[4]

We see that using region inference greatly reduces the number of times, the garbage collector needs to run. Furthermore, using a combination of region inference and garbage collection reduces the running time significantly, compared to using the garbage collector without region inference.

[3] This will become available in Version 4 of the ML Kit, soon to appear!

[4] The "t" in the subscripts stand for "tagging", for reasons that will become apparent below.

Hallenberg also compared the running times in Table 1 to running times (t_r) obtained by using region inference alone, without the garbage collector. For all benchmark programs, the fastest execution was obtained by using region inference without garbage collection (mostly because tags are not necessary, if one does not do garbage collection). So the pattern observed was:

$$t_r < t_{rgt} < t_{gt}$$

Concerning space, programs that had been optimised for regions used up to four times *more* space when running under the combination of region inference and garbage collection than when using region inference only. (This is not surprising, since the garbage collector requires tags and to-spaces.) So for programs that had been optimised for regions, it was best not to add garbage collection, both from the point of view of time and space.

However, programs that had not been optimised for regions all used much less space when run using both the garbage collector and region inference than when using region inference alone. Again, this is not surprising, for programs that have not been optimised for regions often contain some space leaks that makes memory usage linear in the running time. The experiments thus confirmed the hope, that adding a garbage collector to region inference really does take care of the (relatively few) allocations that are not reclaimed well by region inference.

In short: the best results (both concerning time and space) are obtained by optimising programs for regions to the point where they do not need garbage collection at all, but if one does not or cannot do this, garbage collection does take care of the cases, where region inference does a poor job.

An important question remained, however: what was the space usage using garbage collection alone compared to using region inference and garbage collection in combination?

One might think that one would save space by using both region inference and garbage collection compared to using garbage collection alone (since region inference takes care of some of the de-allocation, the garbage collector would need less space to work in). Indeed this effect was observed for programs that had been optimised for regions. But for programs that had not, the opposite happened: it required less space to use the garbage collector alone than using the combination of the garbage collector and region inference. The reason is probably that infinite regions on the region stack result in fragmentation of memory. Many infinite regions containing only a few values each can take up much more space than putting all the values in a global region.

So, unfortunately, it cannot be recommended to add region inference as just an "optimisation" on a basically garbage collected system, if programmers are assumed not to spend time optimising their programs for regions: programs that have not been optimised for regions may well use more space than when using garbage collection only.

On the other hand, if one is willing to spend time optimising programs for regions, the garbage collector provides a fall-back position which one can either use alone or in combination with region inference, depending on how far one gets with optimising the program.

8 Current Beliefs

Things we believe work well:

1. The expressive power of the region inference is fully capable of taking care of the vast majority of memory management that one typically wants to do in a language like SML.

2. The de-allocation that is not done well by region inference can be handled adequately by the garbage collector.

3. Having a proof of soundness of the region inference rules (and the region inference algorithm) gives an unusually high degree of confidence in the memory integrity of compiled programs, even if the proof does not cover all of Standard ML.

4. Learning the discipline of programming with regions is a worthwhile effort, if one is interested in control over memory resources.

5. The technology does scale to complicated language constructs (like ML Modules) and large programs

6. Region-based runtime systems can be small and efficient and the operations they need to perform fit well with both RISC and SISC machines.

7. Finite regions are a very powerful concept. They typically account for the vast majority of allocations at runtime and they can be handled with speed and compactness at runtime.

8. Region profiling is an excellent way of locating and fixing space leaks, except for the fact that region profiling requires inspection of region-annotated terms, which can be verbose.

Things that have disappointed:

1. Leaving region inference completely to the compiler is probably not a good idea. It makes region-annotated terms unnecessary big and vulnerable to program changes.

2. The storage mode analysis was probably not the best way of handling tail recursion; it was too complicated and vulnerable to program changes

3. Infinite regions are perhaps not such a good idea. They give fragmentation problems and there is no natural size of region page to pick. Moreover, they introduce complications throughout the analyses and code generation and the experience with the garbage collector suggests that it is better to use garbage collection for objects that region inference puts into infinite regions, due to fragmentation problems.

9 Future Directions

The general approach taken in the project so far has been to start from Standard ML and then push region inference through a number of program analyses right down to machine language.

What has emerged is that the very heavy employment of automatic program analyses has pragmatic drawbacks and also that the implementation of regions once it gets all the way down to the machine representation becomes somewhat clumsy. On the other hand, as a result of the experimentation,we now know much more about what the strong points of regions are and what parts of the theory and implementation that are candidates for scrapping.

The obvious next step would be to reverse the engineering process. One could start by designing a simple abstract machine that incorporates all the successful features of the region runtime system of the Kit Abstract Machine (and does away with complicated or rarely used features). Then one could try to design a source language which makes it possible for programmers to use the simple abstract machine as their conceptual memory model (much as C programmers have a rough understanding of how the call stack works). This source language should allow (but not force) the programmer to program explicitly with regions. Thus the compiler should perform both region inference and region checking.

All messages that the user needs to understand in order to tune programs must be in terms of concepts and entities that may be written in the source code. This is in order to remove the need for understanding and manipulating intermediate compiler representations. Allowing the user to program directly with regions furthermore addresses the problem that region annotations might change as the program changes, if region annotations are invented by the compiler.

There should only be one infinite region, and it should be garbage collected. Source language syntax (not program analysis) determines whether an allocation is done into the infinite region or into a finite region.

Function calls that are supposed to de-allocate or overwrite the activation record and finite regions of the calling function must be indicated as tail calls in the source code. The compiler must check whether the de-allocation is safe, but it should not try to discover tail calls.

10 Conclusion

One form of interaction between theory and practice is that as one tries to make theory practical, practice produces problems, which one can invent yet more theory to tackle.

But this is perhaps not the most fruitful form of interaction. If practice objects, the reason could be that the theory is too complicated and not that it is in need of further complication or refinement.

Some complexity seems unavoidable - for region inference, polymorphic recursion in regions is a case in point. But when working with theory alone, it is very difficult to know whether some particular expressive capability is important. Our experience has been that one pays for expressive power in the source language or program analyses by a sometimes inordinate amount of further difficult design and implementation choices in the implementation. Other times, one can be fortunate to invent analyses that do just the right thing and work wonderfully well. Only experimentation allows one to tell the difference.

Perhaps the most important power of experimentation and practice is to guide the selection of what expressive power needs to be present in the source language and in the analyses embedded in the compiler. Practice is that wonderful thing that allows us to discard some theory as superfluous, so that we can concentrate on developing and implementing theory that the programmer finds useful.

References

[BRTT93] Lars Birkedal, Nick Rothwell, Mads Tofte, and David N. Turner. The ML Kit (Version 1). Technical Report DIKU-report 93/14, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, 1993.

[Els99] Martin Elsman. *P*rogram Modules, Separate Compilation, and Intermodule Optimisation. PhD thesis, Dept. of Computer Science, University of Copenhagen, 1999.

[Hal99] Niels Hallenberg. Combining garbage collection and region inference in the ml kit. Master's thesis, Dept. of Computer Science, University of Copenhagen, 1999. http://www.itu.dk/research/mlkit/kit_general/papers.html.

[RW93] Colin Runciman and David Wakelin. Heap profiling of lazy functional languages. *J*ournal of Functional Programming, 3(2):217–245, April 1993.

[TBE+97] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, Peter Sestoft, and Peter Bertelsen. Programming with regions in the ML Kit. Technical Report DIKU-TR-97/12, Dept. of Computer Science, University of Copenhagen, 1997. (http://www.diku.dk/research-groups/topps/activities/kit2).

[TBE+98] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, Peter Sestoft, and Peter Bertelsen. Programming with regions in the ML Kit (for version 3). Technical Report DIKU-TR-98/25, Dept. of Computer Science, University of Copenhagen, 1998. (http://www.diku.dk/research-groups/ topps/activities/kit3/manual.ps).

[TJ92] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *J*ournal of Functional Programming, 2(3), 1992.

[TT92] Mads Tofte and Jean-Pierre Talpin. Data region inference for polymorphic functional languages. Manuscript., July 1992.

[TT94] Mads Tofte and Jean-Pierre Talpin. Implementing the call-by-value lambda-calculus using a stack of regions. In *P*roceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 188–201. ACM Press, January 1994.

[Vej94] Magnus Vejlstrup. Multiplicity inference. Master's thesis, Dept. of Computer Science, Univ. of Copenhagen, September 1994. report 94-9-1.