Higher-Order and Symbolic Computation, 17, 245–265, 2004

A Retrospective on Region-Based Memory Management

MADS TOFTE tofte@itu.dk
LARS BIRKEDAL* birkedal@itu.dk
MARTIN ELSMAN mael@itu.dk
NIELS HALLENBERG nh@itu.dk
The IT University of Copenhagen, Denmark

Abstract. We report on our experience with designing, implementing, proving correct, and evaluating a region-based memory management system.

Keywords: dynamic storage management, regions, Standard ML

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 that is less space demanding and more predictable than generic garbage collection techniques, such as generational garbage collection.

Over the subsequent nine years, considerable effort has been devoted to refining the idea, proving it correct, and investigating 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, with an emphasis on what motivated the developments. We then summarise what we think has gone well and what has not gone so well. With these lessons in mind, we suggest some directions for future work and present some thoughts on what we have learned about the interaction between theory and practice during the process.

2. First attempts

In the late 1980s, Standard ML of New Jersey (SML/NJ) was the most sophisticated Standard ML compiler available. While it generated fast code, it did require an inordinate amount of

*Supported in part by STVF Grant No.: 56-00-0309 and SNF Grant No.: 51-00-0315.

space. Tofte had for many years been fascinated by the beauty of the Algol stack discipline 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".) Although, in a theoretical sense, heap allocation could be more efficient than stack allocation (see, e.g., [3]), in practice, the stack discipline could provide better cache locality and less fragmentation than heap allocation (see, e.g., [42]).

Surely, if an expression had type `int` (in a language with no effects) then all memory allocated during the computation of the integer (except for the memory needed to hold the result) could be deallocated once the result had been computed. This observation was in contrast to the way memory was used with garbage collectors (at the time), which allocated memory linearly until memory became too full. The beauty of the stack discipline was that it used memory only proportional to the depth of the call stack, whereas with garbage collection, the program allocated memory as if it needed 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.)

In 1992, Talpin and Jouvelot published a static type discipline for polymorphic references, which used an effect system for controlling quantification of type variables [48]. Like earlier work on effect systems [37–39], their system involved a notion of region (of references). Tofte and Talpin noticed that the approach could be generalised from dealing with references to accounting for allocation and deallocation of all values in the call-by-value lambda calculus. Moreover, Tofte and Talpin developed a basic region inference system for a toy language based on ML [55]. It had no recursive functions.

In the region-based memory model, the store consisted of a stack of regions, as illustrated in Figure 1. All values, including function closures, were put into regions.

Every well-typed source language expression, e, was translated into a region-annotated expression, e', which was identical to e, except for certain region annotations. The evaluation of e' corresponded, step by step, to the evaluation of e. Two forms of annotations were

e_1 `at` ρ

`letregion` ρ `in` e_2 `end`

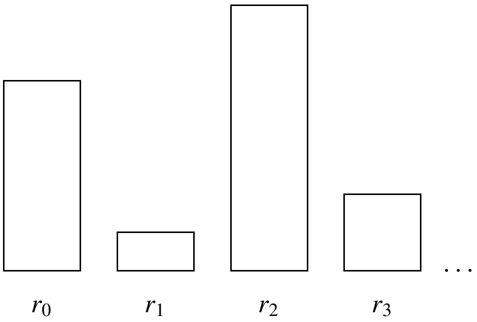

Figure 1. The store consists of a stack of regions; a region is a box in the picture.

The first form was used whenever e_1 was an expression that directly produced a value. (Constant expressions, λ-abstractions and tuple expressions fell into this category.) The annotation `at` ρ indicated that the value of e_1 was to be put in the region bound to the *region variable* ρ.

The second form introduced a region variable ρ with local scope e_2. At runtime, first an unused region, r, was allocated and bound to ρ. Then e_2 was evaluated, probably using r or other regions on the stack. Finally, r was deallocated. The `letregion` expression was the only way of introducing and eliminating regions. Hence regions were 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*, which allowed inference of conclusions of the form

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

Here TE was a *type environment*, which mapped program variables to pairs (σ, ρ) of a type scheme and a region variable. The conclusion was read: "in the type environment TE, the source expression e is 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 was a finite set of region variables. Intuitively, the effect φ contained a superset of the regions needed to be accessed during the evaluation of the expression e.

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 were developed.

Tofte and Birkedal 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, returned a result in a region, then all calls of f had to return their result in the same region. Thus the region had to be kept alive until no result of f was needed (which was very conservative).
2. In particular, when a function called itself recursively, the result of the recursive call had to be put in the same region as the result of the function (even in cases where the recursive call produced a result that was 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 were the result of instantiating the type scheme to region variables at the call site. A function was *region-polymorphic* if it took regions as parameters.

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

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

and one for referring to them:

$$f\,\texttt{[}\rho'_1, \ldots, \rho'_k\texttt{]}$$

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 expression without polymorphic recursion resulted in the region-annotated expression

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

which caused 100 values to pile up in the region `r0`. With polymorphic recursion, however, the expression was translated into the following region-annotated expression:

```
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
```

As a result, each recursive invocation used its own (local) region.

However, even with the presence of region polymorphism, there were still problems with recursion 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` represent the first and second projection of pairs, respectively) and which has a tail call of `sumit` in its own body.

```
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))
```

Region inference would force the two branches of the conditional to put their results in the same region, so (even with polymorphic recursion) `sumit` delivered its result pair in the same region as its argument resided. Thus as a result, 100 pairs would pile up in the region that contained the initial pair `(0,100)`. What would be preferable was to have the pair `(n+acc, n-1)` *overwrite* the pair `p`, because—in this program—`p` is not used after the pair `(n+acc, n-1)` is created.

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

In 1993, the proof of correctness of the region inference rules was extended to deal with region polymorphism and polymorhic recursion for regions. The region inference algorithm was extended to deal with polymorphic recursion by iterative region inference of the recursive function until 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 [57].

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 were obtained for programs where lifetimes were not nested or where higher-order functions were used extensively.

The encouraging facts at this stage were:

1. There were programs for which the region scheme worked extremely well, even without any other form of garbage collection!
2. Soundness of memory use was guaranteed.
3. Memory behaviour was explicated and could be studied by memory conscious programmers.[1]

The worries at this stage were:

1. What would happen for large programs? What was the "typical" ratio between parts of the program for which region inference worked well and the parts for which it did not work well? How difficult and time consuming would it be 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. There was unclarity about the existence of principal types. Even soundness of the region inference algorithm was no longer easy.
4. Soundness of the 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 additional analyses, if the overall scheme did not work in practice?

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

Thus Tofte and Birkedal decided to aim at implementing 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 [6]. Moreover, there existed several sophisticated Standard ML compilers that could be used for comparison and plenty of Standard ML programs that could be used as experimental data. But there was another very important reason for choosing Standard ML as a source language: it is one thing to propose a new way of implementing programming languages—if one at the same time proposes a new programming language, there is a danger that the whole enterprise becomes obscure. Focusing on an already existing programming language, however, would force us to vary as few parameters as possible, which is a key principle of experimental science.

4. Aiming for the Standard ML core language

The work on extending the ML Kit with regions began in the fall of 1993. Birkedal developed a region-based runtime system, written in C, and a code generator from region-annotated terms to C. The runtime system represented regions by linked lists of fixed-size region pages because the size of each region was not known in advance and was potentially unbounded. An interesting point was that values stored in regions did not have to be tagged (as they often were in garbage collected systems). Tofte extended the region inference algorithm to cover most of the Standard ML core language, so it became possible to compile many 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.

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, the results were disappointing. The target programs used more space and ran slower than when the programs were run under other systems. However, inspection of the produced code revealed that slow running times likely had to do with unnecessary overhead in managing regions. It appeared that many regions only ever contained one value. Placing such regions on the stack rather than allocating region pages for them seemed like an obvious possibility for reducing executing times.

Birkedal, Tofte and Vejlstrup then developed and implemented a so-called *multiplicity inference analysis*, the theory of which was developed by Vejlstrup [60]. 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, the analysis was simplified to distinguish between *finite* regions, defined as regions that have a finite upper bound of one value, and *infinite* regions, meaning all other regions. Finite regions were part of the activation record, while an infinite region consisted of a linked list of fixed-size region pages, allocated from a *free-list* of region pages. An example region stack with one finite region (r_3) and three infinite regions (r_1, r_2, and r_4) is shown in Figure 2.

Other optimisations included elimination of regions containing word-sized values only; such regions could be kept in machine registers—or spilled onto the stack, in case of lack of registers. Also, Elsman and Hallenberg wrote a machine code generator for the HP PA-RISC architecture [20].

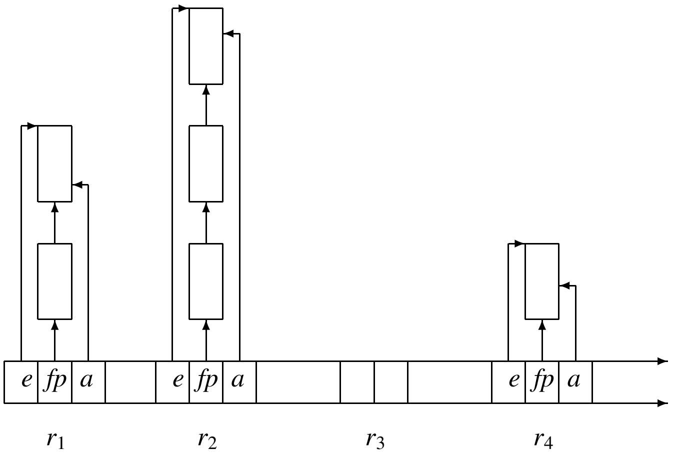

Figure 2. An example runtime stack containing three infinite regions (r_1, r_2, r_4) and a finite region (r_3). An infinite region was represented by a region descriptor on the stack and a linked list of fixed size region pages. A region descriptor was a triple (e, fp, a), where the *allocation pointer a* pointed to the first free position in the last region page, the *end pointer e* pointed to the last position in the last region page, and where the *first page pointer fp* pointed to the first region page. A finite region was represented as a number of words on the stack.

The results of implementing the multiplicity inference and the other optimizations were presented at POPL'96 [8]. The effects of incorporating these improvements were astonishing: in many programs, 90 percent or more of all allocations at runtime were to finite regions. In the largest test program (called simple, a 1000 lines program), more than 99 percent 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 of 3.

After these optimisations, the test programs ran between 10 times faster and four times slower with the ML Kit than with SML/NJ version 0.93, which was generally considered the state-of-the-art Standard ML compiler at the time. Space usage for the test programs run with the ML Kit varied between 8 percent and over 3000 percent of the space usage for the same programs run under SML/NJ version 0.93. For most of the test programs that were modified to be "region friendly" considerably less space was used with the ML Kit than with SML/NJ version 0.93, which was not tailored to run in a small space.

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 lifetimes involved understanding the various test programs in some detail. For example, because 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 was reduced to 376 kilobytes, or around one fourth of the space usage of SML/NJ on the same program. See the manual "Programming with Regions in the ML Kit" [52] for more examples of making programs more region friendly.

For the largest of the test programs (1000 lines), it was estimated that 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 with the ML Kit than with SML/NJ.

The work resulted in 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 Standard ML code could be executed with speed and space usage comparable to that of SML/NJ version 0.93.
3. Programs that were (re)written with care could be made to run in significantly less space than with SML/NJ (version 0.93).

So, from a purely technical point of view, the region-based memory model had passed its first test with the competition in the practical world. However, there were some practical problems with performing the experimental work. Compilation was slow because of region inference and, moreover, because there was no correctness proof for the region inference algorithm used in the ML Kit, debugging the ML Kit compiler was not easy. Thus, Tofte and Birkedal set out to develop provably correct algorithms for region inference. This was not an altogether easy task, mainly because of the presence of polymorphic recursion. The work resulted in two different algorithms for region inference. The first is syntax-directed and based on algorithm W [16] and fixed-point iteration for dealing with polymorphic recursion. It was completed in 1996 and documented in two journal papers [50, 51]. The second algorithm is constraint-based and has the nice property of separating the generation of constraints from the constraint solving. The constraint-based algorithm was completed in 1998 and described in a journal paper [7]. Both algorithms were implemented in the ML Kit and experiments suggested that the constraint-based algorithm was about twice as fast as the syntax-directed algorithm, but that the constraint-based algorithm was considerably more space-consuming. None of the algorithms are complete with respect to the region inference rules, although a restricted form of completeness has been proved for the constraint-based algorithm. The syntax-directed algorithm is the one used by the ML Kit today.

Concerns about the process of programming with regions were also mounting. There were the following problems:

1. Region inference favoured a particular discipline of programming. How would one explain this discipline to programmers?
2. Region inference generated a large number of regions and region parameters to region-polymorphic functions. Thus, region-annotated programs were large and difficult to read.
3. As source programs change, the region annotations changed as well. Thus, the time invested in understanding the region annotations of one program could be lost when the source program was modified slightly.
4. Almost all of the region annotations seemed fine. But given an apparent space leak, how would a programmer locate it, other than by studying the entire program?
5. Could region inference be extended to Standard ML modules?
6. What was a programmer to do if it was not apparent how to rewrite a program to use regions more efficiently (or if it would mean an inordinate amount of work)? And, what about algorithms that simply were not well suited for regions?

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 the results were good. Moreover, if it was 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. Region profiling

At this point (Summer 1995), we became aware of the work by Runciman and Wakeling on profiling of Haskell programs [46]. Based on their system, Hallenberg developed a region profiler for the ML Kit [28]. This profiler was a breakthrough 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, that is, region annotations that cause a program to use much more memory than one would expect.

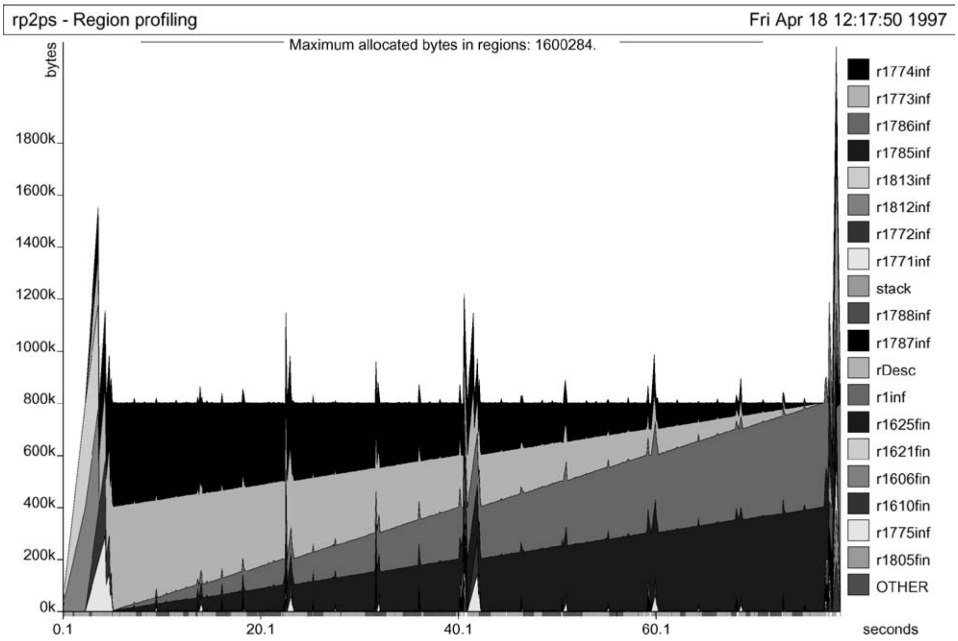

Figure 3. Region profiling of region-optimised mergesort. The two upper triangles contain unsorted elements while the two lower triangles contain sorted elements.

A discipline of programming with regions was emerging. From the peering at the region annotated programs, the authors 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 [53]. The report gave a step-by-step introduction to programming using regions, moving from basic values and lists over first-order recursive functions to data types, references, exceptions, and higher-order functions. The report was released in april 1997 as part of the ML Kit version 2.[2]

We also held a summer school on programming with regions.[3] The summer school covered lectures on the theory behind region-based memory management and practical programming exercises. Concerning the latter, it was interesting to see how students became very excited about getting their programs to run in as little memory as they could possibly manage; this showed that the technology really did give the programmer a handle on understanding memory usage. 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 six 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, because this would mean departing from using Standard ML as the source language).

Rather than delving into the design of a new programming language, we felt that it was more interesting to investigate whether regions could be extended to Standard ML modules. There were two reasons for extending the scheme to work with modules. First, to compile large programs, the ML Kit would need to compile modules. Second, dealing with region inference in some modular fashion was a necessity and an interesting challenge in itself. Region inference depended on a much finer level of description than the Standard ML type system itself offered. Separate compilation of modules normally required only type information. Thus the obvious question to ask was: 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, Elsman [17] 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. This scheme was introduced in version 3 of the ML Kit [54].

The scheme was based on a notion of *static interpretation of modules* [18], in which the module language was regarded as a linking language for specifying how program fragments are combined to form a complete program. As in most ML compilers, functors were type checked when they were declared. Thus, type errors in functors were caught already when the functor was declared (as opposed to when it was applied). However, code generation for the functor was postponed until the functor was applied. Indeed, if a functor was applied to two different arguments, code for its body was generated twice, possibly with different results. When the body of a functor was compiled, the actual argument to which it was applied was known. Thus, the compiler could make use of the information about the actual argument when code was generated for the functor body. In particular, information about region type schemes for functions in the actual argument could be used when region inference was performed for the functor body.

Delaying code generation until functor application time was not feasible for large programs unless it was integrated with a mechanism for avoiding unneccesary recompilation of program units and functor bodies upon change of source code. To solve this problem, the static interpretation of modules scheme collected information, which for each program unit or functor body told which other units it depended on. Such information included region type schemes for free identifiers of the program unit. Upon modification of a program unit, the scheme used the collected information to determine, for each program unit and each functor application, if recompilation was necessary.

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 Standard ML program, took one and a half 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 with the ML Kit than with SML/NJ (version 110.0.3).

This result was very interesting because much of the code in AnnoDomini was written by programmers who did not know how to program with regions—in fact, of the 58,000 lines of code, 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. Nevertheless, there was progress and we solved the fifth problem at the end of Section 4:

1. It was possible to extend region-inference and the related 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 (surprisingly) few modifications 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 tools that programmers would need to 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 suggested that one could get far without performing a major revision of the code, it had to be a concern for everybody who used regions that there was no guarantee that one would be able to solve all problems that one encountered within the regions scheme. If control over memory resources and the other benefits obtained by region inference were called for (e.g., real-time programming in embedded systems), a programmer would likely be willing to invest time in tuning a program with a view to using regions. But this willingness would probably fade if at the very last moment, a problem could occur that would force major revisions to the code or force giving up on regions altogether!

The solution to the second and third problem at the end of Section 4 still seemed to require change of source language, which we were not willing to do. A solution could be to allow region annotations in ML comments. The sixth problem could perhaps be addressed by combining region inference and garbage collection. If successful, a combination of region inference and garbage collection could perhaps even reduce the importance of the second and third problem: 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 [29], Hallenberg developed a scheme for garbage collecting regions and implemented it in the ML Kit. A slight variation of Hallenberg's scheme [30], which targets the x86 architecture, is used in Version 4 of the ML Kit [52].

The scheme was a generalisation of Cheney's stop-and-copy copying garbage collection algorithm [13]. Briefly, the idea was 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.

To combine region inference and garbage collection, it was necessary to ensure that no dangling pointers were introduced at runtime when regions were deallocated. Tofte and Talpin had noticed early that a side condition in the region typing rule for functions would make sure that no dangling pointers appeared at runtime [56, p. 50]. Elsman later stated and proved this property formally and showed that, for a range of benchmark programs, the added inflexibility had little effect on overall memory behavior [19].

For combining region inference and garbage collection, each region was associated with a *from-space* and a *to-space*. The allocation pointer a in the region descriptor (see Figure 2) played the dual rôle of the allocation pointer for region inference and the allocation pointer for garbage collection. The garbage collector never allocated into from-spaces. Scan pointers were kept in a *scan stack*; there was no scan pointer in the region descriptor. In the following explanation, the notation $r \to a$ will be used to refer to the allocation pointer a in a region r; a similar notation will be used to access the other components of a region descriptor. The scan pointer for a region r will be written s_r.

Cheney's algorithm was applied locally on each region using the stop criteria: $\forall r \in \mathit{Reg} : (r \to a) = s_r$, where *Reg* was the set of region descriptors on the region stack. The stop criteria was implemented using the scan stack, which consisted of those scan pointers s_r for which $s_r \neq (r \to a)$.

At the start of a garbage collection, the region stack was traversed and the region pages in the from-space areas (pointed at by $r \to \mathit{fp}$; see Figure 2) were linked together to form a single global from-space area. Next, for every region descriptor r on the region stack, $r \to \mathit{fp}$ was initialised to point at a fresh region page taken from the free-list. Moreover, $r \to a$ was initialised to point at the beginning of the page pointed to by $r \to \mathit{fp}$ and $r \to e$ at the end of the page pointed to by $r \to \mathit{fp}$. While collection was in progress, region pages were allocated from the free-list, which was disjoint from the global from-space area. After garbage collection, the global from-space area was appended to the free-list.

The garbage collector was invoked whenever more than 2/3 of the region pages in the free-list had been used.

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

Our benchmark programs revealed that using region inference greatly reduced the number of times the garbage collector needed to run, [30]. Furthermore, for most of the benchmark programs, using a combination of region inference and garbage collection was faster than using the garbage collector without region inference. For all benchmark programs, the fastest execution was obtained by using region inference without garbage collection (mostly because tags were not necessary if one did not do garbage collection).

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 was not surprising, since the garbage collector required 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 was not surprising, for programs that had not been optimised for regions often contained space leaks that made memory usage linear in the running time. The experiments thus confirmed the hope that adding a garbage collector to region inference really would take care of the (relatively few) allocations that were not reclaimed well by region inference.

An important question remained, however. What was the space usage using garbage collection alone compared to combining region inference and garbage collection?

At first, it seemed likely that combining region inference and garbage collection would use less space than garbage collection alone (because region inference would take care of some of the deallocation, 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: less space was required to use the garbage collector alone than when the garbage collector was combined with region inference. For these programs, we found the *region waste* (i.e., non-used memory in region pages) to be as high as 23 percent when garbage collection was combined with region inference [30]. Many infinite regions with only a few values each could take up much more space than putting all the values in a global region.

Our experiments showed that what strategy to use (i.e., region inference alone, garbage collection alone, or a combination of the two) is not a clear cut and depends on the program. However, the combination of region inference and garbage collection did give the programmer the flexibility to either optimise for regions or choose not to and instead use the garbage collector as a fall-back opportunity. The garbage collector could then be used alone or in combination with region inference, depending on how region unfriendly a program was. We believed this flexibility reduced the importance of the second and third problem at the end of Section 4 and solved the sixth problem.

The garbage collector made it possible to bootstrap the ML Kit using the ML Kit itself [52] and compare against SML/NJ (version 110.0.7) [30]. (A 1 GHz Pentium III, Coppermine, machine equipped with 1 Gb RAM was used for the bootstrapping experiment). In the first setting, the SML/NJ compiler was used to compile the ML Kit sources into a version of the ML Kit that, when running, used the SML/NJ runtime system. This version of the ML Kit was called *kit1*. Using *kit1* to compile the ML Kit sources into *kit2* used 809 Mb and took 40:41 min. The *kit2* executable ran on the runtime system of the ML Kit using the combination of region inference and garbage collection. Using *kit2* to compile the ML Kit sources into *kit3* used 904 Mb and took 17:33 min. This example showed that the combination of region inference and garbage collection can work extremely well on very large programs.

8. Related work

Much work has either influenced or been influenced by the work on region-based memory management and the later work on the ML Kit implementation.

One of the first suggestions for dividing memory into regions (or zones) appears in the AED Free storage package by Ross [45]. Memory regions are now also widely used in the C/C++ language community. For instance, regions (or arenas) are used in the Apache Web server [47] to dispose memory allocated by a Web script once the script has terminated.

The idea of region inference was strongly influenced by the early work on effect systems [37–39, 48, 49]. Other region-based frameworks for reasoning about memory reuse have been proposed, including a system for programs in continuation-passing style [61], one based on linear types [62], and one based on an imperative region sublanguage [32, 44]. Another line of related work investigates techniques for improving region-based memory management [2]. Common to these techniques is that they give up on the stack discipline of the Tofte-Talpin region type system.

Tofte and Talpin's quite involved soundness proof for the region type system [57, 58] has sparked a number of investigations of alternative and simpler techniques for proving type safety and soundness results for the region calculus [5, 11, 12, 31, 66].

Another line of related work includes the work by Aiken et al. on extending C with explicit region annotations [22, 23] and the work on Cyclone, a safe dialect of C [27, 35].

Separate from the already mentioned suggestions for extending region-based memory management, there have been a series of suggestions for making region-based memory management work for logic programming languages [40] and object oriented languages [10, 14, 65]. Moreover, recently there has also been work on a language-independent framework for region inference [41].

Also related to the work on region-based memory management is the work by Hofmann and Jost on static prediction of space usage for first-order functional programs [33, 34]. This work is based on linear typing but does not make use of regions. Instead the work focuses on tracking allocation and possible deallocation of individual memory cells.

There is a large body of work concerning general garbage collection techniques [36, 64] and escape analysis for improving stack allocation in garbage collected systems [9, 26]. The additional complexity of region inference and the polymorphic multiplicity analysis implemented in the ML Kit [8] allow more objects to be stack allocated than does traditional escape analyses, which allows only local, non-escaping values to be stack allocated. Region type systems have also been used for reasoning about the correctness of copying garbage collectors [43, 63]. Related to the latest ML Kit implementation, which uses an almost tag-free scheme for combining region-inference and garbage collection is the large body of work on tag-free garbage collection [1, 4, 24, 25, 59].

9. Current status and beliefs

Region-based memory management has matured quite significantly since its conception in 1994 as exemplified by the following noteworthy points about the status of the latest Version 4 of the ML Kit [52].

- Optimisations can be combined with region inference: Before region inference and other region analyses are performed on program fragments [8], the ML Kit performs a series of optimisations on an intermediate representation of the program fragments. The optimisations that are performed include function in-lining, specialisation of recursive functions and unboxing of function arguments. Each of these optimisations is region memory safe in the sense that executing an optimised program uses no more memory than when executing the corresponding unoptimised program.

- The ML Kit can bootstrap itself; see Section 7.
- Measurements show that the combination of region inference and garbage collection, as implemented in the ML Kit, is as efficient with respect to memory usage and execution time as a state-of-the-art generational garbage collection system [30].
- Recently, Elsman and Hallenberg [21] have implemented a multi-threaded Web server platform for Standard ML called SMLserver. The SMLserver project, which builds on a bytecode backend and interpreter for the ML Kit, demonstrates that region inference scales to environments where programs run quickly, but are executed often. While programs in SMLserver may execute simultaneously (running in different threads, without sharing objects) memory is allocated from a shared pool of region pages; thus, even programs that run simultaneously may use the same region page at different points in time.

Currently our beliefs about region-based memory management are as follows.

Things we believe work well:

1. The expressive power of region inference is capable of taking care of the vast majority of memory management in typical Standard ML programs.
2. The deallocation that is not done well by region inference can be handled adequately by a 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 Standard 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 CISC machines.
7. The concept of finite regions is very powerful. Finite regions 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. Unless combined with garbage collection, 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 is probably not the best way of handling tail recursion; it is complicated and vulnerable to program changes. Attempts to address the same problem as the storage mode analysis include [2, 15, 32].

3. It is not clear that infinite regions are 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. Experience with the garbage collector suggests that it might be better to use garbage collection for objects that region inference puts into infinite regions, due to fragmentation problems. But the experience so far is not conclusive: more investigation is needed to settle the question about whether infinite regions are a good or bad idea.

10. Future directions

The general approach taken in the our work 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 are candidates for scrapping.

One strand of future work is to investigate closer the use of infinite regions. An obvious thing to try is to use only one infinite region and then use a more sophisticated garbage collector (a generational collector, for example) on that region. Another issue is that, in the present implementation of garbage collection, one can only garbage collect at function entry points—it would, of course, be better if one could collect at each allocation point.

Another strand of future work is to use the accumulated knowledge about regions in the design of a new programming language that allows programmers to program explicitly with regions. Interesting steps in this direction have recently been taken in the Cyclone project [27, 35], which uses region-based memory management for a safe variant of the C programming language.

11. 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.

Acknowledgments

We wish to thank Peter Bertelsen, Tommy Højfeld Olesen, Nick Rothwell, Peter Sestoft, and David N. Turner for their contributions to the development of the ML Kit. We would also like to thank the anonymous reviewers and the editors for valuable comments and suggestions.

Notes

1. The type schemes inferred by region inference could be used to locate memory leaks: if the type scheme inferred for a function contained a so-called escaping put-effect, then it indicated that applications of the function could lead to space leaks [57].
2. See http://www.itu.dk/research/mlkit/kit2/readme.html
3. See http://www.itu.dk/research/mlkit/kit2/summerschool.html

References

1. Aditya, S., Flood, C.H., and Hicks, J.E. Garbage collection for strongly-typed languages using run-time type reconstruction. In *LISP and Functional Programming*, 1994, pp. 12–23.
2. Aiken, A., Fähndrich, M., and Levien, R. Better static memory management: Improving region-based analysis of higher-order languages. In *ACM Conference on Programming Language Design and Implementation*, 1995, pp. 174–185.
3. Appel, A.W. Garbage collection can be faster than stack allocation. *IPL*, **25**(4) (1987) 275–279.
4. Appel, A.W. Runtime tags aren't necessary. *Lisp and Symbolic Computation*, **2** (1989) 153–162.
5. Banerjee, A., Heintze, N., and Riecke, J.G. Region analysis and the polymorphic lambda calculus. In *Logic in Computer Science*, 1999, pp. 88–97.
6. Birkedal, L., Rothwell, N., Tofte, M., and Turner, D.N. The ML Kit (Version 1). Technical Report DIKU-report 93/14, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, 1993.
7. Birkedal, L. and Tofte, M. A constraint-based region inference algorithm. *Theoretical Computer Science*, **258**, (2001) 299–392.
8. Birkedal, L., Tofte, M., and Vejlstrup, M. From region inference to von Neumann machines via region representation inference. In *ACM Symposium on Principles of Programming Languages*, 1996, pp. 171–183.
9. Blanchet, B. Escape analysis: Correctness proof, implementation and experimental results. In *ACM Symposium on Principles of Programming Languages*, 1998, pp. 25–37.
10. Boyapati, C., Salcianu, A., Beebee, W., and Rinard, M. Ownership types for safe region-based memory management in real-time java. In *ACM Conference on Programming Language Design and Implementation*, 2003.
11. Calcagno, C. Stratified operational semantics for safety and correctness of the region calculus. In *ACM Symposium on Principles of Programming Languages*, 2001.
12. Calcagno, C., Helsen, S., and Thiemann, P. Syntactic type soundness results for the region calculus. *Information and Computation*, **173**(2) (2002).

13. Cheney, C.J. A non-recursive list compacting algorithm. *Communications of the ACM*, **13**(11) (1970) 677–678.
14. Christiansen, M.V. and Velschow, P. Region-based memory management in Java. Master's thesis, DIKU, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark, 1998.
15. Crary, K., Walker, D., and Morrisett, G. Typed memory management in a calculus of capabilities. In *ACM Symposium on Principles of Programming Languages*, 1999, pp. 262–275.
16. Damas, L. and Milner, R. Principal type schemes for functional programs. In *ACM Symposium on Principles of Programming Languages*, 1982, pp. 207–212.
17. Elsman, M. Program modules, separate compilation, and intermodule optimisation. Ph.D. thesis, Department of Computer Science, University of Copenhagen, 1999.
18. Elsman, M. Static interpretation of modules. In *ACM International Conference on Functional Programming*, Paris, France, 1999, pp. 208–219.
19. Elsman, M. Garbage collection safety for region-based memory management. In *ACM Workshop on Types in Language Design and Implementation*, 2003.
20. Elsman, M. and Hallenberg, N. An optimizing backend for the ML Kit using a stack of regions. Student Project 95-7-8, Department of Computer Science, University of Copenhagen (DIKU), 1995.
21. Elsman, M. and Hallenberg, N. Web programming with SMLserver, In *International Symposium on Practical Aspects of Declarative Languages*, 2003.
22. Gay, D. and Aiken, A. Memory management with explicit regions. In *ACM Conference on Programming Language Design and Implementation*, 1998, pp. 313–323.
23. Gay, D. and Aiken, A. Language support for regions. In *ACM Conference on Programming Language Design and Implementation*, 2001, pp. 70–80.
24. Goldberg, B. Tag-free garbage collection for strongly typed programming languages. In *ACM Conference on Programming Language Design and Implementation*, 1991, pp. 165–176.
25. Goldberg, B. and Gloger, M. Polymorphic type reconstruction for garbage collection without tags. In *LISP and Functional Programming*, 1992, pp. 53–65.
26. Goldberg, B. and Park, Y.G. Higher order escape analysis: Optimizing stack allocation in functional program implementations. In *Proceedings of the third European Symposium on Programming*, LNCS-432, 1990, pp. 152–160.
27. Grossman, D., Morrisett, G., Jim, T., Hicks, M., Wang, Y., and Cheney, J. Region-based memory management in cyclone. In *ACM Conference on Programming Language Design and Implementation*, 2002.
28. Hallenberg, N. A region profiler for a standard ML compiler based on region inference. Student Project 96-5-7, Department of Computer Science, University of Copenhagen (DIKU), 1996.
29. Hallenberg, N. Combining garbage collection and region inference in the ML Kit. Master's thesis, Department of Computer Science, University of Copenhagen. Available via `http://www.it-c.dk/research/mlkit`, 1999.
30. Hallenberg, N., Elsman, M., and Tofte, M. Combining region inference and garbage collection. In *ACM Conference on Programming Language Design and Implementation*, 2002.
31. Helsen, S. and Thiemann, P. Syntactic type soundness for the region calculus. In *Proceedings of the 4th International Workshop on Higher Order Operational Techniques in Semantics*, Published in vol. 41(3) of the Electronic Notes in Theoretical Computer Science, 2000.
32. Henglein, F., Makholm, H., and Niss, H. A direct approach to control-flow sensitive region-based memory management. In *ACM Conference on Principles and Practice of Declarative Programming*, Montréal, Canada, 2001, pp. 175–186.
33. Hofmann, M. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, **7**(4) (2000) 258–289.
34. Hofmann, M. and Jost, S. Static prediction of heap space usage for first-order functional programs. In *ACM Symposium on Principles of Programming Languages (POPL'03)*, 2003, pp. 185–197.
35. Jim, T., Morrisett, G., Grossman, D., Hicks, M., Cheney, J., and Wang, Y. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, 2002.
36. Jones, R. and Lins, R. *Garbage Collection*, Wiley, 1996.
37. Jouvelot, P. and Gifford, D. Algebraic reconstruction of types and effects. In *ACM Symposium on Principles of Programming Languages*, 1991.

38. Lucassen, J. and Gifford, D. Polymorphic effect systems. In *ACM Symposium on Principles of Programming Languages*, 1988.
39. Lucassen, J.M. Types and effects, towards the integration of functional and imperative programming. Ph.D. thesis, MIT Laboratory for Computer Science, MIT/LCS/TR-408, 1987.
40. Makholm, H. A region-based memory manager for Prolog. In *International Symposium on Memory Management (ISMM'2000)*, 2000, pp. 25–34.
41. Makholm, H. A language-independent framework for region inference. Ph.D. thesis, Department of Computer Science, University of Copenhagen, 2003.
42. Miller, J.S. and Rozas, G.J. Garbage collection is fast, but a stack is faster. Technical Report Memo 1462, MIT, Cambridge, Massachusetts, 1994.
43. Monnier, S., Saha, B., and Shao, Z. Principled scavenging. In *ACM Conference on Programming Language Design and Implementation*, 2001.
44. Niss, H. Regions are imperative: Unscoped regions and control-flow sensitive memory management. Ph.D. thesis, Department of Computer Science, University of Copenhagen (DIKU), 2002.
45. Ross, D.T. The AED free storage package. *Communication of the ACM*, **10**(8) (1967) 481–492.
46. Runciman, C. and Wakeling, D. Heap profiling of lazy functional languages. *Journal of Functional Programming*, **3**(2) (1993) 217–245.
47. Stein, L. and MacEachern, D. *Writing Apache Modules with Perl and C*, O'Reilly & Associates, 1999, ISBN 1-56592-567-X.
48. Talpin, J.-P. and Jouvelot, P. Polymorphic type, region and effect inference. *Journal of Functional Programming*, **2**(3), 1992.
49. Talpin, J.-P. and Jouvelot, P. The type and effect discipline. *Information and Computation*, **111**(2) (1994) 245–296. Extended abstract in Proceedings of the *IEEE Conference on Logic in Computer Science (LICS'92)*, June 1992.
50. Tofte, M. and Birkedal, L. A region inference algorithm. *ACM Transactions on Programming Languages and Systems*, **20**(4) (1998) 734–767. (plus 24 pages of electronic appendix).
51. Tofte, M. and Birkedal, L. Unification and polymorphism in region inference. In *Proof, Language, and Interaction. Essays in Honour of Robin Milner*, G. Plotkin, C. Stirling, and M. Tofte (Eds.), Foundations of Computing. Cambridge, Massachusetts: The MIT Press, 2000, pp. 389–425.
52. Tofte, M., Birkedal, L., Elsman, M., Hallenberg, N., Olesen, T., and Sestoft, P. Programming with regions in the ML Kit (for Version 4). The IT University of Copenhagen, 2002. Available via `http://www.it-c.dk/research/mlkit`.
53. Tofte, M., Birkedal, L., Elsman, M., Hallenberg, N., Olesen, T.H., Sestoft, P., and Bertelsen, P. Programming with regions in the ML Kit, Technical Report DIKU-TR-97/12, Department of Computer Science, University of Copenhagen, 1997. Available via `http://www.it-c.dk/research/mlkit`.
54. Tofte, M., Birkedal, L., Elsman, M., Hallenberg, N., Olesen, T.H., Sestoft, P., and Bertelsen, P. Programming with regions in the ML Kit (for Version 3). Technical Report DIKU-TR-98/25, Department of Computer Science, University of Copenhagen, 1998. Available via `http://www.it-c.dk/research/mlkit`.
55. Tofte, M. and Talpin, J.-P. Data region inference for polymorphic functional languages. Manuscript, 1992.
56. Tofte, M. and Talpin, J.-P. A theory of stack allocation in polymorphically typed languages. Technical Report DIKU-report 93/15, Department of Computer Science, University of Copenhagen, 1993.
57. Tofte, M. and Talpin, J.-P. Implementing the call-by-value lambda-calculus using a stack of regions. In *ACM Symposium on Principles of Programming Languages*, 1994, pp. 188–201.
58. Tofte, M. and Talpin, J.-P. Region-based memory management. *Information and Computation*, **132**(2) (1997) 109–176.
59. Tolmach, A.P. Tag-free garbage collection using explicit type parameters. In *LISP and Functional Programming*, 1994, pp. 1–11.
60. Vejlstrup, M. Multiplicity inference. Master's thesis, Department of Computer Science, Univ. of Copenhagen, 1994, report 94-9-1.
61. Walker, D., Crary, K., and Morrisett, G. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **22**(4) (2000) 701–771.
62. Walker, D. and Watkins, K. On regions and linear types. In *ACM International Conference on Functional Programming*, 2001, pp. 181–192.

63. Wang, D.C. and Appel, A.W. Type-preserving garbage collectors. In *ACM Symposium on Principles of Programming Languages*, 2001, pp. 166–178.
64. Wilson, P.R., Johnstone, M.S., Neely, M., and Boles, D. Dynamic storage allocation: A survey and critical review. In *International Workshop on Memory Management*, 1995.
65. Yates, B.N. A type-and-effect system for encapsulating memory in Java. Master's thesis, Department of Computer Science and Information Science, University of Oregon, 1999.
66. Zilio, S.D. and Gordon, A. Region analysis and a pi-calculus with groups. *Journal of Functional Programming*, **12**(3) (2002) 229–292.