Technical Report DIKU-TR-97/12
Department of Computer Science
University of Copenhagen
Universitetsparken 1
DK-2100 KBH Ø
DENMARK

April 1997

Programming with Regions in the ML Kit

Mads Tofte Lars Birkedal Martin Elsman
Niels Hallenberg
Tommy Højfeld Olesen
Peter Sestoft Peter Bertelsen

Programming with Regions in the ML Kit

Mads Tofte Lars Birkedal Martin Elsman
Niels Hallenberg Tommy Højfeld Olesen Peter Sestoft
Peter Bertelsen

April 23, 1997

Values and Their Representation

integer	32 bits, untagged. Unboxed (i.e., not region allocated).
real	64 bits, untagged. Boxed (i.e., allocated in region)
string	Unbounded size. Allocated in region.
bool	one 32 bit word. Unboxed.
α list	`nil` and `::` cells in one region; auxiliary pairs in one region; elements in one or more regions. Size of `::` cell: two 32 bit words; size of auxiliary pair: two 32 bit words.
α tree	A tree and its subtrees reside in one region. Elements in one region (if not unboxed).
`exn`	Nullary exception names are unboxed. A constructed exception value (i.e., a unary exception constructor applied to a value) is stored in a global region.
`fn` *pat* `=>` *exp*	An anonymous function is represented by a boxed, untagged closure. Size (in 32 bit words): 1 plus the number of free variables of the function. (Free region variables also count as variables.)
`fun` f ...	Mutually recursive region-polymorphic functions share the same closure, which is region-allocated, untagged and whose size (in words) is the number of variables that occur free in the recursive declaration.

Regions and Their Representation

Finite (ρ:n)	Region whose size can be determined at compile time. During compilation, a finite region size is given as a non-negative integer. After multiplicity inference, this integer indicates the number of times a value (of the appropriate type) is written into the region. Later, after physical size inference, the integer indicates the physical region size in words. At runtime, a finite region is allocated on the runtime stack.
Infinite (ρ:`INF`)	All other regions. At runtime, an infinite region consists of a region descripter on the stack which contains pointers to the beginning and the end of a linked list of fixed size region pages.

Storage Modes (only significant for infinite regions)

`atbot`	Reset region, then store value.
`sat`	Determine actual storage mode (`attop`/`atbot`) at runtime.
`attop`	Store at top of region, without destroying any values already in the region.

Contents

Preface

The ML Kit with Regions is a Standard ML compiler. It is intended for the development of stand-alone applications which must be small, reliable, fast and space efficient.

There has always been a tension between high-level features in programming languages and the programmer's legitimate need to understand programs at the operational level. Very likely, if a resource conscious programmer is forced to make a choice between the two, he will choose the latter.

The ML Kit with Regions is the result of a research and development effort which has been going on at the University of Copenhagen for the past five years. The goal of this project has been to develop implementation technology which combines the advantages of using a high-level programming language, in this case Standard ML, with a model of computation which allows programmers to reason about how much space and time their programs will use.

In most call-by-value languages, it is not terribly hard to give a model of time usage which is good enough for elementary reasoning.

For space, however, the situation is much less satisfactory. Part of the reason is that many programs must recycle memory while running. For all such programs, the mechanisms that reclaim memory inevitably become part of the reasoning. This is true irrespective of whether memory recycling is done by a stack mechanism or by pointer tracing garbage collection.

In the stack discipline, every point of allocation is matched by a point of de-allocation and these points are obvious from the program. By contrast, garbage collection techniques usually separate allocation, which is done by the programmer, from de-allocation, which is done by a garbage collector. The advantage of using reference tracing garbage collection techniques is that they apply to a wide range of high-level concepts now found in programming languages, for example recursive data types, higher-order functions, excep-

tions, references and objects. The disadvantage is that it is becoming increasingly difficult for the programmer to reason about lifetimes. Lifetimes may depend on subtle details in the compiler and in the garbage collector. Thus it is hard to model memory in a way which is useful to programmers. Also, compilers offer little assistance for reasoning about lifetimes.

In this report we equip Standard ML with a different memory management discipline, namely a *region-based* memory model. Like the stack discipline, the region discipline is, in essence, simple and platform-independent. Unlike the traditional stack discipline, however, the region discipline also applies to recursive data types, references and higher-order functions, for which one has hitherto mostly used reference tracing garbage collection techniques.

The reader we have in mind is a person with a Computer Science background who is interested in developing small, but reliable and efficient applications written in Standard ML. Also, the report may be of interest to researchers of programming languages, since the ML Kit with Regions is a fairly bold exercise in program analysis. We should emphasise, however, that this report is very much intended as a user's guide, not a scientific publication.

This report consists of three parts:

Part I: Overview, in which we give an overview of the ideas that underlie programming with regions in the Kit;

Part II: Understanding Regions, in which we systematically go through the language constructs of the Standard ML Core Language, showing for each one how it can be used when programming with regions;

Part III: System Reference, in which we explain how to interact with the system, how to use the region profiler and how to call C functions from the Kit.

The ML Kit with Regions is also called the ML Kit Version 2, since it is a further development of the ML Kit Version 1, which was developed at Edinburgh University and Copenhagen University. We hope you will enjoy using the ML Kit with Regions as much as we have enjoyed developing it. If your experience with the Kit gives rise to comments and suggestions, specifically with relation to the goals and visions expressed above, please feel free to write. Further information is available at our web site:

`http://www.diku.dk/research-groups/topps/activities/kit2/index.html`

April, 1997

Mads Tofte, Lars Birkedal, Martin Elsman,
Niels Hallenberg, Tommy Højfeld Olesen,
Peter Sestoft, Peter Bertelsen

Part I

Overview

Chapter 1

Region-Based Memory Management

Region-Based Memory Management is a technique for managing memory for programs that have dynamic data structures, such as lists, trees, pointers and function closures.

1.1 Prevailing Approaches to Dynamic Memory Management

Many programming languages rely on a memory model consisting of a *stack* and a *heap*. Typically, the stack holds temporary values, activation records, arrays and in general, values whose lifetime is closely connected to procedure activations and whose size can be determined at the latest when creation of the value begins. The heap is what holds all the other values. In particular, the heap holds values whose size can grow dynamically, such as lists and trees. The heap also holds values whose lifetime does not follow procedure activations closely (for example lists and, in functional languages, function closures and suspensions).

The beauty of the stack discipline (apart from the fact that it is often very efficient in practice) is that it couples allocation points and de-allocation points in a manner which is intelligible to the programmer. C programmers appreciate that whatever memory is allocated for local variables in a procedure ceases to exist (and take up memory) when the procedure returns. C programmers also know that counting from one to some large number, N, is

not best done by making N recursive C procedure calls, since that would use stack space proportional to N.

By contrast, programmers have much less help when it comes to managing the heap. Two approaches prevail. The first approach is that the programmer manages memory herself, using explicit allocation and de-allocation instructions (e.g., `malloc` and `free` in C). For non-trivial programs this can be a very significant burden, since it is, in general, very hard to make sure that none of the values that reside in the memory which one wishes to de-allocate are not needed for the rest of the computation. This puts the programmer in a very difficult position. If one is too eager to reclaim memory in the heap, the program might crash under some peculiar circumstances which might be hard to find during debugging. If one is too conservative reclaiming memory, the program might end up "leaking space", i.e., using more memory than expected, perhaps eventually exhausting the memory of the machine.

The other prevailing approach is to use automatic garbage collection in the heap. Some implementers of some languages even dispense with the stack entirely, relying only on a heap with garbage collection. Garbage collection techniques separate allocation, which is done by the programmer, from de-allocation, which is done by the garbage collector. At first, this might seem like the perfect solution: no longer does the programmer have to worry about whether memory that is being reclaimed really is dead, for the garbage collector only reclaims memory which cannot be reached by the rest of the computation. However, reality is less perfect. Garbage collectors are typically based on the idea that if data is reachable via pointers (starting from the stack and other root data) then those data must be kept. Consequently, programs have to be written with care to avoid hanging on to too many pointers. Space conscious programmers (and language implementers) can work their way around these problems, for example by assigning `nil` to pointers that are no longer used. However, such tricks often rely on assumptions about the code which cannot be checked by the compiler and which are likely to be invalidated as the program evolves.

1.2 Checked De-allocation of Memory

Regions offer an alternative to these two approaches. The runtime model is very simple, at least in principle. The store consists of a stack of *regions*, see Figure 1.1. Regions hold values, for example tuples, records, function

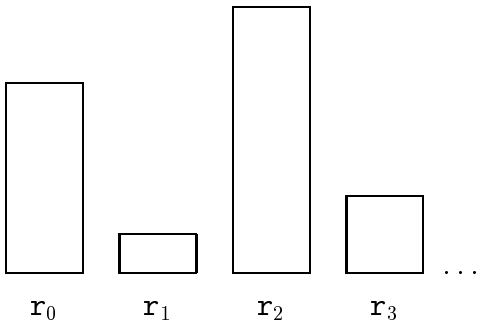


Figure 1.1: The store is a stack of regions; every region is depicted by a box in the picture.

closures, references and values of recursive types (such as lists and trees). All values, except those that fit within one machine word (for example integers) are stored in regions.

The size of a region is not necessarily known when the region is allocated. Thus a region can grow gradually (and many regions can grow at the same time) so one might think of the region stack as a stack of heaps. However, the region stack really is a stack in the sense that (a) if region r_1 is allocated before region r_2 then r_2 is de-allocated before r_1 and (b) when a region is de-allocated, all the memory occupied by that region is reclaimed in one constant time operation.

Values which reside in one region are often, but not always, of the same type. A region can contain pointers to values that reside in the same region or in other regions. Both forward pointers (i.e., pointers from a region into a region closer to the stack top) and backwards pointers (i.e., pointers to an older region) occur.

Conceivably, one can combine the region scheme with pointer tracing garbage collection techniques.[1] In the present version of the ML Kit, however, the region stack is the only form of memory management provided. How can that be so? Is the region model really general enough to fit a wide variety of

[1]Indeed we might well provide a release of the ML Kit which has both regions and reference-tracing garbage collection.

computations?

First note that the pure stack discipline (a stack, but no heap) is a special case of the region stack. Here the size of a region is known at the latest when the region is allocated. Another special case is when one has just one region in the region stack and that region grows dynamically. This can be thought of as a heap with no garbage collection, which again would not be sufficient.

But when one has many regions, one obtains the possibility of distinguishing between values according to what region they reside in. The ML Kit contains operations for allocating, de-allocating and extending regions. But it also has an explicit operation for resetting an existing region, i.e., reclaiming all the memory occupied by the region without eliminating the region from the region stack. This primitive, simple as it is, enables one to cope with most of those situations where lifetimes simply are not nested. Figure 1.2 shows a possible progression of the region stack.

In the ML Kit the vast majority of region management is done automatically by the compiler and the runtime system. Indeed, with one exception, source programs are written in Standard ML, with no added syntax or special directives. The exception has to do with resetting of regions. The Kit provides two built-in functions (`resetRegions` and `forceResetting`) which instruct the program to reset regions. Here `resetRegions` is a safe form of resetting where the compiler only inserts region resetting instructions if it can prove that they are safe, and prints thorough explanations of why it thinks resetting might be unsafe otherwise. Function `forceResetting` is for potentially unsafe resetting of regions, which is useful in cases where the programmer jolly well knows that resetting is safe even if the compiler cannot prove it. `forceResetting` is the only way we allow users to make decisions that can make the program crash; many programs do not need `forceResetting` and hence cannot crash (unless we have bugs in our system).

All other region directives, including directives for allocation and de-allocation of regions, are inferred automatically by the compiler. This happens through a series of fairly complex program analyses and transformations (in the excess of twenty-five passes involving three typed intermediate languages). These analyses are formally defined and the central one, called *region inference*, has been proved correct for a skeletal language. Although the formal rules that govern region inference and the other program analyses are complex, we have on purpose restricted attention to program analyses which we feel capture natural programming intuitions. Moreover, the Kit im-

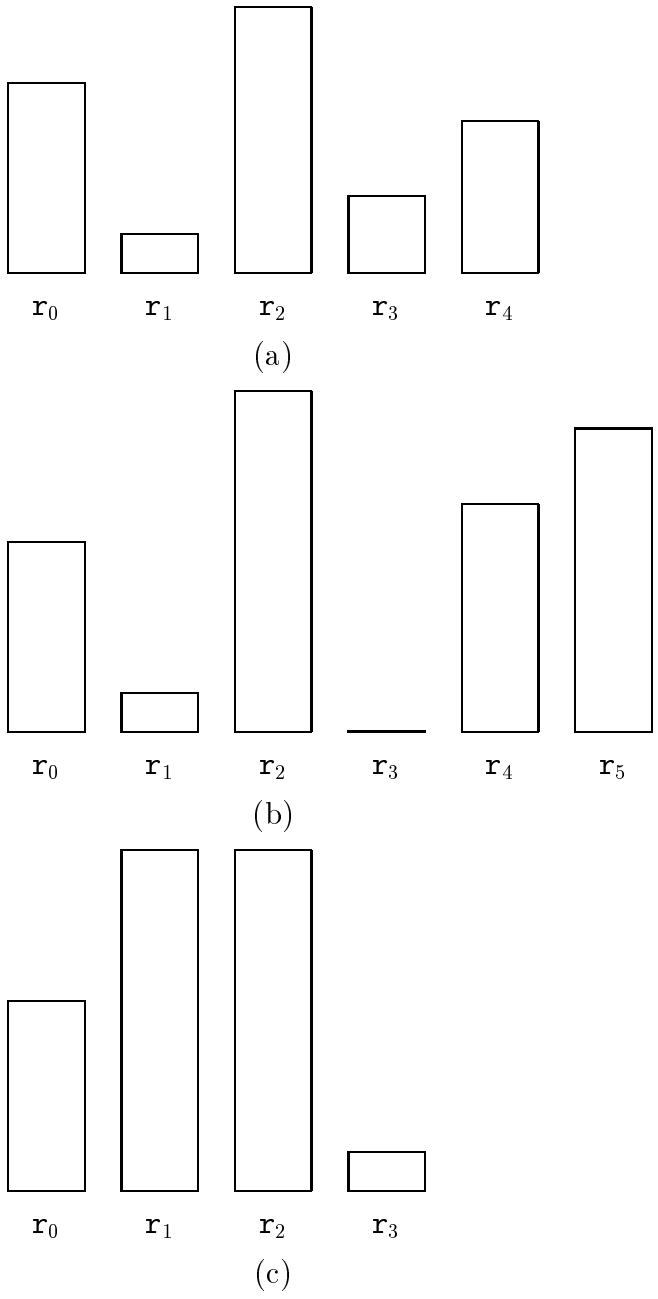


Figure 1.2: Further development of the region stack: (a) after allocation of $\mathtt{r}_4$; (b) after growth of $\mathtt{r}_2$ and $\mathtt{r}_4$, resetting of $\mathtt{r}_3$ and allocation of $\mathtt{r}_5$; (c) after popping of $\mathtt{r}_4$ and $\mathtt{r}_5$ but extension of $\mathtt{r}_1$ and $\mathtt{r}_3$.

plementation is such that, with one exception[2], every region directive takes constant time and constant space to execute. The fact that we avoid interrupting program execution for unbounded lengths of time gives a nice smooth experience when programs are run and should make the scheme attractive for real-time programming.

To help programmers get used to the idea of programming with regions, the ML Kit can print region-annotated programs, i.e., source programs it has annotated with region directives. Also, it provides a *region profiler* for examining run-time behaviour. The region profiler gives a graphical representation of region sizes as a function of time. This makes it possible to see which regions use the most space and even to relate memory consumption back to individual allocation points in the (annotated) source program.

To sum up, the key advantages obtained by using regions compared to more traditional memory management schemes are

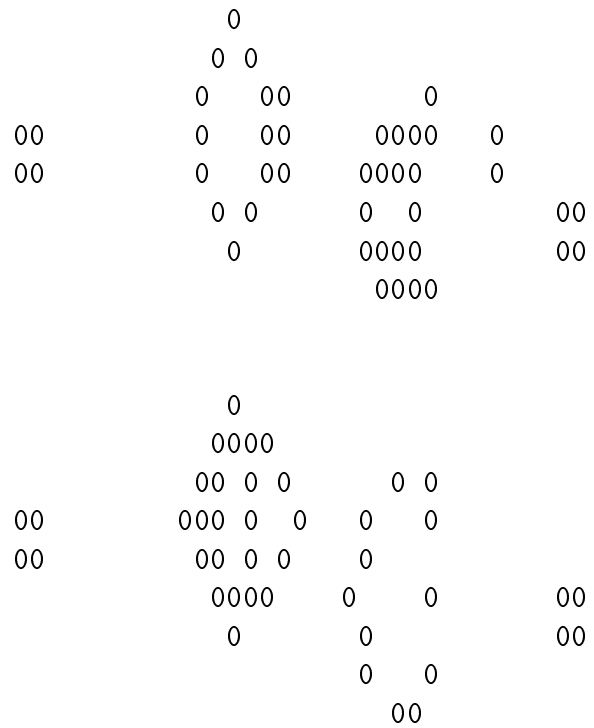
1. safety of de-allocation is checked by the compiler;
2. the compiler can in many cases spot potential space leaks;
3. region management is under the control of the user, provided one understands the principles of region inference;
4. each of the region operations that are inserted use constant time and constant space at runtime;
5. it is possible to relate runtime space consumption to allocation points in the source program; we have found region profiling to be a powerful tool for eliminating space leaks.

Regions are not a magic wand to solve all memory management problems. Rather, the region scheme encourages a particular discipline of programming. The purpose of this report is to lay out this discipline of programming.

[2]The exception has to do with exceptions. When an exception is raised, a search down the stack for a handler takes place; this search is not constant time and it involves popping of regions on the way. However, the number of region operations is bounded by the number of handlers that appear on the stack.

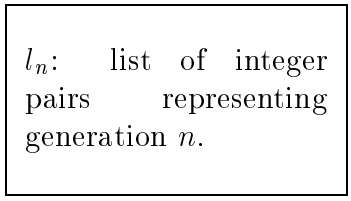
1.3 Example: the Game of Life

To illustrate the general flavour of region-based memory management, let us consider the problem of implementing the game of Life. The game takes place on a board that resembles a chess board, except that the size of the board can grow as the game evolves. Thus every position has eight neighbouring positions (perhaps after extension of the board). At any point in time, every position is either *alive* or *dead*. A snapshot of the game consisting of the board together with an indication of which positions are alive is called a *generation*. The rules of the game specify how to progress from one generation to the next. Consider generation n from which we want to create generation $n+1$ ($n \geq 0$). Let (i, j) be a position on the board, relative to some fixed point $(0, 0)$ in the plane. Assume (i, j) is alive in generation n. Then (i, j) stays alive in generation $n+1$ if and only if it has two or three live neighbours in generation n. Assume (i, j) is dead at generation n. Then it is born in generation $n+1$ if and only if it has precisely three live neighbours at generation n. We assume that only finitely many positions are alive initially. An example of two generations of Life is shown below:



To represent the game board, we need a data structure which can grow dynamically (so a two-dimensional array of fixed size is not sufficient). A simple solution is to represent a generation by a list of integer pairs, namely the positions that are alive. Since we want to give all pairs belonging to one generation the same lifetime (in the computer memory, that is!) it is natural to store all the integer pairs belonging to one generation in the same region. Indeed region inference forces this decision upon us, as it happens, since it requires that all elements belonging to the same list lie in the same region. (Different lists can lie in different regions, however.)

Thus, after having built the initial generation, we expect the region stack to look like this



The computation of the next generation involves a considerable amount of list computation. Xavier Leroy has expressed the key part of the computation as shown in Figure 1.3. Despite the extensive use of higher-order functions here, there is a great deal of stack structure in this computation. For example, the `survivors` list can be allocated in a local region which can be de-allocated after the list has been appended (`@`) to the `newborn` list. The computation of `survivors`, in turn, involves the creation of a closure for `(twoorthree o liveneighbours)` and additional creation of closures as part of the computation of the application of `filter`. Each time `liveneighbours` is called (by `filter`) additional temporary values are created. All of this data should live shorter than `survivors` itself. The details of these lifetimes are determined automatically by the region inference algorithm which ensures that when the above expression terminates it will simply have created a list containing the live positions of the new generation.

But now we have a design choice. Should we put the new generation in the same region as the previous region or should we arrange that it is put in a separate region? Piling all generations on top of each other in the same region would clearly be a waste of space: only the most recent generation is ever

```
  let val living = alive gen
      fun isalive x = member eq_int_pair_curry living x
      fun liveneighbours x = length(filter isalive (neighbours x))
      fun twoorthree n = n=2 orelse n=3
      val survivors = filter (twoorthree o liveneighbours) living
      val newnbrlist = collect
                       (fn z => filter (fn x => not(isalive x))
                                       (neighbours z)
                       ) living
      val newborn = occurs3 newnbrlist
  in
     mkgen (survivors @ newborn)
  end
```

Figure 1.3: An excerpt of a (modified version of) Xavier Leroy's Game of Life benchmark.

needed. Similarly, giving each generation a separate region on the region stack is no good either, since it would make the stack grow ad infinitum (although this could be alleviated somewhat by resetting all regions except the topmost one). The solution is simple, however: use two regions, one for the current generation and one for the new generation. When the new generation has been created, reset the region of the old region and copy the contents of the the new region into the old region. This is achieved by organising the main loop of the program as follows:

```
  local
(*1*) fun nthgen'(p as(0,g)) = p
(*2*)   | nthgen'(p as(i,g)) =
(*3*)       nthgen' (i-1, let val g' = nextgen  g
(*4*)                     in  show g;
(*5*)                         resetRegions g;
(*6*)                         copy g'
(*7*)                     end)
  in
(*8*) fun iter n = #2(nthgen'(n,gun()))
  end
```

Here `nthgen'` is the main loop of the program. It takes a pair as argument; the first component of the pair indicates the number of iterations desired, while the second, `g`, is the current generation. The use of the `as` pattern in line 1 forces the argument and the result of `nthgen'` to be in the same regions. Such a function is called a *region endomorphism*. In line 3, we compute a fresh generation which lies in fresh regions, as it happens. Having printed the generation (line 4) we then reset the regions containing `g`. The compiler checks that this is safe. Then, in line 6 we copy `g'` and the target of this copy must be the regions of `g`, since `nthgen'` is a region endomorphism (see Figure 1.4). All in all we have achieved that at most two generations are live at the same time (a fact which can be checked by inspecting the region annotated code, if one feels passionately about it).[3]

The above device, which we refer to as *double copying*, can be seen as a much expanded version of what is often called "tail recursion optimisation". In the case of regions, not just the stack space, but also region space, is re-used. Indeed, double copying is similar to invoking a copying garbage collector on specific regions which are known not to have live pointers into them. But by doing the copying ourselves, we have full control over when it happens, we know that the cost of copying will be proportional to the size of the generation under consideration and that all other memory management is done automatically by the region mechanism. Since each of the region management directives which the compiler inserts in the code are constant time and space operations, we have now avoided unpredictable interruptions due to memory management. This might not be terribly important for the purpose of the game of Life, but if we were writing control software for the ABS brakes of a car, having control over all costs, including memory management, would be crucial!

Region profiles for two hundred generations of `life` starting from the configuration shown earlier appear in Figures 1.6 and 1.5. The highest amount of memory used for regions during the computation is 29.000 bytes. Figure 1.6, which has data collected from 1000 snapshots of the computation, clearly shows that most of the 29.000 bytes are reclaimed between every two generations of the game. It turns out that the game essentially stabilises with

[3]The entire life program is available in `kitdemo/life.sml`, project `kitdemo/life`. (Running projects is described in Section 3.6.) Run with n=10000 on the HP PA-RISC, the memory consumption (program + data) quickly reaches 192Kb and it stays there for the remaining generations. The size of the executable program, which includes the runtime system, is 164Kb.

a small number of live positions on the board after roughly 150 generations. This is clearly reflected in the region profile.

Figure 1.6 is from the same computation, but it only includes data from 80 snapshots. This makes it easier to see that the largest regions are `r1588` and `r1121`. To find out what these regions contain, however, one needs to master the methods described in Part II.

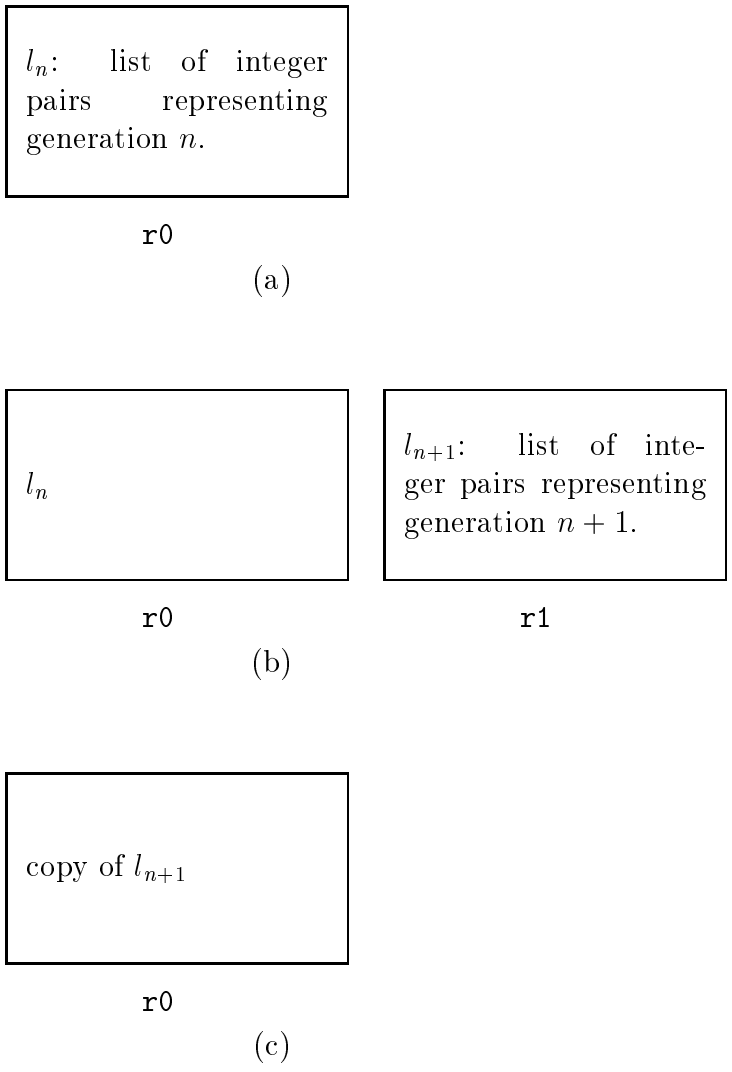


Figure 1.4: Using double-copying in the game of Life: (a) generation number n resides in region `r0`; (b) generation $(n+1)$ has been built in `r1`; (c) region `r0` has been reset, the new generation copied into `r0` and `r1` has been deallocated.

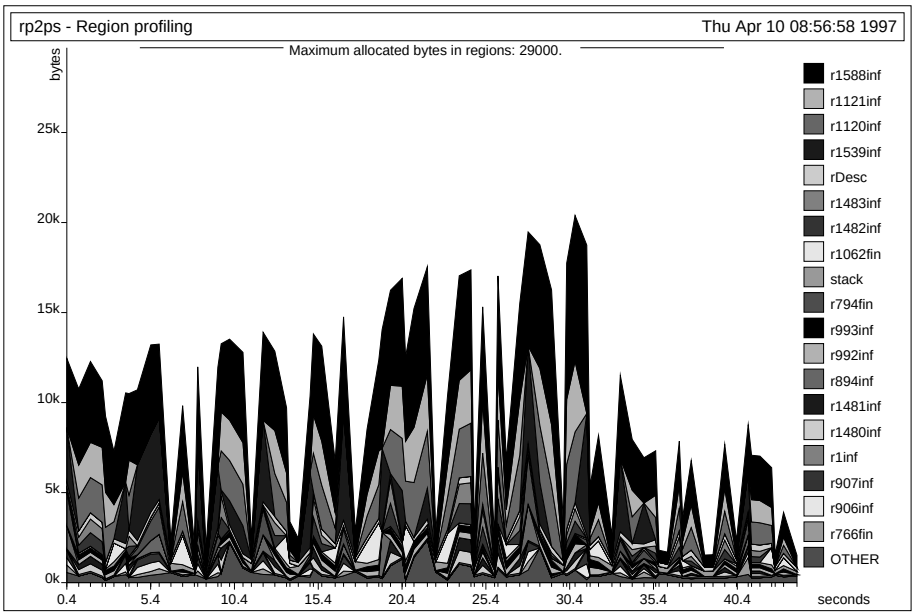


Figure 1.5: A region profile of two hundred generations of the "Game of Life", showing region sizes as a function of time (80 snapshots).

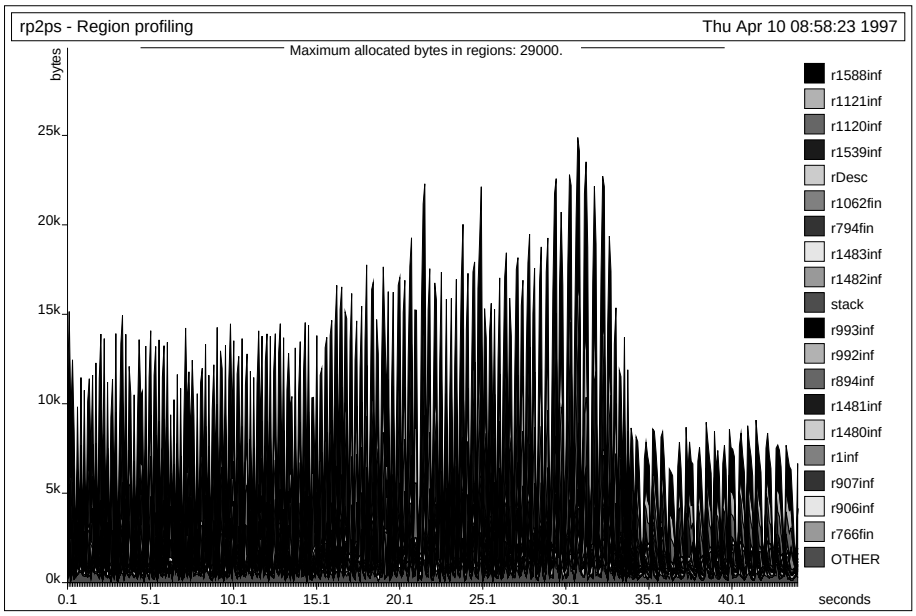


Figure 1.6: Region profile of two hundred generations of the "Game of Life", showing region sizes as a function of time (1000 snapshots).

Chapter 2

Making Regions Concrete

In this chapter we give a brief overview of how the abstract memory model presented in the last chapter is mapped down to conventional memory. In doing so, we shall introduce notation and concepts that will be used extensively in what follows.

2.1 Finite and Infinite Regions

Not every region has the property that its size is known at compile-time, or even when the region is first allocated at runtime. As we have seen, one typical use of a region is to hold a list, and in general there is no way of knowing how long a given list is going to be.

For efficiency reason, however, the Kit distinguishes between two kinds of regions: those regions whose size it can determine at compile-time and those it cannot. These regions are referred to as *finite* and *infinite* regions, respectively.[1] Finite regions are always allocated on the runtime stack. An infinite region is represented as a linked list of fixed-size pages. The runtime system maintains a free list of such pages. An infinite region is represented by a *region descriptor*, which is a record kept on the runtime stack. The region descriptor contains two pointers: one to the first and one to the last region page in the linked list which represents the region. Allocating an infinite region involves getting a page from the free list and pushing a region descriptor onto the runtime stack. Popping a region is done by appending

[1] "finite" and "unbounded" would have been better terms, but it is too late to change that.

the region pages of the region and the free list (this is done in constant time) and then popping the region descriptor off the runtime stack.

At runtime, every region is represented by a 32-bit entity, called a *region name*. If the region is finite, the region name is a pointer into the stack, namely to the beginning of the region. If the region is infinite, the region name is a pointer to the region descriptor of the region.

The *multiplicity* of a region is a statically determined upper bound on the number of times a value is put into the region. The Kit operates with three multiplicities: 0, 1 and ∞, ordered by $0 < 1 < \infty$. Multiplicities annotate binding occurrences of region variables. An expression of the form

$$\texttt{letregion}\ \rho : m\ \texttt{in}\ e\ \texttt{end}$$

where m is a multiplicity, gives rise to an allocation of a region which is finite if $m < \infty$, and infinite otherwise.

2.2 Runtime Types of Regions

Every region has a runtime type. The following runtime types exist: `real`, `string` and `top`. Not surprisingly, regions of runtime type `real` and `string` contain values of ML type `real` and `string`, respectively. Regions with runtime type `top` can contain all other forms of allocated values, i.e., constructed values, tuples, records and function closures.

It is often, but not always, the case that all values that reside in the same region have the same type (considered as representations of ML values).

2.3 Allocation and De-Allocation of Regions

The analysis which decides when regions should be allocated and de-allocated is called *region inference*. Region inference inserts several forms of memory management directives as directives into the program. The target language of region inference is called *RegionExp*.

In *RegionExp*, region allocation and de-allocation are explicit, they are always paired and they follow the syntactical structure of the source program. If e is an expression in *RegionExp*, then so is

$$\texttt{letregion}\ \rho\ \texttt{in}\ e\ \texttt{end}$$

Here ρ is a *region variable*. At runtime, first a region is allocated and bound to ρ. Then e is evaluated, presumably using the region bound to ρ for storing values. Upon reaching `end`, the program pops the region.

Region inference also decides, for each value-producing expression, into which region (identified by a region variable) the value will be put.

We emphasise that region variables and `letregion`-expressions are not present in source programs. The source language is unadulterated Standard ML, so programs that run on the Kit should be easy to port to any other Standard ML implementation.

2.4 The Kit Abstract Machine

The Kit contains a virtual machine, called the *Kit Abstract Machine* (KAM, for short), which details the above ideas. The KAM is a register machine with one linear address space which it partitions into a stack and a heap. The heap holds region pages, all of the same size. The KAM has simple RISC-like instructions, for example for moving word-size data between two registers or between a register and a memory location. More complex operations, such as function application, are expressed by sequences of KAM instructions.

For the purpose of this report, we assume that the KAM has infinitely many registers. In reality, there is a fixed number of 32 bit registers and register allocation assigns machine registers to KAM registers, using the runtime stack for spilling. However, register allocation will not be described in this report. Also, we do not discuss the interaction between hardware cache strategies and the code generated by the Kit. While both can be important in practice, we do not want to go to that level of detail. Our primary concern is with establishing a model which the user can safely use as a worst-case model of what happens at runtime.

2.5 Intermediate Languages

The Kit compiles Standard ML programs via a sequence of typed intermediate languages into KAM instructions, which in turn are compiled into ANSI C or to HP PA-RISC assembly language. The intermediate languages we shall refer to in the following are (in the order in which they are used in the compilation process):

Lambda A lambda-calculus like intermediate language. The main difference between the Standard ML Core Language and *Lambda* is that the latter only has trivial patterns.

RegionExp Same as *Lambda*, but with explicit region annotations (such as the `letregion`-bindings mentioned in Section 2.3). Region variables have their runtime type (Section 2.2) as an attribute, although, for brevity, the pretty printer omits runtime types when printing expressions, unless instructed otherwise.

MulExp Same as *RegionExp*, but now every binding region variable occurrence is also annotated with a multiplicity (Section 2.1) in addition to a runtime type. Again, the default is that the runtime type is not printed. The terms of *MulExp* are polymorphic in the information that annotate the nodes of the terms. That way, *MulExp* can be used as a common intermediate language for a number of the internal analyses of the compiler which add more and more information on the syntax tree. The analysis which computes multiplicities is called the *multiplicity analysis*.

The Kit contains a *Lambda-optimiser* which will happily rewrite *Lambda*-terms when it is clear that this results in faster programs (as long as the transformations cannot lead to increased space usage).

Region inference takes *Lambda* to be the source language. Region Inference happens after the lambda-optimiser has had a go at the lambda term. Therefore, it wasn't really true when we said that region inference simply annotates source programs; we ignored the translation from SML to *Lambda* and the *Lambda* optimiser. Thus one has to get used to (mostly minor) differences between the source language and the intermediate languages of the compiler if one wants to read programs in their intermediate forms.

When we want to show the result of the analyses, we usually show a *MulExp* expression.

2.6 Runtime System

The runtime system is written in C. It is small (less than 100Kb of code when compiled). It contains operations for allocating and de-allocating regions,

extending regions, obtaining more space from the operating system, recording region profiling information and performing low-level operations on strings.

It is possible to call C functions from ML Kit code. The Kit takes care of the memory allocation, by allocating regions before the call and de-allocating regions after the call. The C functions can build ML data structures such as lists through abstract operations provided by the Kit runtime system. C functions have to obey certain restrictions, see Chapter 17 for further details.

Part II

Understanding Regions

Chapter 3

Records and Tuples

In this chapter we describe construction of records and selection of record components. We also use records to introduce *region-annotated types* and *effects* which are crucial for understanding when regions are allocated and de-allocated.

3.1 Syntax

As part of the SML to *Lambda* translation, all SML records and SML tuples are compiled into *Lambda*-tuples. The components of *Lambda*-tuples are numbered from left to right, starting from 0. Selection is a primitive operation, both in *Lambda* and in the other intermediate languages. This primitive is printed using ML notation `#`i. Components are numbered from 0: the ith components of a tuple of type $\tau_1 * \ldots * \tau_n$ is accessed by `#`i, for $0 \leq i \leq n-1$.

The tuple constructor in *Lambda* is written as in SML:

$$(e_1, \ldots, e_n)$$

However, the corresponding expression in *RegionExp* and *MulExp* takes the form

$$(e_1, \ldots, e_n)\ \texttt{at}\rho$$

where ρ is a region variable indicating where the tuple should be put. In the case $n = 0$, the `at`ρ is not printed, since the empty tuple is not allocated: it is just a constant which fits in a KAM register.

Records are evaluated left to right.

3.2 Example: Basic Record Operations

Consider the source program

```
val xy = ((),())
val x = #1 xy;
```

Here is the resulting *MulExp* program:[1]

```
let val xy = ((), ()) at r1; val x = #0 xy
in  {|xy: (_,r1), x: (_,r2)|}
end
```

There are several things to note from this example.

1. The *MulExp* program contains two free region variables `r1` and `r2`. Note that the construction of the pair `xy` has been annotated by "`at r1`", indicating where the pair should be put. Similarly, `r2` is the place of `x` (although, as we shall see below, this does not denote a real region);

2. The expression {`|xy:(_,r1), x:(_,r2)|`} is an example of a *frame expression*. A frame enumerates the components that are exported from a compilation unit.

3.3 Region-Annotated Types

ML type inference infers a type for every expression in the program. Region-inference extends this idea by inferring for each expression a *region-annotated type*. The region-annotated type of an expression is the ML type of the expression decorated with extra region information. In a region-annotated type, every type constructor (e.g., `int`, `unit` and `list`) is paired with a region variable, indicating where the value is going to be put at runtime.

The following are examples of region-annotated types

$\boxed{(\texttt{int}, \rho)}$ The type of integers in region ρ.

[1] project: `kitdemo/proj`, file `kitdemo/projection.sml`.

$\boxed{(\texttt{unit}, \rho)}$ The type of 0-tuples in region ρ. Integers and 0-tuples are represented unboxed at runtime (rather than being stored in regions).[2] The types `unit`, `bool` and `int` are always decorated with one particular region variable, `r2`.

$\boxed{((\texttt{int}, \rho_1) * (\texttt{string}, \rho_2), \rho_3)}$ Denotes pairs in ρ_3 whose first component is an integer in ρ_1 and whose second component is a string in region ρ_2.

A pair of a region-annotated type and a region variable is called a "(region-annotated) type and place". We use μ to range over types and places

$$\mu ::= (\tau, \rho)$$

One can get the Kit to print the region-annotated types it infers for binding occurrences of variables. The above example then becomes

```
let val xy:(((unit,r2)*(unit,r2)),r1) = ((), ()) at r1;
    val x:(unit,r2) = #0 xy
in  {|xy: (((unit,r2)*(unit,r2)),r1), x: (unit,r2)|}
end
```

3.4 Effects and `letregion`

Here is an example of an SML program which first creates a pair and then selects a component of the pair, after which the pair is garbage:[3]

```
val n = let
           val pair = if true then (3+4, 4+5)
                      else (4, 5)
        in
           #1 pair
        end;
```

The Kit compiles the declaration into the *MulExp* program shown in Figure 3.1.[4] The compiler compiles the program as it is, without reducing the conditional to its `then` branch. During evaluation, a region (denoted by

[2] To *box* a value means to store the value in memory and represent it by its address. Values which are kept in registers are said to be *unboxed*.

[3] Project: `kitdemo/effect`, file: `kitdemo/elimpair.sml`.

[4] In general, the *Lambda*-optimiser performs various optimisations; elimination of case analyses whose outcome are known statically is not one of them.

```
let val n =
        letregion r7:1
        in let val pair =
                   (case true
                      of true => (3 + 4, 4 + 5) at r7
                       | false => (4, 5) at r7
                   ) (*case*)
           in  #0 pair
           end
        end
in  {|n: (_,r2)|}
end
```

Figure 3.1: Region inference decides that the pair is to be allocated in a local, finite region; the region will be de-allocated as soon as the pair becomes garbage.

`r7`) is introduced before the pair is allocated; it remains on the region stack till the projection of the pair has been computed, after which the region is de-allocated.

The "`:1`" on the binding occurrences of `r7` is a multiplicity indicating that there is only one store operation into the region. (The multiplicity analysis has discovered that there is at most one store from the `then` branch and at most one store from the `else` branch and that at most one of the branches will be chosen.) Thus the pair will be allocated in a little region on the runtime stack.

But how does the Kit know that it is safe to de-allocate `r7` where the `letregion` ends?

The answer lies in the fact that the Kit infers for every expression not just a region-annotated type, but also a so-called *effect*. An effect is a finite set of atomic effects. Two forms of atomic effect are $\mathbf{put}(\rho)$ and $\mathbf{get}(\rho)$, where ρ as usual ranges over region variables. $\mathbf{put}(\rho)$ indicates that a value is being stored in region ρ and $\mathbf{get}(\rho)$ indicates that a value is being read from region ρ. In our example, the region inference algorithm considers the sub-expression $e_0 =$

```
let val pair =
```

```
            (case true
               of true => (3 + 4, 4 + 5) at r7
                | false => (4, 5) at r7
            ) (*case*)
in  #0 pair
end
```

and finds that it has region-annotated type $(\texttt{int}, \texttt{r2})$ and effect

$$\{\mathbf{put}(\texttt{r7}), \mathbf{get}(\texttt{r7}), \mathbf{put}(\texttt{r2}), \mathbf{get}(\texttt{r2})\}.$$

The atomic effects on `r2` stem from the integer operations: all integers are put into the same virtual region.

Whenever a region variable occurs free in the effect of an expression but occurs free neither in the region-annotated type of the expression nor in the type of any program variable which occurs free in the expression then that region variable denotes a region which is only used locally within the expression. That this is true is of course far from trivial, but it has been proved for a skeletal version of *RegionExp*. Consequently, when this condition is met, the region inference algorithm wraps a `letregion`-binding of the region variable around that expression.

In our example, there are no free variables in e_0; moreover, `r7` occurs in the effect of e_0 but not in the region-annotated type of e_0. Thus the region inference algorithm inserts a `letregion`-binding of `r7` around e_0.

3.5 Runtime Representation

A record with 0 components (the value of type `unit`) is stored in a KAM register, not in a region. A record with n components ($n \geq 2$) takes up precisely n 32-bit words in a region; the tuple is represented by the (32-bit) address of the first component of the tuple. Note that the Kit boxes tuples. However, records are not tagged. Avoiding tags is possible, because (a) there is no pointer tracing garbage collection; and (b) polymorphic equality is compiled into monomorphic equality functions that do not have to examine the type of objects at runtime.

Lambda, *RegionExp* and *MulExp* allow one to express unboxed tuples, also in the case of function calls and returns, but the Kit does not (yet) have a boxing analysis which exploits it, nor does the code generator generate code

for unboxed tuples, multiple function arguments or multiple function return values.

A tuple is not allocated until its components have been evaluated.

3.6 A First Session with The Kit

The Kit is a batch compiler. A *project* consists of a number of SML source files, enumerated in a *project file*. Executing a project consists of first compiling the project giving your so-called script file as an argument and then running the generated target program.

3.6.1 Getting your own Script File

To compile a project, you need your own personal *script file*. Your script file contains your personal preferences concerning where source files should be read from, where target programs should be put, what should be printed on log files, what format should be used when printing programs etc. When you start the Kit, you give your script file as an argument.

If you have built the executable Kit yourself, there will be a script file called `kit.script` in the same directory as you instructed the build program to put the executable Kit in and you can use that script file as is. Otherwise, obtain a script file from a friend or from the Web site mentioned in the Preface and modify it as described in Section 16.4.

3.6.2 Compiling a Project

To compile a project, you need an executable version of the Kit; let us assume it is available on your system as a UNIX program called `kit`.

To compile Example 3.2, start the Kit with the shell command

`kit -script` *script*

where *script* is the name of your script file.

After the Kit has uttered various greetings, you will find yourself in a rudimentary menu-driven dialogue, see Figure 3.2. First, you are going to ask the Kit to print one of the intermediate forms that arise under compilation (this is how the annotated programs shown in this section were obtained). Choose `Printing of intermediate forms` (i.e., type `1` followed by carriage

```
        0       Project.....................  >>>
        1       Printing of intermediate forms  >>>
        2       Layout......................  >>>
        3       Control.....................  >>>
        4       File........................  >>>
        5       Profiling...................  >>>
        6       Test environment............  >>>
        7       Debug Kit...................  >>>
        8       Compile an sml file.........  >>>
        9       Compile it again............ ("dummy") >>>

Toggle line (t <number>), Activate line (a <number>), Up (u), or Quit(q):

>
```

Figure 3.2: The top-most Kit menu

return), and then `print drop regions expression` to toggle on the printing of the *MulExp* program. Go up one level in the menu tree by typing `u` followed by return, and you are back in the main menu.

Before proceeding, check the `File` menu to see that the `source_directory` is set to the `kitdemo` directory (which is part of the distribution). The settings you see are the ones that come from the script file. You can change them, if you want to. Finish by going back up to the main menu.

Next, you are going to interact with the separate compilation system. Select the `Project` menu. Choose `Set project file name`; then type `"proj"` (including the quotes) followed by return. Here `proj` is a project file which contains the names of the files one wants to compile. Our project consists of two files `kitdemo/prelude.sml` and `kitdemo/projection.sml`. The prelude must be included in all projects (not just demonstration programs). The file `projection.sml` contains the ML declaration shown in Section 3.2.

Then select `Read the project file`, and then `Show project status`. This will list the program units that are in the project. The first column gives the file name of the source code of the program unit (the extension `.sml`, which is required for all source files, is not printed). The second column shows the status of the program unit. `new` means that it has not yet been

compiled.

Now select `Compile and link project`. If you inspect the status of the project by selecting `Show project status` you will see that the two units have been compiled. The Kit tells you where it puts the target files it creates.

For each source file *f*`.sml` the Kit produces a *log file* *f*`.log`. You will find the output shown in Section 3.2 in the log file `projection.log` after the heading

```
Report:  AFTER DROP REGIONS:
```

Go up one on the menu tree. Printing of the region-annotated types can now be done by selecting `Layout` from the main menu, and then `print types`.

Thereafter, go back to the `Project` menu, select `Touch program unit` and enter the string `"projection"`, to indicate that you want to recompile that program unit. Select `Compile and link project` again. This will bring the project up to date. One can then inspect the log file again to see the region-annotated types.

Next, you can try the example in Section 3.4: select `Set project file name`, enter `"effect"`, select `Read the project file`, and then `Compile and link project`.

3.6.3 Running a Target Program

If no errors were found during compilation, the Kit produces a *target program* in the form of an executable file, called `run`. The Kit places `run` in `target_directory`, which is defined in the script file. As mentioned in the previous section, you can change the value of `target_directory` interactively from the `File` menu in the Kit, before compiling your project.

Running the target program is done from the UNIX shell by changing directory to the target directory and typing

```
run
```

The file will probably be around 100Kb large, even for the trivial examples considered in the chapter. This is because it contains the Kit runtime system and the compiled code for the prelude.

Running the programs presented in this chapter is not particularly exciting, since none of them produce output! However, as an exercise, try executing the `hello` project, which, like all other example files in this document, is located in the `kitdemo` directory.

Chapter 4

Basic Values

Values of types `int`, `real` and `string` are defined in accordance with the 1990 Definition of Standard ML. In due course, they will be modified to comply to the revised Definition of Standard ML and the emerging Basis Library.

4.1 Integers

Values of type `int` are represented as 32 bit integers. The following operations on integers are pre-defined:

```
infix  4 = <> < > <= >=
infix  6 + -
infix  7 div mod  *
val ~ :  int -> int
val abs: int -> int
```

At runtime, integers are represented without any form of boxing or tagging, so all 32 bits are available. Integers are kept in KAM registers or, when necessary, on the runtime stack.

For uniformity, region inference pairs all type constructors with a region variable. In the case of integers, where no region is required for storing the values, a fixed region, `r2`, is used throughout the program.

At present, arithmetic operations do not raise any exceptions when their result is undefined or out of range.

4.2 Reals

The prelude defines the following operations on reals:

```
infix  4 = <> < > <= >=
infix  6 + -
infix  7  * /
val ~ : real -> real
val abs: real -> real
val real: int -> real
val floor : real -> int
val sqrt  : real -> real
val exp   : real -> real
val ln    : real -> real
val sin   : real -> real
val cos   : real -> real
val arctan: real -> real
```

Values of type `real` are implemented as 64 bit floating point numbers. They are always boxed, i.e., represented as a pointer to two consecutive 32 bit words. These two words reside in a region and start on a double-aligned address. For that reason, regions with runtime type `real` (Section 2.2) are never unified with regions of any other runtime type.

A real constant c in the source program is translated into an expression of the form c `at` ρ, where ρ is a region variable, indicating the region into which the real will be stored.

At present, arithmetic operations do not raise any exceptions when their result is undefined or out of range.

4.3 Strings

The prelude defines the following operations on strings:

```
infix  4 =
infix  6 ^
val ord: string -> int
val chr: string -> int
val size: string -> int
```

```
val explode: string -> string list
val implode: string list -> string
val ^ : string * string -> string
```

A string is represented by a 32 bit pointer into an infinite region. The string is stored in consecutive bytes in the region, except if the size of the string exceeds the length of one region page, in which case the string is split into smaller strings which are linked together. This is completely transparent to the programmer, who does not have to worry about the actual size of region pages.

Calls of `ord`, `chr` and `size` take constant time and space. Calls of `explode`, `implode` and `^` take time and space proportional to the sum of the size of their input and their output.

The string operations can raise exceptions, as detailed in the 1990 Definition.

4.4 Booleans

The boolean values `true` and `false` are represented unboxed. Size: one word.

Chapter 5

Lists

Section 5.1 gives a summary of the list concept in Standard ML, introduces the notion of the *auxilary pairs* of a list and presents the syntax of constructors and destructors in the intermediate languages. Section 5.2 introduces region-annotated list types and show how they correspond to the layout of lists in memory. Section 5.3 gives a small example.

5.1 Syntax

In Standard ML all lists are constructed from the two constructors `::` (read: cons) and `nil`. As a shorthand, one can write `[`exp_1`,` $\cdots$`,`exp_n`]` for

$$exp_1\texttt{::}\ \cdots\ \texttt{::}exp_n\texttt{::nil}$$

which in turn is short for

$$\texttt{op ::(}exp_1\texttt{, }\cdots\texttt{, op ::(}exp_n\texttt{,nil)}\cdots\texttt{)}$$

where *exp* ranges over expressions. The type schemes of `nil` and `cons` are

$$\texttt{nil} \mapsto \forall\alpha.\alpha\texttt{list} \qquad \texttt{::} \mapsto \forall\alpha.\alpha * \alpha\texttt{list} \to \alpha\texttt{list}$$

In particular, note that `::` is always applied to a pair. The construction of the pair and the application of `::` should not be confused: the pair and the constructed value are separate values. For example, the declaration

```
val p = (2, nil)
val mylist = (op ::) p
val n = #1 p
```

is legal in Standard ML. We refer to the pairs to which :: is applied as *auxiliary pairs (of the list data type)*.

Decomposition of list values in Standard ML is done by pattern matching. A pattern can extract the pair to which :: is applied. Pattern matching on pairs can then give access to the components of the pair.

```
val abc = ["a", "b", "c"]
val op :: p = abc (* binds p to the pair ("a", ["b","c"]) *)
val (x::y::_) = abc  (* binds x to "a" and y to "b" *)
```

In the last declaration, the pattern `(x::y::_)` is short for the pattern

$$\texttt{(op ::(x, op ::(y, \_)))},$$

which combines decomposition of constructed values with decomposition of pairs.

Many ML implementations represent lists differently by always packing the constructor and the pair into one value.

The intermediate languages *Lambda* and *RegionExp* have SML-like constructs for applying constructors, but they decompose constructed values by applying a deconstructor primitive, not by pattern matching.

Lambda	*RegionExp*	
`nil`	`nil at` ρ	store nil in region ρ
`::` (e)	(`::at`ρ) (e)	create cons cell in region ρ
`decon_::` (e)	`decon_::` (e)	cons decomposition

In *Lambda*, which has essentially the same type system as SML, `decon_::`, the decomposition function for ::, has type $\forall\alpha.\alpha list \rightarrow \alpha * \alpha list$. In addition, *Lambda* and *RegionExp* have a simple case construct:

$$\texttt{(case } e \texttt{ of :: => } e_1 \texttt{ | \_ => } e_2\texttt{)}$$

where e must have list type.

5.2 Region-Annotated List Types

In Standard ML all elements of a given list must have the same type. We extend this constraint to region inference by saying that all values in the same list must reside in the same region(s), that all the constructed values

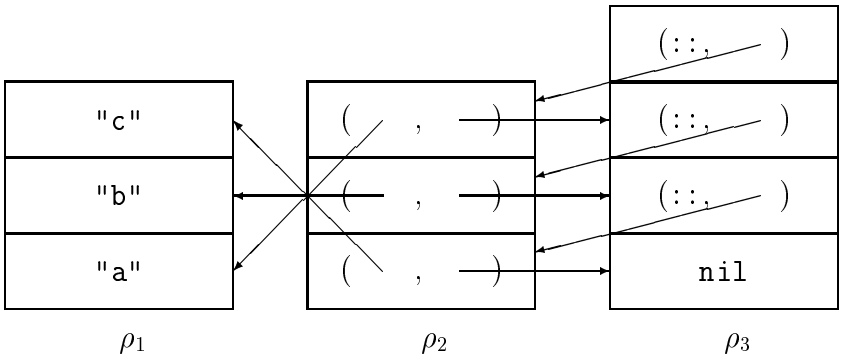


Figure 5.1: Layout of the list `["a","b","c"]` : $(((\texttt{string}, \rho_1), [\rho_2])\texttt{list}, \rho_3)$ in memory. The spine of the list resides in ρ_3 while the auxiliary pairs reside in ρ_2. Each auxiliary pair takes up two words; each constructed value $(::, \_)$ takes up two words; `nil` takes up one word.

must reside in one region and that all auxiliary pairs :: must reside in the same region.

Thus region inference does not distinguish between a list and its tail. Indeed a typical use of an infinite region is to hold the *spine* of a list, i.e., all the :: cells and the `nil` cell of the list. For an example, Figure 5.1 shows how the list `["a","b","c"]` is laid out in memory.

In general, the region-annotated type and place of a list takes the form

$$((\mu, [\rho_2])\texttt{list}, \rho_3)$$

where $\mu = (\tau_1, \rho_1)$ is the type and place of the members of the list, ρ_3 is the region where the spine of the list resides, and ρ_2 is the region where the auxiliary pairs of the list are stored. For example, the type

$$(((\texttt{string}, \rho_1), [\rho_2])\texttt{list}, \rho_3)$$

classifies lists which has their spine in ρ_3, auxiliary pairs in ρ_2 and strings in a region ρ_1.

Not all lists need to live in the same regions! Formally, `nil` and :: have

the following region-annotated types:

$$\begin{array}{rcl} \texttt{nil} & \mapsto & \forall\alpha\rho_1\rho_2.((\alpha,\rho_1),[\rho_2])list \\ \texttt{::} & \mapsto & \forall\alpha\rho_1\rho_2\rho_3\epsilon.((\alpha,\rho_1)*(((\alpha,\rho_1),[\rho_2])list,\rho_3),\rho_2) \xrightarrow{\epsilon.\{\mathbf{put}(\rho_3)\}} \\ & & \qquad (((\alpha,\rho_1),[\rho_2])list,\rho_3) \end{array}$$

Despite its verbosity, the type scheme for :: deserves careful study. It is polymorphic not just in types (signified by the bound type variable α) but also in region variables (signified by the bound ρ_1, ρ_2 and ρ_3). The ϵ is a so-called *effect variable*. The $\epsilon.\{\mathbf{put}(\rho_3)\}$ appearing on the function arrow is called an *arrow effect*. Occurring in a function type, an arrow effect describes the effect of applying the function. In this case, the effect is to create a constructed value in ρ_3, hence the effect is the singleton set $\{\mathbf{put}(\rho_3)\}$. The effect variable ϵ is used for expressing dependencies between effects (examples follow in Chapter 13). Due to the fact that the variables are universally quantified, every occurrence of :: can, potentially, be in its own region. But notice that the type of :: forces the element, which is consed onto the list, to be in the same region (ρ_1) as the already existing elements of the list. Similarly, the type forces the pairs to be in one region (ρ_2) and the spine cells to be in one region (ρ_3).

5.3 Example: Basic List Operations

The Kit compiles this program[1]

```
let val l = [1, 2, 3];
    val (x::_) = l
in x end;
```

into the *RegionExp*-program shown in Figure 5.2.

[1] Project file `kitdemo/lists`, file `kitdemo/onetwothree.sml`.

```
let val it =
        letregion r7:INF, r8:INF
        in let val l =
                   let val v2290 =
                           (1,
                            let val v2291 =
                                    (2,
                                     let val v2292 =
                                             (3,
                                              nil at r7
                                             ) at r8
                                     in  :: at r7 v2292
                                     end
                                    ) at r8
                            in  :: at r7 v2291
                            end
                           ) at r8
                   in  :: at r7 v2290
                   end
           in  (case l
                  of :: =>
                     let val v2287 = decon_:: l
                     in  #0 v2287
                     end
                   |  _ => raise Bind
               ) (*case*)
           end
        end
in  {|it: (_,r2)|}
end
```

Figure 5.2: Example showing construction and deconstruction of a small list. Layout of the list `l` is analogous to Figure 5.1. The infinite regions `r7` and `r8` hold the spine of the list and the auxiliary pairs, respectively.

Chapter 6

First-Order Functions

In this chapter we shall treat functions which are declared with `fun` and which are first-order (i.e., they neither take functions as arguments nor produce functions as results). Higher-order functions are treated in Chapter 13. Region polymorphism works uniformly over all types; we use lists as an example of the general scheme.

6.1 Region-Polymorphic Functions

It would be a serious limitation if all lists produced by a function were stored in the same region, for then all those lists would have to be kept alive till the last time one of them were used. The solution which the Kit offers to this problem is *region-polymorphic functions*, i.e., functions which are passed regions at runtime.

When one declares a function which, when called, produces a fresh list, the region inference algorithm will automatically insert extra formal region parameters in the function declaration. At every place one refers to the function, for example because one calls the function, the region inference algorithm inserts a list of actual region parameters thus telling the function where to put its result. This is all done automatically: the user does not have to introduce region parameters or pass them as arguments. But it is useful to understand the general principle so that one can exploit the feature fully.

The syntax of a (single) function declaration in *MulExp* is:

`fun` f `at` ρ_0 `[`ρ_1`,` $\cdots$`,` ρ_k`]` x `=` e

Here ρ_0 denotes the region in which the closure for f is stored, $\rho_1, \ldots, \rho_k$ are the *formal region parameters*, x is the value parameter (a single variable) and e is the body of the function. A call to f takes the form

$$f\ \texttt{[}\rho'_1\texttt{,}\ \cdots\texttt{,}\ \rho'_k\texttt{]}\ \texttt{at}\ \rho'_0\ e'$$

where $\texttt{[}\rho'_1\texttt{,}\ \cdots\texttt{,}\ \rho'_k\texttt{]}$ is a record of *actual region parameters*, ρ'_0 is the region where this record is stored, and e' is an expression denoting the argument to the call. Note that region parameters are enclosed in angle brackets (`[ ]`); this should not cause confusion with ML lists, since *RegionExp* and *MulExp* do not use the angle brackets for lists.

Different calls of f can use different actual regions, and this is essential for obtaining good separation of lifetimes.

For an example, consider[1]

```
fun fromto(a, b) = if a>b then []
                   else a :: fromto(a+1, b)
val l = #1(fromto(1,10), fromto(100,110));
```

The corresponding *MulExp*-program is shown in Figure 6.1. Note that `r7` and `r8` are formal regions of `fromto`. In the last call of `fromto`, a record consisting of region descriptors for `r20` and `r21` are passed to `fromto`; the region record is stored in `r22`. Note that the regions that hold the two lists generated by this program are disjoint. The reason that `r1` is passed twice to `fromto` in the call `fromto[at r1,at r1] at r17 (1, 10) at r19` is that, for reasons to do with separate compilation, the Kit only has one region for global values that are not of runtime type `string`, `real` or `word`. Thus `r1` holds both the pairs and the spine of the first list. In the second list, which does not escape to top level, the pairs and the spine are kept separate, in `r21` and `r20`, respectively.

6.2 Region Type Schemes

A *region-polymorphic type scheme* takes the form

$$\sigma ::= \forall \alpha_1 \cdots \alpha_n \rho_1 \cdots \rho_k \epsilon_1 \cdots \epsilon_m . \tau$$

[1] Project `kitdemo/fromto`, file `kitdemo/fromto.sml`.

```
let fun fromto at r1 [r7:INF, r8:INF] (var263)=
            let val a = #0 var263; val b = #1 var263
            in  (case a > b
                   of true => nil at r7
                   |  false =>
                      let val v2725 =
                              (a,
                               letregion r13:1,
                                         r15:1
                               in fromto[r7,r8] at r13
                                  (a + 1,
                                   b
                                  ) at r15
                               end
                              ) at r8
                      in  :: at r7 v2725
                      end
                ) (*case*)
            end ;
    val l =
        let val v2737 =
                letregion r17:1, r19:1
                in fromto[r1,r1] at r17
                   (1, 10) at r19
                end
            val _not_used =
                letregion r20:INF, r21:INF
                in let val v2738 =
                           letregion r22:1, r24:1
                           in fromto[r20,r21] at r22
                              (100, 110) at r24
                           end
                   in  ()
                   end
                end
        in  v2737
        end
in  {|fromto: (_,r1), pair: (_,r1)|}
end
```

Figure 6.1: The region-annotated version of `fromto` shows that `fromto` is region-polymorphic.

where $\alpha_1, \ldots, \alpha_n$ are type variables, $\rho_1, \ldots, \rho_k$ are region variables, $\epsilon_1, \ldots, \epsilon_m$ are effect variables and τ is a region-annotated type.

The types of `nil` and `::` in Section 5.2 are examples of region-polymorphic type schemes.

There is a close connection between, on the one hand, the formal and actual region parameters found in *RegionExp* (and *MulExp*) programs, and, on the other hand, the region type schemes which the region inference algorithm assigns to recursively declared functions. The formal region parameters of a function stem from the bound region variables of the region type scheme of that function. The actual region parameters which annotate a call of the function are the region variables to which the bound region variables are instantiated at that particular application.

For example, the region type scheme of `fromto` from Figure 6.1 is

$$\forall\rho_7\rho_8\rho_6\epsilon.((\texttt{int},\rho_2)*(\texttt{int},\rho_2),\rho_6) \xrightarrow{\epsilon.\{\mathbf{get}(\rho_2),\mathbf{put}(\rho_2),\mathbf{get}(\rho_6),\mathbf{put}(\rho_7),\mathbf{put}(\rho_8)\}} (((\texttt{int},\rho_2)[\rho_8])\texttt{list},\rho_7)$$

At the last call of `fromto` in Figure 6.1, the type scheme is instantiated to the type and place

$$((\texttt{int},\rho_2)*(\texttt{int},\rho_2),\rho_{24}) \xrightarrow{\epsilon'.\{\mathbf{get}(\rho_2),\mathbf{put}(\rho_2),\mathbf{get}(\rho_{24}),\mathbf{put}(\rho_{20}),\mathbf{put}(\rho_{21})\}} (((\texttt{int},\rho_2)[\rho_{21}])\texttt{list},\rho_{20})$$

The instantiation of bound variables of the type scheme which achieves this is

$$\{\rho_7 \mapsto \rho_{20}, \rho_8 \mapsto \rho_{21}, \rho_6 \mapsto \rho_{24}, \epsilon \mapsto \epsilon'\}$$

In general, the actual region parameters annotating a call of a region-polymorphic function are obtained from the range of the substitution by which the type scheme of the function is instantiated at that application.

To avoid passing regions that are never used, the Kit only introduces formal region variables for those bound region variables in the type scheme for which there appears at least one **put** effect in the type of the function. Reading a value is done simply by following a pointer to the value, irrespective of which region the value resides in, whereas storing a value in a region uses the name (Section 2.1) of the region. This explains why ρ_6 does not become a formal region parameter and why ρ_{24} is not passed to `fromto` at the call site. This optimisation, which is called *dropping of regions*, is the key reason why the Kit takes the trouble to distinguish between **put** and **get** effects.

Region-polymorphic functions also have to be allocated somewhere. Therefore, the region information associated with a region-polymorphic function is a *region type scheme and place*, i.e., a pair (σ, ρ). Indeed every binding occurrence of a variable (whether the binding is done by `fun`, `let` or `fn`) associates a region type scheme and place with the binding occurrence. (In the case of `let`, the type scheme will have no quantified region and effect variables, however, and in the case of `fn`, the type scheme will have no quantified variables at all.) In the following, when we refer to "the region type (scheme) and place" of some variable, we mean the region type (scheme) and place which is associated with the binding occurrence of the variable. The region type scheme should be clearly distinguished from instances of the type scheme which decorate non-binding occurrences of the variable.

6.3 Endomorphisms and Exomorphisms

The `fromto` function from Section 6.2 has the property that it can put its result in regions that are separate from the regions where its argument lies. This is not surprising, if one looks at the declaration of the function: it creates a brand new list which does not share with the argument (a, b), except for the integers a and b which may end up in the list. The freshness of the generated list is also evident from the region type scheme of the function: different region variables are used for the argument and the result.

Not all region-polymorphic functions create brand new values. Very often, a region-polymorphic function simply adds values to regions which are determined by the argument to the function. A good example is the list append function from the prelude:

```
infixr 5 @
fun [] @ ys = ys
  | (x::xs) @ ys = x :: (xs @ ys)
```

Append successively conses the elements of the first list onto the second list. Thus `ys` and `xs @ ys` must be in the same regions. However, `xs` and `ys` need not be in the same regions, although the elements of `xs` and `ys` clearly must be in the same regions, since they end up in the same list. These properties of append are summarised in the inferred region type scheme:

$$\forall\alpha\rho_1\rho_2\rho_3\rho'_1\rho'_2\rho_4\epsilon.((((\alpha,\rho_3),[\rho_2])\texttt{list},\rho_1)*(((\alpha,\rho_3),[\rho'_2])\texttt{list},\rho'_1),\rho_4)$$

$$\xrightarrow{\epsilon.\{\mathbf{get}(\rho_4),\mathbf{get}(\rho_1),\mathbf{get}(\rho_2),\mathbf{put}(\rho_1'),\mathbf{put}(\rho_2')\}}$$
$$(((\alpha,\rho_3),[\rho_2'])\texttt{list},\rho_1')$$

One of the key things one needs to be conscious of when programming with regions is whether one want functions to create fresh values or whether one wants to add to existing regions. Adding to existing regions can of course make these regions too large and long-lived, since the entire region will be alive for as long as one of the values in the region may be needed in the future. Here are two more examples to highlight the difference between functions that can put values in fresh regions and functions that add values to existing regions:

```
fun cp1 [] = []
  | cp1 (x::xs) = x :: cp1 xs
fun cp2 (l as []) = l
  | cp2 (x::xs) = x :: cp2 xs
```

Here `cp1` can copy a list into fresh regions, whereas `cp2` always copies a list into the same region:

$$\begin{aligned}
\texttt{cp1} &\mapsto \forall\alpha\rho_1\rho_2\rho_3\rho_2'\rho_3'\epsilon.((\alpha,\rho_3),[\rho_2])\texttt{list},\rho_1) \xrightarrow{\epsilon.\{\mathbf{get}(\rho_1),\mathbf{get}(\rho_2),\mathbf{put}(\rho_1'),\mathbf{put}(\rho_2')\}} \\
&\qquad ((\alpha,\rho_3),[\rho_2'])\texttt{list},\rho_1') \\
\texttt{cp2} &\mapsto \forall\alpha\rho_1\rho_2\epsilon.((\alpha,\rho_3),[\rho_2])\texttt{list},\rho_1) \xrightarrow{\epsilon.\{\mathbf{get}(\rho_1),\mathbf{get}(\rho_2),\mathbf{put}(\rho_1),\mathbf{put}(\rho_2)\}} \\
&\qquad ((\alpha,\rho_3),[\rho_2])\texttt{list},\rho_1)
\end{aligned}$$

As we saw in Section 1.3, there are cases where it is useful to copy a list from one region into another region, in order to make it possible to de-allocate the old region. This copying can be used as a kind of programmer-controlled garbage collection in cases where garbage has accumulated in the original region.

Since it is often useful to distinguish between functions that can put their result into fresh regions and functions that simply add to regions determined by their value argument, we shall refer informally to the former functions as *region exomorphisms* and the latter as *region endomorphisms*. Note that this is not a clear-cut distinction, however. Often, functions have both an endomorphic and an exomorphic side to them. Also note that even a region exomorphic function can be forced to act as an endomorphism by the calling context. Example:

```
if true then cp1 l else l
```

Since the two branches of the conditional are required to have the same region-annotated type, `l` and `cp1 l` are forced to be in the same regions.

6.4 Polymorphic Recursion

A recursive region-polymorphic function

`fun` f `at` ρ_0 `[`ρ_1`,` $\cdots$`,` ρ_k`]` x `=` e

may call itself inside its own body (e) with regions that are different from its own formal region parameter (`[`ρ_1`,` $\cdots$`,` ρ_k`]`). This feature is called *polymorphic recursion in regions*, named after polymorphic recursion, the analogous concept for types. Polymorphic recursion in regions is vital for achieving good recursion. It is also a major source of complication of the region inference algorithm we use in the Kit, but we shall not tire the reader with the details here.

We now show a typical use of polymorphic recursion in regions, namely merge sorting of lists. The basic idea of merge sort is simple: first split the input list into two lists l and r of roughly equal length. Then sort l and r recursively and merge the results into a single sorted list. When programming with regions, we need to plan which of these lists we want to reside in the same regions. We do not want to waste space. In particular, if n is the length of the list, it would be quite irresponsible to use $O(n\log n)$ space, say. Let us aim at arranging that the sorting function is a region exomorphism which does not produce any values in its result regions except the sorted list. To sort n elements we shall need n list cells (to hold the input list) plus roughly $2 \times (n/2)$ list cells to hold l and r, the two lists that arise from splitting the input list. To sort l recursively, we need space for the two lists obtained by splitting l etc. This grows to a maximum of $3n$ list cells (including the n cells to hold the input), before any merging is done. By the time all of l is sorted, i.e., just before r is sorted recursively, we have the following lists: the input (n cells), l ($n/2$ cells), l sorted ($n/2$ cells), r ($n/2$ cells). Continuing this way, one sees that the maximal memory usage occurs at the rightmost merge to two lists of length at most one, at which point approximately $4n$ list cells are live. Here is code which uses these ideas:

```
fun cp [] =[]
  | cp (x::xs)= x :: cp xs

(* exomorphic merge *)
fun merge(xs, []):int list = cp xs
  | merge([], ys) = cp ys
  | merge(l1 as x::xs, l2 as y::ys) =
        if x<y then x :: merge(xs, l2)
        else y :: merge(l1, ys)

(* splitting a list *)
fun split(x::y::zs, l, r) = split(zs, x::l, y::r)
  | split([x], l, r) = (x::l, r)
  | split([], l, r) = (l, r)

(* exomorphic merge sort *)
fun msort []  = []
  | msort [x] = [x]
  | msort xs = let val (l, r) = split(xs, [], [])
               in merge(msort l, msort r)
               end;
```

The exomorphic merge function is a bit inefficient in that it copies one argument when the other is empty, but the exomorphism ensures that `msort l` and `msort r` are not forced into the same regions. The polymorphic recursion in regions makes it possible for `xs`, `l`, `r`, `msort l` and `msort r` all to be in distinct regions. For example, in the call `msort l`, the polymorphic recursion makes it possible for `l` to be in regions different from `xs` and it also makes it possible for the result of the call to be in a region different from the result of `msort xs`.

Based on the above analysis we conclude that the space required by `msort xs` is approximately $4nc_1 + c_2 \log_2 n$, where n is the length of `xs`, c_1 is the size of a list cell (4 words in this case) and c_2 is the space on the runtime stack used by one recursive call of `msort` (probably less than 10 words).

To check the above analysis, we sorted 50,000 integers with the region profiler enabled. According to our analysis, the maximal space usage should be roughly $4 \times 50,000 \times 4$ words, i.e., 3,200,000 bytes, i.e, $3.125MB$. As one sees in Figure 6.2, the analysis was accurate to within a kilobyte.

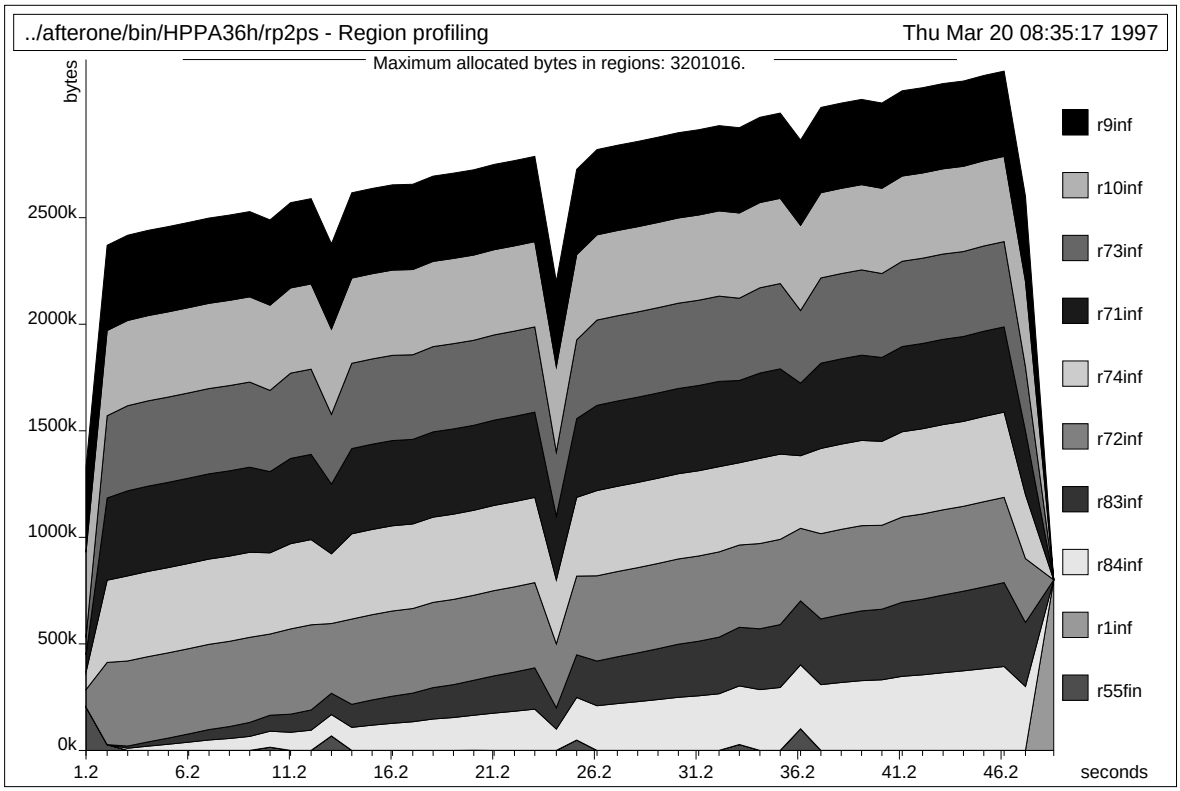


Figure 6.2: Region profiling of `msort` sorting 50,000 integers. The high-level mark of 3,201,016 bytes is exact (i.e., not sampled).

In Chapter 12 we shall see how one can use resetting of regions to reduce the space usage to roughly $2nc_1$.

The project `kitdemo/msort` contains the above declarations. After compiling the project, the region-annotated code may be found in the file `kitdemo/msort.log`.

Chapter 7

Value Declarations

Although region inference is based on types and effects, it is also to some extent syntax dependent: two programs can easily be equivalent in their input-output behaviour and yet result in very different memory behaviour. In this chapter we discuss how to write declarations in order to obtain good results with region inference. The region inference rules that underlie the ML Kit with Regions are related to the scope rules of ML, so we start by a (very informal) summary of the scope rules of ML declarations.

7.1 Syntax

A Standard ML *value declaration* binds a value to a value variable. For example, the result of evaluating the value declaration

```
val x = 3+4
```

is the environment $\{\texttt{x} \mapsto 7\}$. More generally, evaluation of a value binding `val` *id* `=` *exp* proceeds as follows. Assume the result of evaluating *exp* is a value, v. Then the result of evaluating `val` *id* `=` *exp* is the environment $\{id \mapsto v\}$.

The value declaration is just one form of Core Language declaration (the others being type and exception declarations). We use *dec* to range over declarations. Declarations can be combined in several ways. For example,

$$dec_1 \,;\, dec_2$$

is a *sequential declaration*. The identifiers declared by this declaration are the identifiers that are declared by dec_1 or dec_2; moreover, identifiers declared

in dec_1 may be referenced in dec_2. The semicolon is associative. Thus in a sequence $dec_1 ; \ldots ; dec_n$ of declarations, identifiers declared in dec_i may be referenced in $dec_{i+1}, \ldots, dec_n$ $(1 \leq i \leq n)$.

The Core Language has two forms of local declarations. The expression

`let` *dec* `in` *exp* `end`

declares identifiers whose scope does not extend beyond *exp*. Similarly, the declaration

`local` dec_1 `in` dec_2 `end`

first declares identifiers (in dec_1) whose scope does not extend beyond dec_2 and it then uses these declarations to perform the declaration in dec_2. An identifier is declared by the entire local construct if and only if it is declared by dec_2.

7.2 On the Relationship between Scope and Lifetime

Scope is a syntactic concept: a declaration of an identifier contains a binding occurrence of the identifier; the scope of the declaration is the part of the ensuing program text whose free occurrences of that identifier are bound by that binding occurrence. By contrast, lifetime, as we use the word, is a dynamic concept. A value is "live" if and only if the remainder of the computation uses it (or part of it). The traditional stack discipline couples these two concepts very closely. For example, in the pure stack discipline, the evaluation of

`let` *dec* `in` *exp* `end`

in an environment E proceeds as follows. First evaluate *dec*, yielding an environment, E_1. Then evaluate *exp* in the environment E extended with E_1, yielding value v. Then v is the result of evaluating the `let`-expression in E. In implementation terms: first push an environment E_1 onto the stack, use it to evaluate the expression in the scope of the declaration and then pop the stack. That this idea works in block-structured languages hinges on a number of carefully made language design decisions. In functional and object-oriented languages, memory cannot be managed that simply. The problem is that while environments can be managed in a stack-like manner,

the values in the range of the environment cannot (unless one uses regions, that is). For example consider the ML expression:

```
local
   val private = [2,3,5,7,11,13]
in
   fun smallPrime(n:int): bool =
          List.member n private
end
```

Although the scope of the declaration is only the declaration of `smallPrime`, `private` is accessed (at runtime) whenever `smallPrime` is called. Thus the lifetime of the list of small primes is at least as long as the lifetime of the `smallPrime` function itself.

The region discipline still has a coupling between scope and lifetimes, but, since we want to be able to handle recursive data types and higher-order functions, the coupling is less tight. The ground rule of region inference is that as long as a value variable *id* is in scope, the value bound to it at runtime will remain allocated. More precisely:

> Ground Rule: The region rules forbid transforming an expression *exp* into `letregion` ρ `in` *exp* `end` if *exp* is in the scope of an identifier which has ρ free in its region type scheme or place.

For an example, consider

```
let
   val list = [1,2,3]
   val n = length list
   val r = sin(real n)
in
   cos(r)
end
```

At runtime, the list bound to `list` is not used (i.e., it is not live) after its length has been computed; similarly, the value of `n` is not live after it has been converted to a floating point number, and so on. In short, at runtime we have a sequence of short, non-overlapping lifetimes.

With region inference, however, the list bound to `list` will stay allocated throughout the evaluation of the remainder of the `let`-expression.[1]

It is crucial to bear the ground rule in mind when programming with regions. For a more interesting example, consider the following declarations, taken from a program which computes prime numbers using the Sieve of Eratosthenes:

```
fun cp [] = []
  | cp (x::xs) = x :: cp xs

fun sift (n, []) = []
  | sift (n, (x::xs)) = if x mod n = 0 then sift(n,xs)
                        else x::sift(n,xs)
fun sieve(a as ([], p)) = a
  | sieve(x::xs, p) = let val rest = sift(x,xs)
                      in sieve(cp rest,x::p)
                      end
```

Here `sift(n, l)` produces a list of the numbers from `l` that are not divisible by `n`; `sieve(xs, p)` repeatedly calls `sift`, adding primes to the front of `p`, until the list of numbers remaining in the sieve becomes empty. The programmer has employed the copying technique suggested in Section 1.3 to avoid that the lists that are bound to `rest` during the repeated filtering all are put in the same region. The programmer's intention is that the `cp rest` should overwrite `x::xs` by a copy of `rest`, so that space consumption would be bounded by a constant times the size of the input. But it does not work as intended: since `rest` is in scope at the recursive application of `sieve`, the list which is bound to `rest` will stay allocated for the duration of that call, which is in fact the remainder of the entire computation!

In many cases, the solution is simply to shorten the scope of the declaration. In the above example, a good solution is to move the application of `sieve` outside the `let`:

[1] One can force de-allocation of the list by inserting `val _ = resetRegions(list)` after the declaration of `n`; but, as we shall see, there are less draconian ways of achieving the same result.

```
fun sieve(a as ([], p)) = a
  | sieve(x::xs, p) =
        sieve let val rest = sift(x,xs)
              in (cp rest,x::p)
              end
```

Although we cannot explain why the copying really overwrites the input list until we have dealt with resetting of regions (Chapter 12) we can explain why this transformation ensures that the list bound to `rest` will not live to see the recursive call of `sieve`. Unless forced by context to do otherwise, `sift` will create a list using fresh regions. Since `cp` is also exomorphic, there will be no sharing between `rest` and the other lists. Precisely, the two region variables that denote the two regions which hold the spine and the auxiliary pairs of `rest` appear in the effect of the (revised) `let`-expression but neither of them occur free in the region type scheme and place of any variable in scope at that point, not even in the region type scheme and place of `sieve`, whose only *free* region variables are the global integer region `r2` and the region which contains `sieve` itself. Consequently, region inference will wrap the `let`-expression by a `letregion`-binding of the two region variables in question, e.g.,

```
fun sieve(a as ([], p)) = a
  | sieve(x::xs, p) =
        sieve letregion r10, r11
              in let val rest = sift[r10,r11](x,xs)
                 in (cp rest,x::p)
                 end
              end
```

7.3 Summary

Informally, region inference forces lifetime to be at least "as long" as scope. However, region inference will introduce a `letregion` ρ-binding around an expression containing a free occurrence of ρ as soon as ρ occurs free neither in the type of the expression nor in the region type scheme and place of any variable in scope at the expression.

Useful program transformations to shorten lifetimes include:

1. Inwards let floating: transform

   `let val` id_1 `=` exp_1 `val` id_2 `=` exp_2 `in` exp `end`

   into

   `let val` id_2 `=` `let val` id_1 `=` exp_1 `in` exp_2 `end in` exp `end`

   provided id_1 does not occur free in exp.

2. Application extrusion: transform

   `let` dec `in` f`(`exp`)` `end`

   into

   f `let` dec `in` exp `end`

   provided f is an identifier which is not declared by dec.

These meaning-preserving transformations are useful in situations where early de-allocation is important. Application extrusion is a useful programming habbit, especially in connection with tail recursion; the reader will see it employed several times in what follows.

Chapter 8

Static detection of space leaks

"Space leak" is the informal term used when a program uses much more memory than one would expect, typically because of memory not being recycled as early as it should (or not at all).

If a region-polymorphic function with region type scheme σ has a **put**-effect on a region variable which is not amongst the bound region variables of σ then one quite possibly has a space leak: every application of the function may write values into a region which is the same for all calls of the function. For example, consider the source program[1]

```
fun g() =
  let val x = [5,7]
      fun f(y) = (if y>3 then x@x else x;
                  5)
  in
      f 1; f 4
  end;
```

Here `f` has type `int` $\rightarrow$ `int`; yet, when `y>3` evaluates to `true`, an append operation producing a list in the same region as `x` is performed. The first call of `f` will not cause the append operation to be called, but the second one will. One can say that `f` has a space leak in that it can write values into a more global region, namely a region which is allocated at the beginning of the body of `g`. Hence the sequence of calls to `f` would accumulate copies of `x@x` in that region, although none of these lists are accessible anywhere. In

[1] Project `kitdemo/escape`, file `kitdemo/escape.sml`.

this particular case, the values are not even part of the result type of `f`, so the writing is a "side-effect" at the implementation level, even though there are no references in the program.

The region type scheme inferred for `f` is:

$$\forall\epsilon.(\texttt{int},\texttt{r2}) \xrightarrow{\epsilon.\{\mathbf{put}(\texttt{r4}),\mathbf{put}(\texttt{r5}),\mathbf{get}(\texttt{r2}),\mathbf{put}(\texttt{r2})\}} (\texttt{int},\texttt{r2})$$

where the region-annotated type of `x` is

$$(((\texttt{int},\texttt{r2}),[\texttt{r5}])\texttt{list},\texttt{r4})$$

Here we see that `r4` and `r5` are free in the type scheme but appear with **put** effects.

8.1 Warnings About Space Leaks

The Kit issues a warning each time it meets a `fun`-declared function which has a free put effect occurring somewhere in its type scheme. In practice, we have found this to be an extremely valuable device for predicting space leaks. In our example, the following warning is printed on the log file:

```
fun g at r1 [] (var314)=
   letregion r8:INF, r9:INF
   in let val x =
              let val v3294 =
                      (5,
                       let val v3295 = (7, nil at r8) at r9
                       in  :: at r8 v3295
                       end
                      ) at r9
              in  :: at r8 v3294
              end
      in  letregion r12:1
          in let fun f at r12 [] (y)=
                    let val _not_used =
                            let val v3291 =
                                    (case y > 3
                                       of true =>
                                          letregion
```

```
                                        ...
                                        in @[r8,r9] at r15
                                           (x, x) at r17
                                        end
                                     |  false => x
                                  ) (*case*)
                            in  ()
                            end
                    in  5
                    end ;
                val _not_used =
                    let val v3293 =
                            letregion r18:1
                            in f[] 1
                            end
                    in  ()
                    end
            in  letregion r20:1
                in f[] 4
                end
            end
        end
    end

 *** Warnings ***
f        has a type scheme with escaping put effects on region(s):
r8, which is also free in the type (schemes) of :  x
r9, which is also free in the type (schemes) of :  x
```

We are told that the program might space leak in regions `r8` and `r9`. Looking at the function `f`, we see that these two regions are actual region parameters to `@`. This reveals that the problem is the call to `@`.

8.2 Fixing Space Leaks

Often one can fix a space leak by delaying the creation of the global value which causes the space leak. In the above example, we can move the con-

struction of the list into `f`:[2]

```
fun g() =
  let fun mk_x() = [5,7]
      fun f(y) = let val x = mk_x()
                 in if y>3 then x@x else x; 5
                 end
  in
      f 1; f 4
  end;
```

Of course, this means that the list will be re-constructed upon each application of `f`. Another solution is to move the creation of the list as close to the calls as possible and then pass the list as an extra argument:[3]

```
fun g() =
  let
      fun f(x,y) = (if y>3 then x@x else x; 5)
  in
      let val x = [5,7]
      in  f(x, 1); f(x, 4)
      end
  end;
```

Both solutions stop warnings from being printed, but the second solution is better than the first: `f` still has a put effect on the regions containing `x`, but the difference is that these are now represented by bound region variables in the type scheme of `f`. This has two advantages: (a) allocation of space for the list is delayed till the list is actually used; and (b), the list can be de-allocated after the calls have been made (whereas in the original version, `x` occurs free in the declaration of `f` and will be kept alive as long a `f` can be called).

At other times, there is no clean way of avoiding escaping put regions. One example is found in the prelude:

```
exception Io of string
exception CANNOT_OPEN
```

[2]Project: `kitdemo/escape`, file `kitdemo/escape1.sml`.
[3]Project: `kitdemo/escape`, file `kitdemo/escape2.sml`.

```
fun open_in(f: string): instream =
       INS(prim(31, ("openInStream", "openInStream", f,
                     CANNOT_OPEN)))
       handle CANNOT_OPEN => raise Io("Cannot open " ^ f)
fun open_out(f: string): outstream =
       OUTS(prim(31, ("openOutStream", "openOutStream", f,
                      CANNOT_OPEN)))
       handle CANNOT_OPEN => raise Io("Cannot open " ^ f)
```

As explained in Chapter 11, our region inference algorithm is very simple-minded about unary exception constructors: when a unary exception constructor is applied to a value, both the argument value and the resulting constructed value are forced into a global region. Thus the application `Io("Cannot open " ^ f)` has a potential space leak in it: every time we concatenate the two strings, the resulting string will be put into a global region. This particular space leak is perhaps not something that would keep one awake at night, since most programs do not make a large number of failed attempts to open files, but it is useful to be warned about this potential problem.

Chapter 9

References

Section 9.1 gives a brief summary of references in Standard ML; it may be skipped by readers who know the language. Thereafter we discuss runtime representation of references, region-annotated reference types and show examples.

9.1 References in Standard ML

A reference is a memory address (pointer). Standard ML has three built-in operations on references

`ref`	$\forall\alpha.\alpha \to \alpha$`ref`	create reference
!	$\forall\alpha.\alpha$`ref` $\to \alpha$	dereferencing
`:=`	$\forall\alpha.\alpha$`ref` $* \alpha \to$ `unit`	assignment

If the type of a reference r is τ`ref` then one can store values of type τ (only) at address r. A reference is a value and can therefore be bound to a value identifier by a value declaration (`val`$\cdots$). While the value stored at a reference may change, the binding between variable and reference does not change. We show an example, since this point can be confusing to programmers who are familiar with updatable variables in languages like C and Pascal.

```
val it = let
             val x: int ref = ref 3
             val y: bool ref = ref true
             val z: int ref  = if !y then x else ref 5
```

```
in
   z:= 6;
   !x
end
```

Since `!y` evaluates to true, `z` becomes bound to the same reference, r, as `x`. So the subsequent assignment to `z` changes the contents of the store at address r to contain 6. Since `x` and `z` are aliases, the result of the `let`-expression is the contents of the store at address r, i.e., 6.

9.2 Runtime Representation of References

The Kit translates an SML expression of the form `ref` *exp* into an expression of the form

$$\texttt{ref at } \rho \quad e$$

which is evaluated as follows. First e is evaluated. Assume this yields a value, v. Here v may be a boxed or an unboxed value. Next, a 32-bit word is allocated in the region denoted by ρ; let r be the address of this word. Then v is stored at address r and r is the result of the evaluation.

Note that a reference really is a pointer in the implementation. In particular, a reference is not tagged and may be stored in a KAM register. The contents of the reference is also one word, either an unboxed value (e.g., an integer or a boolean) or a pointer (if the contents is boxed). So the contents of a reference is not tagged either.

Dereferencing a reference r is done by reading the contents of the memory location r. Note that this does not require knowledge of what region the word with address r resides in.

Assigning a value v to a reference r simply stores v in the memory at address r. When v is an unboxed value, this can be regarded as copying v into the memory cell r; othewise v is a pointer which the assignment stores in the memory cell r. Either way, assignment is a constant-time operation.

9.3 Region-annotated Reference Types

The general form of a region-annotated reference type and place is:

$$(\mu\, \texttt{ref}, \rho)$$

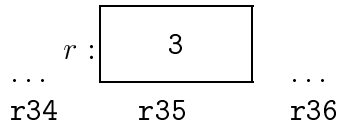


Figure 9.1: Creating a reference allocates one word in a region on the region stack. Above, the region is drawn as a finite region, but it could equally well be infinite.

Informally, a reference r has this type if it is the address of a word in the region denoted by ρ and, moreover, μ is the type and place of the contents of that word. For example, assume ρ is bound to some region name, say `r35`; then the evaluation of `val x = ref at` ρ `3` results in the environment $\{\texttt{x} \mapsto r\}$, where r is the address of a word with contents 3 residing in region `r35`, see Figure 9.1.

References are treated like all other values by region inference. The region-annotated types given to the three built-in operations are:

`ref`	$\forall\alpha\rho_1\rho_2\epsilon.(\alpha,\rho_1) \xrightarrow{\epsilon.\{\mathbf{put}(\rho_2)\}} ((\alpha,\rho_1)\texttt{ref},\rho_2)$
!	$\forall\alpha\rho_1\rho_2\epsilon.((\alpha,\rho_1)\texttt{ref},\rho_2) \xrightarrow{\epsilon.\{\mathbf{get}(\rho_2)\}} (\alpha,\rho_1)$
:=	$\forall\alpha\rho_1\rho_2\rho_3\rho_4\epsilon.(((\alpha,\rho_1)\texttt{ref},\rho_2) * (\alpha,\rho_1),\rho_3) \xrightarrow{\epsilon.\{\mathbf{get}(\rho_3),\mathbf{put}(\rho_2),\mathbf{put}(\rho_4)\}} (\texttt{unit},\rho_4)$

Note that within each of these type schemes, α is paired with the same region variable. The reason is that assigning a value v to a reference r does not make a copy of v (unless v is unboxed). The advantage of the chosen scheme for handling references is that reference creation, dereferencing and assignment all are constant-time operations. The disadvantage is that if two values may be assigned to the same reference, they are forced to be in the same regions (cf. the region type schemes given above).

If we compile the example from Section 9.1 we get the program shown in Figure 9.2.[1] The region denoted by `r7` contains the memory word whose

[1] Project `kitdemo/refs`, file `kitdemo/refs3.sml`.

```
let val it =
    letregion r7:INF
    in let val x = ref at r7 3
       in  letregion r8:1
           in let val y =
                      let val v3845 = true
                      in ref at r8 v3845 end ;
                  val z =
                      (case letregion r9:1 in ![] y end
                      of true => x | false => ref at r7 5)
                  val v3842 =
                      letregion r11:1, r13:1
                      in :=[r7] at r11 (z, 6) at r13
                      end
              in  letregion r14:1 in ![] x end
              end
           end
    end
in  {|it: (_,r2)|}
end
```

Figure 9.2: Region-annotated reference creation.

address is bound to `x` and `z`, and whose contents is first 3, then 6. The region denoted by `r8` contains a single boolean. Also note that the word containing 5 is designated `r7`, since the `then` and `else` branches must give the same type and place. Finally note that all references will be reclaimed automatically, at the end of `letregion` constructs which bind `r7` and `r8`.

9.4 Local References

References to words which are created locally within a function and do not escape the function naturally reside in regions which are local to the function body. For example, the declaration:[2]

```
fun id(x) = let val r = ref x in ! r end;
```

is compiled into

```
let fun id at r1 [] (x)=
        letregion r9:1
        in let val r = ref at r9 x
           in  letregion r10:1 in ![] r end
           end
        end
in  {|id: (_,r1)|}
end
```

Here `r9` will be implemented as one word on the runtime stack. The evaluation of `ref at r9 x` moves the contents of the standard argument register (`standardArg`) to that word on the stack. At the end of the `letregion r9 ... end`, the word is popped off the stack.

Now let us turn to an example of a memory cell whose lifetime extends the scope of its declaration, because it is accessible via a function (in Algol terminology, the reference is an *own variable* of the function.)[3]

```
local
  val r = ref ([]:string list)
in
  fun memo_id x = (r:= x:: !r; x)
```

[2]Project: `kitdemo/refs`, file `kitdemo/refs1.sml`.
[3]Project: `kitdemo/refs`, file `kitdemo/refs2.sml`.

```
        end
        val y = memo_id "abc"
        val z = memo_id "efg";
```

This compiles into

```
   let val r =
           let val v3756 = nil at r1 in ref at r1 v3756 end ;
       fun memo_id at r1[] (x) =
           let val v3752 =
                   letregion r8:1, r10:1
                   in :=[r1] at r8
                      (r,
                       let val v3753 =
                               (x,
                                letregion r12:1
                                in ![] r
                                end
                               ) at r1
                       in  :: at r1 v3753
                       end
                      ) at r10
                   end
           in  x
           end ;
       val y = letregion r14:1 in memo_id[] "abc"at r4 end
       val z = letregion r16:1 in memo_id[] "efg"at r4 end
   in  {|
       r: (_,r1),
       memo_id: (_,r1),
       y: (_,r4),
       z: (_,r4)
       |}
   end
```

and the Kit warns us that there is a possible space leak (Chapter 8):

```
 *** Warnings ***
memo_id  has a type scheme with escaping put effects on region(s):
r1, which is also free in the type (schemes) of :  ! := r Match Bind
```

9.5 Hints on Programming with References

There is no need to shy away from using references when programming with regions. However, one needs to be aware of the restriction that values that may be assigned to the same references are forced to live in the same region, and this region with all its values will be alive for as long as the reference is live. This poses no problem if the contents type is unboxed (e.g., `int`), for in that case no region for the contents is allocated at all. But one should avoid creating long-lived references which are assigned many different large values.

Chapter 10

Recursive Data Types

Standard ML permits the programmer to declare (possibly recursive) data types using the `datatype` declaration. For example, one can declare a polymorphic, recursive data type for binary trees as follows:

```
datatype 'a tree = Lf | Br of 'a * 'a tree * 'a tree;
```

10.1 Spreading Data Types

The Kit performs an analysis, called "spreading of data types", of the `datatype` declarations contained in the program. Spreading determines (a) a so-called arity of every type name which the data type declaration introduces and (b) a region type scheme for every value constructor introduced by the data type declaration. In Standard ML, every type name has an attribute, called its arity. For example, `int` has arity 0 while the type name introduced by the above declaration would have arity 1. However, the notion of arity has to be extended internally in the Kit to account for regions and effects. For lists, for example, we need not just a region for holding the constructors `nil` and `::`, but also a region for holding the pairs to which `::` is applied. For the data type

```
datatype 'a foo = A | B of ('a * 'a) * ('a * 'a)
```

the type of `B` introduces the possibility of three region variables (one for each star), even if we decide to pair all occurrences of `'a` by the same region variable. Region variables which are induced by the types of constructors

and which do not hold the constructed values themselves are called *auxiliary region variables*. For example, the `list` data type:

```
datatype 'a list = nil | op :: of 'a * 'a list
```

has one auxiliary region variable, namely the region variable which describes where the pairs of type `'a * 'a list`, i.e., the auxiliary pairs, reside.

One also needs auxiliary arrow effects, for cases such as

```
datatype V = N of int | F of V -> V
```

where we need an arrow effect for the function type `V -> V`. We refer to such an arrow effect as an *auxiliary arrow effect* of the data type in question.

We define the *(internal) arity* of a type name t to be a triple (n, k, m) of non-negative integers, where n is the usual Standard ML arity of the type name, k is the *region arity* of t and m is the *effect arity* of t. The region and effect arity indicate the number of auxiliary regions and arrow effects of the data type, respectively.

For efficiency purposes, we have found it prudent to restrict the maximal number of auxiliary regions a data type can have to 3 (one for each kind of runtime type of regions) and to restrict the maximal number of auxiliary effects to 1. Otherwise, the number of auxiliary regions can grow exponentially in the size of the program:

```
datatype t0 = C
datatype t1 = C1 of t0 * t0
datatype t2 = C2 of t1 * t1
...
```

Here the number of auxiliary region variables would double for each new data type declaration.

Furthermore, all type names introduced by a `datatype` declaration are given the same arity (a `datatype` declaration can declare several types simultaneously). Within one constructor binding (*conbind*), all occurrences of the same type variable are paired with the same region variable. Different type variables are paired with different region variables. Since we allow at most one auxiliary region variable for each `datatype` declaration, the analysis of data type declarations sometimes has to unify two auxiliary region variables that would otherwise be distinct, but it only unifies auxiliary region variables that have the same runtime type.

The practical consequence of these restrictions is that sometimes applying a constructor to a value v forces identification of regions of v that hold otherwise unrelated parts of v.

The automatic memory management we have discussed for lists extends to other recursive data types without problems. For example, binary trees are put into regions and are subsequently de-allocated (in a constant time operation) when the region is popped. The next section is an example to illustrate the point.

For simplicity, constructed values are always boxed.

10.2 Example: Balanced Trees

Consider the following program in Figure 10.1.[1] Note that we would hope that the balanced tree produced by `balpre` is removed after it has been collapsed into a list by `preord`. And indeed it is. Here is the proof:

```
val it =
    letregion r72:1, r74:1
    in say[]
       letregion r75:1, r77:INF, r78:INF, r79:INF
       in implode[r74] at r75
          letregion r80:1, r82:1, r83:INF, r84:INF
          in preord[r77,r78] at r80
             (letregion r85:1, r87:INF, r88:INF
              in balpre[r83,r84] at r85
                 letregion r89:1, r91:1
                 in explode[r87,r88,r79] at r89
                    "Greetings from the Kit\n"at r91
                 end
              end
              nil at r77
             ) at r82
          end
       end
    end
```

[1] Project `kitdemo/trees`, file `kitdemo/trees.sml`.

```
datatype 'a tree = Lf | Br of 'a * 'a tree * 'a tree

(* preorder traversal of tree *)

fun preord (Lf, xs) = xs
  | preord (Br(x,t1,t2),xs) =
      x::preord(t1,preord(t2,xs))


(* building a balanced binary tree
   from a list: *)

fun balpre [] = Lf
  | balpre(x::xs) =
     let val k = length xs div 2
     in Br(x, balpre(take(xs, k)),
              balpre(drop(xs, k)))
     end

(* preord o balpre is the identity: *)

val it = say(implode(preord(balpre(explode
    "Greetings from the Kit\n"),[])));
```

Figure 10.1: Example showing recycling of memory used for an intermediate data structure.

This is the kind of certainty about lifetimes we are aiming at. Imagine, for example, that the trees under consideration were terms representing different intermediate forms in a compiler. Then one would like to know that (possibly large) syntax trees are not kept in memory longer than needed.

Chapter 11

Exceptions

11.1 Exception Constructors and Exception Names

Standard ML exception constructors are introduced by *exception declarations*. The two most basic forms are

`exception` *excon*

and

`exception` *excon* `of` *ty*

for introducing nullary and unary exception constructors, respectively. Unary exception constructors are typically used when one wants to raise an exception which contains a "reason" (represented by a value of type *ty*).

Unlike in some languages (for example Java), exception declarations need not occur at top level. For example, a function body may contain exception declarations. Each evaluation of an exception declaration creates a fresh *exception name* and binds it to the exception constructor. This is sometimes referred to as the *generative* nature of ML exceptions.

In the ML Kit, an exception name is implemented as a pointer to a pair consisting of an integer and a string pointer; the string pointer points to the name of the exception, which is a global constant in the target program. The string is used for printing the name of the exception if it ever propagates to the top level. The cost of creating the pair is, as always with pairs, two words.

11.2 Exception Values

Standard ML has a type `exn` of *exception values*. An exception value is either an exception name or a *constructed exception value*. A constructed exception value can be thought of as a pair (en, v) of an exception name en and a value v; we refer to v as the *argument* of en.

An exception value which is just a nullary exception name is represented as the name itself, i.e., by a pointer to a pair of an integer and a string. Thus referring to a nullary exception constructor allocates no memory. By contrast, applying a unary exception constructor to an argument constructs a constructed exception value. Cost: two words.

The distinction between nullary and unary exception constructors is important in the Kit because our region inference analysis takes a simple-minded approach to exceptions: *all exception names and all constructed exception values are put in global regions and thus never reclaimed automatically.*

We therefore make the following recommendations:

1. Put exception declarations at top-level, if possible. That way, the memory required by exception names will be bounded by the program size.

2. Avoid applying unary exception constructors frequently; there is no harm in raising and handling constructed exception values frequently, it is the creation of many different constructed exception values that can lead to space leaks. Nullary constructors may be used freely without incurring memory costs.

11.3 Raising Exceptions

An expression of the form

`raise` *exp*

is evaluated as follows. First *exp*, an expression of type `exn`, is evaluated to an exception value. Then the runtime stack is scanned from top towards bottom in search of a handler which can handle the exception. The KAM has a register which points to the top-most exception handler; the exception handlers are linked together as a linked list interspersed with the other contents of the runtime stack. If a matching handler is found, the runtime stack is popped down to the handler. This popping includes popping of regions

that lie between that stack top and the handler. Put differently, consider an expression of the form `letregion` ρ `in` e `end`; if e evaluates to an exception packet, then the region bound to ρ is de-allocated and the packet is also the result of evaluating the `letregion` expression.

We have not attempted to design an analysis which would estimate how far down the stack a given exception value might propagate. Of course, it would not be a very good idea to allocate a constructed exception value in a region which is popped before the exception is handled! This is why we put all exception names and all constructed exception values in global regions.

11.4 Handling Exceptions

The ML expression form

$$exp_1 \texttt{ handle } match$$

is compiled into a *MulExp* expression of the form

```
letregion ρ in
 let f = fn match at ρ in e1 handle f end
end
```

where f is a fresh variable. So first a handler (expressed as a function) is evaluated and stored in some region ρ. This region will always have multiplicity one and therefore be a finite region which is put on the stack. Then e_1, the result of compiling exp_1, is evaluated. If e_1 terminates with a value, the `letregion` construct will take care of de-allocating the handler. If e_1 terminates with an exception, however, f is applied.

Thus the combined cost of raising an exception and searching for the appropriate handler takes time proportional to the depth of the runtime stack in the worst case.

This is the only operation which takes time which cannot be determined statically, provided one admits arithmetic operations as constant-time operations.

11.5 Example: Prudent Use of Exceptions

Here is an example of prudent use of exceptions in the ML Kit:

```
exception Hd                (* recommendation 1 *)

fun hd [] = raise Hd
  | hd (x::_) = x

exception Tl

fun tl [] = raise Tl
  | tl (_ ::xs) = xs

exception Error of string

local
  val error_f = Error "f"  (* recommendation 2 *)
in
  fun f(l) =
      hd(tl(tl l)) handle _ => raise error_f
end

val r = f[1,2,3,4];
```

Note that the application `Error "f"` has been lifted out from the body of `f`. No matter how many times `f` is applied, it will not create additional exception values.[1]

[1] Project: `kitdemo/exceptions`, file: `kitdemo/exceptions.sml`.

Chapter 12

Resetting Regions

In Section 1.2 we explained that resetting regions is an important ingredient in programming with regions. This chapter gives an informal explanation of the rules that govern resetting, rules which play a key rôle in Kit programming irrespective of whether one leaves resetting of regions to the Kit or prefers to control resetting explicitly in the program.

Resetting only makes sense for infinite regions. It is a constant-time operation.

The Kit contains an analysis, the *storage mode analysis*, which has two purposes:

1. inserting automatic resetting of infinite regions, when possible;
2. checking applications of `resetRegions` (and `forceResetting`) in order to report on the safety of the resetting requested by the programmer.

As a matter of design, one might wonder whether it would not be sufficient to rely on the user indicating where resetting should be done. However, checking whether resetting is safe at a particular point chosen by the user is of course no easier than checking whether resetting is safe at an arbitrary point in the program, so one might as well let the compiler insert region resetting whenever it can prove that it is safe.

In this chapter we describe the principles that underlie the storage mode analysis. Even if one is willing to insert `resetRegions` and `forceResetting` instructions in the program, one still needs to understand these principles, in order to be able to act upon the messages that are generated by the system in response to explicit `resetRegions` and `forceResetting` instructions.

12.1 Storage Modes

As we have seen in previous chapters, region inference decorates every allocation point with an annotation of the form "`at` ρ", indicating into which region the value should be stored.

Now the basic idea is that storing a value into a region can be done in one of two ways, at runtime. One either stores the value at the *top* of the region, thereby increasing the size of the region; or one stores the value into the *bottom* of the region, by first resetting the region (so that it contains no values) and then storing the value into the region.

The storage mode analysis transforms an allocation point "`at` ρ" into "`attop` ρ" when it estimates that ρ contains live values at the allocation point, whereas it transform it into "`atbot` ρ", if it can prove that the region will not contain live values at that allocation point. The tokens `attop` and `atbot` are called *storage modes*.

Region polymorphism introduces several interesting problems. Let f be a region-polymorphic function with formal region parameter ρ and consider an allocation point `at` ρ in the body of f. Whether it is safe for f to store the value `atbot` in the region depends not only on the body of f but also on the context in which f is called.

For example, consider the compilation unit

```
fun f [] = []
  | f (x::xs) = x+1 :: f xs

val l1 = [1,2,3]
val l2 = if true then f l1 else l1
val x::_ = l1;
```

When `f` stores the empty list, it can potentially reset the region it writes into (as well as the auxiliary region intended for the auxiliary pairs of the list). In the above program, however, the conditional forces `f l1` and `l2` to be in the same regions as `l1`. Since `l1` is live after the application of `f`, this application must not use `atbot` as storage mode. Indeed, even if we removed the last line of the program, the application could still not use `atbot`, for `l1` is exported from the compilation unit and thus potentially used by subsequent compilation units.

By contrast, consider[1]

[1] Project: `kitdemo/sma`, file: `kitdemo/sma1.sml`.

```
fun f [] = []
  | f (x::xs) = x+1 :: f xs

val n = length(let val l1 = [1,2,3]
               in if true then f l1 else l1
               end)
```

When `f` stores `nil`, it is welcome to reset the regions that hold `l1`, for by that time, `l1` is no longer needed! (`f` traverses `l1`, but when it reaches the end of the list, `l1` is no longer needed.) Indeed the Kit will *replace* the list `[1,2,3]` by `[2,3,4]`. The ability to replace data in regions is crucial in many situations (as we illustrated with the game of Life in Section 1.3).

Since the Kit allows separate compilation, it cannot know all the call sites of a region-polymorphic function, when it is declared. Therefore, when considering an allocation point "`at` ρ" inside the body of some region-polymorphic function, f, which has ρ as a formal region parameter, one cannot know at compile time whether to use `attop` or `atbot`. Instead, the storage mode analysis operates with a third kind of storage mode: `sat`, read: "somewhere at". Consider an application of f in which ρ is instantiated to some region variable, ρ', say. At runtime, ρ' is bound to some region name (Section 2.1), r'. Then r' is combined with a definite storage mode (i.e., `attop` or `atbot`), to yield r, say, which is then bound to ρ. When r' was originally created (by a `letregion`-expression), r' was also made to contain an indication of whether it is an infinite region or a finite region.[2] At runtime, an allocation `sat` ρ in the body of f will test r to see whether the region is infinite and whether the value should be stored at the top or at the bottom.[3]

The relevant parts of the result of compiling the last example above are shown in Figure 12.1. To see the storage modes, switch on the flag `print atbot expression` in the menu `Printing of intermediate forms`. The intermediate form obtained by enabling this flag is from before the optimisation that drops **get**-regions (page 56) and may therefore have more region variables.

[2] On machines that have at least four bytes per word, the two least significant bits of a pointer to a word will always be 00. These two bits hold extra information in the region name. One bit, called the "atbot bit", holds the current storage mode of the region. Another bit, called the "infinity bit", indicates whether the region is finite or infinite.

[3] When ρ has multiplicity infinity, r' must be the name of an infinite region, so the runtime check on whether r has its infinity bit set is omitted.

```
fun f attop r1 [r7:INF, r8:INF, r9:0, r10:0] (var180)=
    (case var180
       of nil => nil sat r7
       |  :: => let val v2007 = decon_:: var180;
                    val x = #0 v2007;
                    val xs = #1 v2007;
                    val v2010 =
                        (x + 1attop r2,
                         letregion r15:1
                         in f[sat r7,sat r8,sat r9,sat r10]
                               atbot r15 xs
                         end) attop r8
                in  :: attop r7 v2010
                end) (*case*) ;
val n =
    letregion r17:1, r19:INF, r20:INF
    in length[atbot r19,atbot r20,attop r2] atbot r17
       let val l1 =
            let val v2014 =
                 (1attop r2,
                  let val v2015 =
                       (2attop r2,
                        let val v2016 =
                             (3attop r2, nil atbot r19) attop r20
                        in :: attop r19 v2016
                        end
                       ) attop r20
                  in  :: attop r19 v2015
                  end) attop r20
            in  :: attop r19 v2014
            end
       in  (case true attop r2
              of true =>
                 letregion r24:1
  (*1*)          in f[atbot r19,atbot r20,atbot r19,atbot r20]
                       atbot r24 l1
                 end
               |  false => l1) (*case*)
       end
    end (*r17:1, r19:INF, r20:INF*)
```

Figure 12.1: Storage modes inferred by the storage mode analysis.

12.2 Storage Mode Analysis

For the purpose of the storage mode analysis, actual region parameters to region-polymorphic functions are considered allocation points. Passing a region as an actual argument to a region-polymorphic function involves neither resetting the region nor storing any value in it, but a storage mode has to be determined at that point nonetheless, since it has to be passed into the function together with the region. The storage mode expresses whether, at the call site, there may be any live values in the region *after* the call. For example, in Figure 12.1 the call to `f` at `(*1*)` passes both `r19` and `r20` with storage mode `atbot` since the only value that exists before the call of `f` and is needed after the call of `f` is `length`, which is declared in a different compilation unit and therefore obviously resides neither in `r19` nor in `r20`.

Within every lambda abstraction, the Kit performs a backwards flow analysis which determines, for every allocation point, a set of *locally live variables*, i.e., a set of variables used by the remainder of the computation in the function up to the syntactic end of the function. (This includes variables which appear in function application expressions.) Prior to the computation of locally live variables, a program transformation, called *K-normalisation*, has made sure that every intermediate result which arises during computation becomes bound to a variable. (This happens by introducing extra `let` bindings, when necessary.)[4]

The Kit also computes a set of locally live variables for each allocation point which does not occur inside any function.

We now give an informal explanation of the rules that assign storage modes to allocation points. Let an allocation point

$$\texttt{at}\,\rho \tag{12.1}$$

be given.

CASE A: ρ is a global region. Then `attop` is used. There is a deficiency we have to admit here. The Kit only puts `letregion` around expressions, not around declarations. Thus, if one writes

[4]K-normalisation is transparent to users: although the storage mode analysis and all subsequent phases up to code generation operate on K-normal forms, programs are always simplified to eliminate the extra let-bindings before they are presented to the user.

```
local
  fun f [] = []
    | f (x::xs) = x+1 :: f xs
  val l1 = [1,2,3]
in
  val n = length(if true then f l1 else l1)
end
```

at top level, then `l1` is put into a global region, although this is really unnecessary. As a consequence, `f` would be called with storage mode `attop` and thus `l1` would not be overwritten.

CASE B: The region variable ρ is not a global region and the allocation point (12.1) occurs inside a lambda abstraction, i.e, inside an expression of the form `fn` *pat* `=>` e. Here we regard every expression of the form

$$\texttt{let fun f(x) =}\ e\ \texttt{in}\ e'\ \texttt{end}$$

as an abbreviation for

$$\texttt{let val rec f = fn(x) =>}\ e\ \texttt{in}\ e'\ \texttt{end}$$

Then it makes sense to talk about *the smallest enclosing lambda abstraction (of the allocation point)*.

Now there are the following cases:

B1 *ρ is bound outside the smallest enclosing lambda abstraction (and this lambda abstraction is not the right-hand side of a declaration of a region-polymorphic function which has ρ as formal parameter):* use `attop` (see Figure 12.2);

B2 *ρ is bound by a* `letregion`*-expression inside the smallest enclosing function:* use `atbot` if no locally live variable at the allocation point has ρ free in its region type scheme and place (Section 6.2), and use `attop` otherwise (see Figure 12.3);

B3 (first attempt) *ρ is a formal parameter of a region-polymorphic function whose right-hand side is the smallest enclosing lambda abstraction:* use `sat`, if no locally live variable at the allocation point has ρ free in its region type scheme and place, and use `attop` otherwise (see Figure 12.4).

```
letregion ρ
in ... (fn pat => ...at ρ...)
end

fun f at ρ1[ρ] =
   (fn x => (fn y => ... at ρ ...)at ρ2)at ρ1
```

Figure 12.2: Two typical situations where `at` ρ is turned into `attop` ρ by rule B1.

```
(fn pat => ...
 letregion ρ
 in ...at ρ...l...
 end ...
)
```

Figure 12.3: The situation which is considered in B2. If no locally live variable l has ρ occurring in its type scheme and place, replace `at` ρ by `atbot` ρ, otherwise by `attop` ρ.

```
fun f at ρ0 [ρ, ...] =
  (fn pat => ...at ρ...l...)
```

Figure 12.4: The situation which is considered in B3. If no locally live variable l has in its type scheme and place a region variable which may be aliased with ρ, replace `at` ρ by `sat` ρ, otherwise by `attop` ρ.

The motivation for (B1) is that if ρ is declared non-locally, then we do not attempt to find out whether ρ contains live data (this would require a more sophisticated analysis.) The intuition behind (B2) is as follows. Region inference makes sure that the region-annotated type and place of a variable always contains free in it region variables for all the regions which the value bound to that variable needs when used. The lifetime of the region bound to ρ is given by the `letregion` expression which is in the same function as the allocation point. Thus, if no locally live variable at the allocation point has ρ free in its type scheme or place, then ρ really does not contain any live value at that allocation point.

The intuition behind (B3) is the same as behind (B2), but in this case there is a complication: ρ is only a formal parameter so it may be instantiated to different regions; in particular it may be instantiated to a region variable which *does* occur free in the type scheme and place of a locally live variable at the allocation point. If that happens, rule (B3), as stated, is not sound!

We refer to the phenomenon that two different region variables in the program may denote the same region at runtime as *region aliasing*. In order to determine whether to use `sat` or `attop` in case (B3), the Kit builds a *region flow graph* for the entire compilation unit. (This happens in a phase prior to the storage mode analysis proper.) The nodes of the region flow graph are region variables and arrow effects that appear in the region-annotated compilation unit. Whenever ρ_1 is a formal region parameter of some function declared in the unit and ρ_2 is a corresponding actual region parameter in the same unit, a directed edge from ρ_1 to ρ_2 is created. Similarly for arrow effects: if $\epsilon_1.\varphi_1$ is a bound arrow effect of a region-polymorphic function declared in the compilation unit and $\epsilon_2.\varphi_2$ is a corresponding actual arrow effect then an edge from ϵ_1 to ϵ_2 is inserted into the graph. Also, edges from ϵ_2 to every region and effect variable occurring in φ_2 are inserted. Finally, for every region-polymorphic function f declared in the program and every formal region parameter ρ of f, if f is exported from the compilation unit, then an edge from ρ to the global region of the same runtime type as ρ is inserted into the graph. (This is necessary, in order to cater for applications of f in subsequent compilation modules.) Let G be the graph thus constructed. For every node ρ in the graph, we write $\langle\rho\rangle$ to denote the set of region variables which can be reached from ρ, including ρ itself. The rule that replaces (B3) is:

B3 *ρ is a formal parameter of a region-polymorphic function whose right-*

hand side is the smallest enclosing lambda abstraction: use `sat`, if, for every variable l which is locally live at the allocation point and for every region variable ρ' which occurs free in the region type scheme and place of l, it is the case that $\langle\rho\rangle \cap \langle\rho'\rangle = \emptyset$; use `attop` otherwise.

CASE C: ρ is bound by a `letregion`-expression and the allocation point (12.1) does not occur inside any function abstraction. As in (B2), use `atbot` if no locally live variable at the allocation point has ρ free in its region type scheme and place, and use `attop` otherwise.

12.3 Example: Computing the Length of a List

We shall now illustrate the storage mode rules of Section 12.2 with some small examples which also allow us to discuss benefits and drawbacks associated with region resetting.

Consider the functions declared in Figure 12.5[5]; they implement five different ways of finding the length of a list! The first, `nlength`, is the most straightforward one. It is not tail recursive. Textbooks in functional programming often recommend that functions are written iteratively (i.e., using tail calls) whenever possible. This we have done with `tlength`. Next, `klength` is a version which contains a local region endomorphism `loop` to perform the iteration; `llength` is similar to `klength`, except that the region endomorphism is declared outside `llength`, using `local`. Region and stack profiles resulting from running the program are shown in Figure 12.6. The diagram shows how much space is used in regions, both finite regions on the stack and infinite regions. The `rDesc` band shows how much space is used on the stack for holding region descriptors. The `stack` band shows how much space is used on the stack, including neither finite regions nor region descriptors; the `stack` band mainly consists of registers and return addresses that have been pushed onto the stack.

In Figure 12.6 we clearly see the five phases. In each phase, first a list is built — seen as an almost linear growth in two regions; then follows a shorter computation of the length of the list. The space behaviour of the five ways

[5]Project: `kitdemo/nlength10000`, file: `kitdemo/length.sml`.

```
fun upto n =
  let fun loop(p as (0,acc)) = p
        | loop(n, acc) =
            loop(n-1, n::acc)
  in
      #2(loop(n,[]))
  end

fun nlength [] = 0
  | nlength (_::xs) = 1 + nlength xs

fun tlength'([], acc) = acc
  | tlength'(_::xs, acc) = tlength'(xs,acc+1)

fun tlength(l) = tlength'(l,0)

fun klength l =
  let fun loop(p as ([], acc)) = p
        | loop(_::xs, acc) = loop(xs,acc+1)
  in
      #2(loop(l,0))
  end

local
  fun llength'(p as ([], acc)) = p
    | llength'(_::xs, acc) = llength'(xs,acc+1)
in
  fun llength(l) = #2(llength'(l, 0))
end

fun global(p as ([], acc)) = p
  | global(_::xs, acc) = global(xs, acc+1)

fun glength(l) = #2(global(l, 0))

val run =   nlength(upto 10000) + tlength(upto 10000) +
            klength(upto 10000) + llength(upto 10000) +
            glength(upto 10000);
```

Figure 12.5: Five different ways of computing the length of lists.

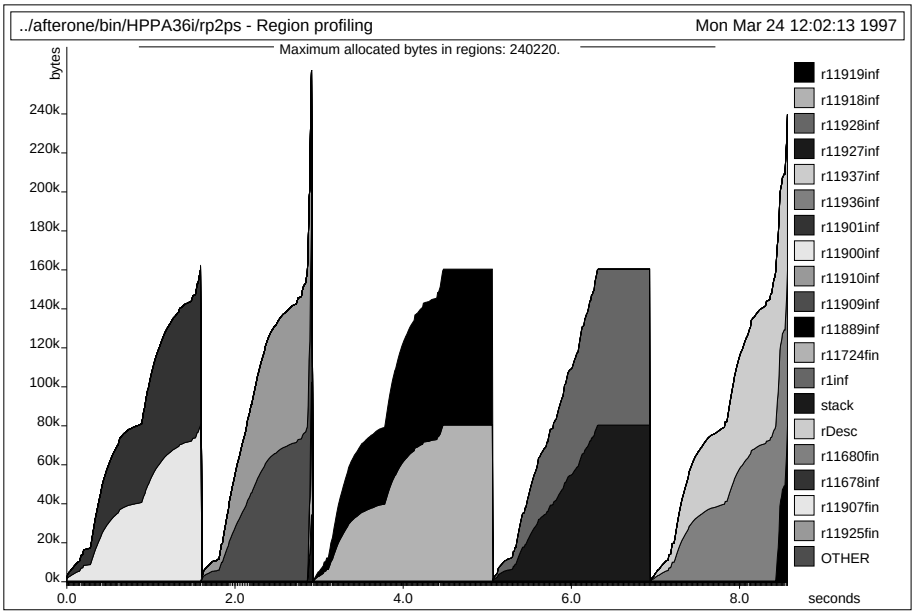


Figure 12.6: Region profiling of five different ways of computing the length of a list, namely, from left to right: `nlength`, `tlength`, `klength`, `llength` and `glength`.

of computing the length vary considerably. We shall have more to say about the time behaviour below.

As one would expect, `nlength` leads to a peak in stack size and it does not use regions. (The peak in stack size is caused by the stacking of a return address.)

Next we see that `tlength` is not an improvement over `nlength`! Note that `tlength'` is region-polymorphic and that the polymorphic recursion in regions allows the pair `(xs, acc+1)` to be stored in a region different from the argument pair to `tlength'`. Thus what appears to be a tail call is in fact not a tail call, for it is automatically enclosed in a `letregion` construct which introduces a fresh region for each argument pair `(xs, acc+1)`. This region is finite, so it is allocated on the stack. That is why we see a sharp increase in stack size for `tlength'`.

The next function, `klength`, deserves careful study, since it is a prototype of a particular schema which can be used again and again when programming with regions. Iteration is done by a region endomorphism, `loop`, which is declared as a local function to the main function. The use of the same variable `p` on both the left-hand side and the right-hand side of the declaration of `loop` forces `loop` to be a region endomorphism. Since the result of `loop(xs,acc+1)` is also the result of `loop`, the result of `loop(xs,acc+1)` therefore has to be in the same region as `p`; but since `loop` is an endomorphism, this forces `(xs, acc+1)` to be in the same region as `p`. Thus what appears to be a tail call (`loop(xs,acc+1)`) really will be a tail call; in particular there will be no fresh region for the argument and no growth of the stack.

Better still, we have carefully arranged that memory consumption will be constant throughout the computation of the length of the list. First, the argument to the initial call of `loop` is a pair `(l, 0)` constructed at that point. Since `loop` is a region endomorphism, the result of `loop(l, 0)` will be in the same region as `(l, 0)`. Moreover, since we then immediately take the second projection of that pair, that region is clearly local to the body of `klength`. Call the region ρ. Since there can be an unbounded number of stores into this region, ρ is classified as infinite by multiplicity inference.

The storage mode passed along with ρ in the initial call `loop(l,0)` is `atbot`, by rule (B2) of Section 12.2. Inside `loop`, the storage mode given to the allocation of `(xs, acc+1)` is `sat`, by rule (B3) in Section 12.2: the only locally live variable at the point where the allocation takes place is `loop` — which we must not destroy before calling! — and the region which `loop` lies

in is clearly different from ρ.

Therefore, every iteration of `loop` resets the "infinite" region ρ so that it will in fact only contain at most one pair. This is seen very clearly in the third hump of Figure 12.6.

Next consider `llength`. The difference from `klength` is that `llength'` is now declared outside `llength`. Note that although the use of `local` makes it clear that `llength'` is not exported from the compilation unit, `llength'` must in fact reside in a global region, since `llength`, which is exported, calls `llength'`. Nonetheless, the storage mode analysis still achieves constant memory usage. As before, we have arranged that iteration is done by a region endomorphism which is initially applied to a freshly constructed pair. This pair can reside in a region which is local to the body of `llength` (once again, the projection `#2(llength'(l, 0))` makes sure that the pair does not escape the body of `llength`). The crucial bit is now which storage mode `llength'` uses when it stores `(xs, acc+1)`. The only locally live variable at that point is `llength'` itself and, as we noted above, `length'` lives in a global region which is clearly different from the region inside `llength` which contains all the pairs. Thus storage mode `sat` will be used, as desired.

Finally, consider `glength`, which is similar to `llength`, but with the crucial difference that `global` is exported from the compilation unit. Since `global` may be called from a different compilation unit, then, for all we know, `global` may be applied to a pair which resides in the same (global) region as `global` itself. Using `sat` when storing `(xs, acc+1)` would then be a big mistake: it would destroy the very function we are trying to call! Therefore, the storage mode analysis assigns `attop` to that storage operation.[6] Consequently we get a memory leak, as shown in the final hump of Figure 12.6.

To sum up, here is how one writes a loop without using space proportional to the number of iterations:

1. The iteration should be done by an auxiliary, uncurried function which is declared as local to the function that uses it; we refer (informally) to this auxiliary function as the *iterator*.

2. The iterator should be a region endomorphism and it should be tail recursive;

[6]To be precise, `attop` comes about by using rule (B3) of Section 12.2. This example illustrates why we put edges from formal region parameters to global regions for exported functions when constructing the region flow graph.

program	`upto`	`nlength`	`tlength`	`klength`	`llength`	`glength`
sec.	0.10	0.14	0.15	0.18	0.20	0.14

Figure 12.7: User time in seconds for building a list of 100,000 elements and computing its length, using five different length functions. `upto` builds the list, but does not compute a length. Times are average over three runs.

3. Iteration should start from a suitably fresh initial argument; the result of the iteration should be kept clearly separate from the region where the iterator function lies.

Mutual recursion poses no additional complications. All functions in a block of mutually recursive functions are put in the same region.

Finally, the reader may be concerned that the two recommended solutions, `klength` and `llength`, seem to be much slower than the other versions. This is mostly an artifact of the profiling software, however.[7] To get a better picture of the actual cost of the different versions, we compiled the five programs separately (using lists of length 100,000 instead of 10,000) using the HP backend and then ran the programs on an HP-9000s700. The results are shown in Figure 12.7. Since `upto` alone takes 0.10 seconds to build the list, the differences in times are clear: the two programs that reset regions (`klength` and `llength`) are slower than those that leak space. Writing `attop` into an infinite region `glength` is only slightly slower than storing values on the stack (`nlength`). Thus most of the extra cost of `klength` and `llength` stems from the operation which resets regions. This extra cost could probably be reduced significantly by an analysis which discovered that the regions that have been marked as infinite only ever contain one value at the time and could therefore be treated as finite regions.

Programming with storage modes is useful if one wants to miniaturise programs using the Kit. However, it is often the case that there are only a few places in the program where resetting is really essential, for example in some main loop which is supposed to run forever. Therefore, the Kit provides two operations which the programmer can use to encourage (or force) the Kit to perform resetting at particular places in the program. These are described

[7] When profiling is turned on, every resetting of a region involves a scan of an entire region page (typically 1 Kb) and this cost far dominates the cost of allocating a pair into the region.

in the next section.

12.4 resetRegions and forceResetting

The programmer can reset regions using the two built-in primitives

`resetRegions` *id*

and

`forceResetting` *id*

Note that, in both cases, the argument has to be an identifier (more specifically, a value variable). To port programs that contain `resetRegions` and `forceResetting` to other ML systems, simply declare

```
fun resetRegions _ = ()
fun forceResetting _ = ()
```

before compiling the program developed using the Kit.

Let ρ be a region variable which occurs free in the type and place of *id*. Let m be the storage mode determined for ρ at a program point according to the rules of the previous section. Whether resetting of *id* at that program point actually takes place at runtime, depends on m and on whether resetting is forced, see Figure 12.8.

12.5 Example: Improved Mergesort

We can now improve on the mergesort algorithm (Section 6.4) by taking storage modes into account. Splitting a list can be done by an iterative region endomorphism which is made local to the sorting function. Also, when the input list has been split, it is no longer needed, so the region it resides in can be reset. Similarly, when the two smaller lists have been sorted (into new regions) the regions of the smaller lists can be reset. These three simple observations lead to the following variant of `msort`:[8]

[8]Project: `kitdemo/msortreset1`, file `kitdemo/msortreset1.sml`.

Does resetting really take place at runtime?

	`resetRegions`	`forceResetting`
$m =$ `atbot`	yes	yes
$m =$ `sat`	only if runtime storage mode is `atbot`	yes$*$
$m =$ `attop`	no$*$	yes$*$

($*$): A compile-time warning is printed in this case.

Figure 12.8: The storage modes that will be used when resetting a region depending on m, the storage mode inferred by the storage mode analysis and depending on whether the resetting is safe (`resetRegions`) or potentially unsafe (`forceResetting`).

```
local
  (* splitting a list *)
  fun split(x::y::zs, l, r) = split(zs, x::l, y::r)
    | split(x::xs, l, r) = (xs, x::l, r)
    | split(p as ([], l, r)) = p


  infix footnote
  fun x footnote y = x

  (* exomorphic merge sort *)
  fun msort []  = []
    | msort [x] = [x]
    | msort xs = let val (_, l, r) = split(xs, [], [])
                 in resetRegions xs;
                    merge(msort l footnote resetRegions l,
                          msort r footnote resetRegions r)
                 end

in
  val runmsort = msort(upto(50000))
```

```
        val result = output(std_out, "Really done\n");
    end
```

Unfortunately, the storage mode analysis complains:

```
resetRegions(v7038):
   You have suggested resetting the regions that appear free
   in the type scheme and place of 'v7038', i.e., in
   (([(int,r2)],[r63],) list,r62)
   (1)
        'r63': there is a conflict with the locally
        live variable
        l :(([(int,r2)],[r72],) list,r71)
        from which the following region variables can be reached
        in the region flow graph:
             {r71,r2,r72}
        Amongst these, 'r72' can also be reached from 'r63'.
        Thus I have given 'r63' storage mode "attop".
   (2)
        'r62': there is a conflict with the locally
        live variable
        l :(([(int,r2)],[r72],) list,r71)
        from which the following region variables can be reached
        in the region flow graph:
             {r71,r2,r72}
        Amongst these, 'r71' can also be reached from 'r62'.
        Thus I have given 'r62' storage mode "attop".
```

Here `v7038` turns out to be `xs` (by inspection of the region-annotated term), so there are two complaints concerning the first `resetRegions`, but none concerning the two remaining ones. Consider `(1)`. By inspecting the region-annotated term one sees that `r62` and `r63` are formal parameters of `msort`. Due to the recursive call `msort r`, the region graph contains an edge from `r63` to `r72` and, as pointed out in `(2)`, an edge from `r62` to `r71`. Thus the analysis decides on `attop`, using rule (B3) This shows a weakness in the analysis, for using `sat` would really be sound. (The problem is that, unlike polymorphic recursion, the region flow graph does not distinguish between different calls of the same function.) Seeing that this is the problem, we

```
local
  (* splitting a list *)
  fun split(x::y::zs, l, r) = split(zs, x::l, y::r)
    | split(x::xs, l, r) = (xs, x::l, r)
    | split(p as ([], l, r)) = p


  infix footnote
  fun x footnote y = x

  (* exomorphic merge sort *)
  fun msort []  = []
    | msort [x] = [x]
    | msort xs = let val (_, l, r) = split(xs, [], [])
                 in forceResetting xs;
                    merge(msort l footnote resetRegions l,
                          msort r footnote resetRegions r)
                 end

in
  val runmsort = msort(upto(50000))

  val result = output(std_out, "Really done\n");
end
```

Figure 12.9: Using `forceResetting` to reset regions.

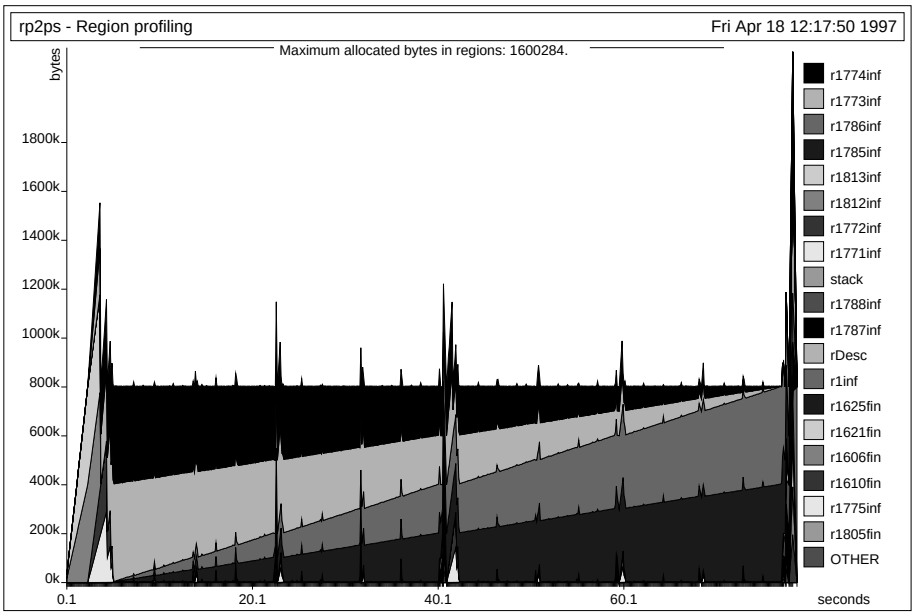


Figure 12.10: Region profiling of the improved mergesort. The two upper trianges contain unsorted elements, while the two lower triangles contain sorted elements. Project: `kitdemo/msortreset2`, file `kitdemo/msortreset2.sml`; program compiled with profiling enabled and then run with the command `run -microsec 100000`. The picture (`region.ps`) was generated by the command: `rp2ps -region -sampleMax 1000 -eps 120 mm` and then previewed using the command `ghostview region.ps` .

decide to put `forceResetting` to work, see Figure 12.9. The region profile of the improved merge sort appears in Figure 12.10. Note that, as expected, we have now brought space consumption down from four times to two times the size of the input. Figure 12.10 may be compared to Figure 6.2, page 61.

12.6 Example: Scanning Text Files

In this section we present a program which can scan a sequence of Standard ML source files in order to compute what percentage of the source files is made up by comments. Recall that an ML comment begins with the two characters `(*`, ends with `*)` and that comments may be nested but must be

balanced (within each file, we require).

The obvious solution to this problem is to implement an automaton with counters to keep track of the level of nesting of parentheses, number of characters read and number of characters within comments. This provides an interesting test for region inference: although designed with the lambda calculus in mind, does the scheme cope with good old-fashioned state computations?

Let us be ambitious and write a program which only ever holds on to one character at a time when it scans a file. In other words, the aim is to use constant space (i.e., space consumption should be independent of the length of the input file).

To this end, let us arrange to use a region with infinite multiplicity to hold the current input character and then reset that region before we proceed to the next character. The iteration is done by tail recursion, using region endomorphisms to ensure constant space usage.

The bulk of the program appears below. The scanning of a single file is done by `scan`, which contains three mutually recursive region endomorphisms (`count`, `after_lparen` and `after_star`) written in accordance with the guidelines in Section 12.3. The built-in `input` function understands storage modes: if called with storage mode `atbot` it will reset the region where the string should be put before reading the string from the input. Consequently, at every call of `next`, the "input buffer region" will be reset.

The other important loop in the program is `driver`, a function which repeatedly reads a function name from a given input stream, opens the file with that name and calls `scan` to process the file. Once again, we want to keep at most one file name in memory at a time, so we would like the region containing the file name to be reset upon each iteration. As it turns out, our `readWord` will always store the string it creates at the top of the region in question. The reason is that it calls `implode`, which is declared in and exported from the prelude. It is `implode` which always stores `attop`, by rule B3 and the fact that formal region parameters of exported functions are connected to global regions in the region flow graph. Thus the two occurrences of resetRegions are necessary. In general, when splitting a program unit into two, one may have to insert explicit `resetRegions` into the second unit, when operations from the first unit are called.

```
local
  exception NotBalanced
```

```
fun scan(is: instream) : int*int =
  let
    fun next() = input(is, 1)
    fun up(level,inside) = if level>0 then inside+1
                           else inside

    (* n: characters read in 'is'
       inside: characters belonging to comments
       level : current number of unmatched (*
       s     : next input character or empty *)
       count is endo *)
    fun count(p as (n,inside,level,s:string))=
      case s of
        "" => (* end of stream: *) p
      | "(" => after_lparen(n+1,inside,level,next())
      | "*" => after_star(n+1,up(level,inside),level,next())
      | ch  => count(n+1,up(level,inside), level,next())
    and after_lparen(p as (n,inside,level,s))=
      case s of
        "" => p
      | "*" => count(n+1,inside+2, level+1,next())
      | "(" => after_lparen(n+1, up(level,inside), level,next())
      | ch => count(n+1,up(level,up(level,inside)),level,next())
    and after_star(p as (n,inside,level,s)) =
      case s of
        "" => p
      | ")" => if level>0 then
                  count(n+1,inside+1,level-1,next())
               else raise NotBalanced
      | "*" => after_star(n+1,up(level,inside), level,next())
      | "(" => after_lparen(n+1,inside,level,next())
      | ch  => count(n+1,up(level,inside),level,next())

    val (n, inside,level,_) = count(0,0,0,next())
  in
   if level=0 then (n,inside) else raise NotBalanced
  end
```

```
fun report_file(filename, n, inside) =
    writeln(implode[filename , ": size = " , toString n ,
             " comments: ", toString inside, " (",
             toString(percent(inside, n)) handle Quot => "",
             "%)"]);

(* scan_file(filename) scans through the file named
   filename returning either Some(size_in_bytes, size_of_comments)
   or, in case of an error, None. In either case a line of
   information is printed. *)

fun scan_file (filename: string) : (int*int)Option=
 let val is  = open_in filename
 in let val (n,inside)  = scan is
    in close_in is;
       report_file(filename, n, inside);
       Some(n,inside)
    end handle NotBalanced =>
        (writeln(filename ^ ": not balanced");
         close_in is;
         None)
 end handle Io msg  => (writeln msg; None)

fun report_totals(n,inside) =
     writeln(implode["\n\nTotal sizes: ", toString n,
          " comments: ", toString inside,
          " (", toString(percent(inside,n)) handle Quot => "",
          ")%"])

(* main(is) reads a sequence of filenames from is,
   one file name pr line (leading spaces are skipped;
   no spaces allowed in file names). Each file is
   scanned using scan_file after which a summary
   report is printed *)

fun main(is: instream):unit =
let
  fun driver(p as(None,n,inside)) =
```

```
            (report_totals(n, inside); p)
      | driver(p as (Some filename,n:int,inside:int)) =
          driver(case scan_file filename of
                   Some(n',inside') =>
                   (resetRegions p;
                    (readWord(is), n+n',inside+inside'))
                 | None => (resetRegions p;
                            (readWord(is),n,inside)))
  in
    driver(readWord(is),0,0);
    ()
  end

in
  val result = main(std_in)
end
```

The program was compiled[9] both with and without profiling turned on. The output from running the program on 10 files listed in the `ML_CONSULT` file of the Kit is shown below

```
Parsing/INFIX_STACK.sml: size = 585 comments: 417 (71%)
Parsing/InfixStack34g.sml: size = 7641 comments: 3120 (40%)
Parsing/Infixing34g.sml: size = 28389 comments: 4645 (16%)
Parsing/LEX_BASICS.sml: size = 2102 comments: 1334 (63%)
Parsing/LEX_UTILS.sml: size = 1294 comments: 399 (30%)
Parsing/LexBasics36e.sml: size = 11674 comments: 2968 (25%)
Parsing/LexUtils33b.sml: size = 7566 comments: 1834 (24%)
Parsing/MyBase.sml: size = 33735 comments: 10896 (32%)
Parsing/PARSE.sml: size = 1151 comments: 644 (55%)
Parsing/Parse34g.sml: size = 6926 comments: 924 (13%)


Total sizes: 101063 comments: 27181 (26)%
```

A region profile for that run is shown in Figure 12.11. The almost-constant space usage is evident. The occasional disturbances are due to the non-

[9]Project: `kitdemo/scan`.

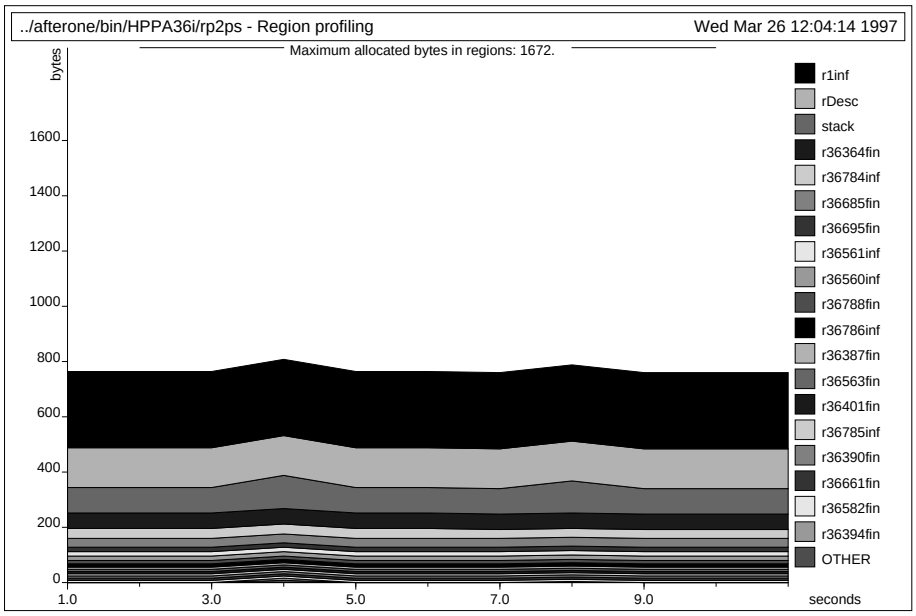


Figure 12.11: Region profiling of the scanner. Note that the unit of measure on the y-axis is bytes, not kilobytes. The occasional increase is due to the functions which read a file name from an instream. Project: `kitdemo/scan`. The program was compiled with profiling enabled, then run by the unix command `run -microsec 100000 < ../../kitdemo/scanfiles`. A postscript picture (`region.ps`) can be generated by the unix command `rp2ps -region -sampleMax 1000 -eps 120 mm`.

iterative functions which read a file name from input by first reading one line and then extracting the name.

Chapter 13

Higher-Order Functions

13.1 Lambda-Abstraction (`fn`)

A *lambda abstraction* in Standard ML is an expression of the form

`fn` *pat* `=>` *exp*

where *pat* is a pattern and *exp* an expression. Lambda expressions denote functions. We refer to the *exp* as the *body* of the function; variable occurrences in *pat* are binding occurrences; informally, the variables that occur in *pat* are said to be *lambda-bound* with scope *exp*.

Lambda-abstractions are represented by closures, both in the language definition and in the Kit. In the Kit, a closure for a lambda abstraction consists of a code pointer plus one word for each free variable of the lambda abstraction. Closures are not tagged.

At this stage, it will hardly come as a surprise to the reader that closures are stored in regions. Sometimes they reside in finite regions on the stack, other times they live in infinite regions, just like all other boxed values.

Every occurrence of `fn` in the program is considered an allocation point; the region-annotated version of the lambda abstraction is

`fn at` ρ *pat* `=>` *exp*

Standard ML allows functions to be declared using `val` rather than `fun`, e.g.,

```
val h = g o f
```

Whereas functions declared with `fun` automatically become region polymorphic, functions declared with `val` do not in general become region-polymorphic.[1] However, in the special case where the right-hand side of the value declaration is a lambda abstraction, the Kit automatically converts the declaration into a `fun` declaration, thereby making the function region polymorphic after all.

ML allows declarations of the form

$$\texttt{fun}\ f\ \ atpat_1\, atpat_2 \cdots atpat_n\ \texttt{=}\ exp$$

as a shorthand for

$$\texttt{fun}\ f\ \ atpat_1\ \texttt{=>}\ \texttt{fn}\ atpat_2\ \texttt{=>}\ \cdots\ \texttt{fn}\ atpat_n\ \texttt{=>}\ exp$$

where *atpat* ranges over atomic patterns. We say that functions declared using this abbreviation are *Curried*.

13.2 Region-Annotated Function Types

The general form of a region-annotated function type is

$$\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2$$

where μ_1 is the region-annotated type and place of the argument and μ_2 is the region-annotated type and place of the result. A region-annotated function type with place takes the form

$$(\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho)$$

where ρ is the region containing the closure. As mentioned in Section 5.2, the unusual looking object $\epsilon.\varphi$ is called an *arrow effect*. Its first component is an effect variable, whose purpose will be explained shortly. The second component is called the *latent effect*, and describes the effect of evaluating the body of the function.

The following example illustrates why latent effects are crucial for knowing the lifetimes of closures.[2] Consider

[1] The reason for this is that the expression on the right-hand side of the value declaration might have an effect (e.g, print something) before returning the function. It would not be correct to suspend this effect by introducing formal region parameters.

[2] Project: `kitdemo/lambda`, file `kitdemo/lambda.sml`.

```
val n = let val f = let val xs = [1,2]
                    in fn ys => length xs + length ys
                    end
        in f [7]
        end
```

Note that `xs` has to be kept alive for as long as the function `fn ys =>` $\cdots$ may be called, for this function will access `xs`, when called. The region-annotated version appears in Figure 13.1. (To see the output programs discussed in this section, enable the flag `print drop regions expression` and look under the heading `Report: AFTER DROP REGIONS`.) We see that `xs` is put in `r11` and `r12`, that the function closure for `(fn ys =>` $\cdots$`)` is put in `r7` and indeed, `r7`, `r11` and `r12` all have the same lifetime. To understand how the region inference system figured that out, let us consider the effect and the region-annotated type of particular sub-expressions. Looking at the lambda abstraction, it must have a functional type of the form $(\mu \xrightarrow{\epsilon.\varphi} \mu', \mathtt{r7})$ where φ is the effect

$$\{\mathbf{get}(\mathtt{r1}), \mathbf{get}(\mathtt{r2}), \mathbf{get}(\mathtt{r11}), \mathbf{get}(\mathtt{r12}), \mathbf{get}(\mathtt{r9}), \mathbf{get}(\mathtt{r10}), \mathbf{put}(\mathtt{r2})\}$$

Note that `r11` and `r12` occur free in the type of the lambda abstraction. But, as pointed out in Section 3.4, the criterion for putting `letregion` ρ `in` $\ldots$ `end` around an expression e is that ρ occurs free neither in the region-annotated type and place of e nor in the region-annotated type scheme and place of any variable in the domain of the type environment. The smallest sub-expression of the program for which `r11` and `r12` does not occur free in the type and place of the expression is the right-hand side of the `val`-binding of `n`, for that expression simply has region-annotated type and place `(int,r2)`. And at that point, the only region variables that occur free in the type environment are global region variables. Hence the placement of the letregion-binding of `r11` and `r12`.

13.3 Arrow Effects

In a first-order language, effect variables might not be particularly important. But in a higher-order language like ML, effect variables are useful for tracking dependencies between functions. The following example illustrates the point:[3]

[3]Project: `kitdemo/lambda`, files `kitdemo/lambda1.sml` and `kitdemo/lambda2.sml`.

```
let val n =
    letregion r7:1, r9:INF, r10:1, r11:INF, r12:INF
    in let val f =
           let val xs =
               let val v9260 =
                   (1,
                    let val v9261 = (2, nil at r11) at r12
                    in  :: at r11 v9261
                    end
                   ) at r12
               in  :: at r11 v9260
               end
           in   fn at r7 ys =>
                letregion r16:1 in length[] xs end +
                letregion r18:1 in length[] ys end
           end
       in  f
           let val v9257 = (7, nil at r9) at r10
           in  :: at r9 v9257
           end
       end
    end
in  {|n: (_,r2)|}
end
```

Figure 13.1: Region-annotated program illustrating that the lifetime of a closure is at least as long as the lifetime of the values that evaluation of the function body will require.

```
fun apply f x = f x
val y = apply (fn n => n+1) 5
val z = apply (fn m => m) 9
```

Here is the region type scheme of `apply`:

$$\forall\alpha_1\alpha_2\rho_1\rho_2\rho_3\rho_4\epsilon_1\epsilon_2\epsilon_3.((\alpha_1,\rho_1)\xrightarrow{\epsilon_1.\emptyset}(\alpha_2,\rho_2),\rho_3)\xrightarrow{\epsilon_2.\{\mathbf{put}(\rho_4)\}}((\alpha_1,\rho_1)\xrightarrow{\epsilon_3.\{\mathbf{get}(\rho_3),\epsilon_1\}}(\alpha_2,\rho_2),\rho_4)$$

The latent effect associated with ϵ_2 shows that when `apply` is applied to a function, it may create (in fact: will create) a function closure in ρ_4. The latent effect associated with ϵ_1 is empty, since the declaration of `apply` does not tell us anything about what effect the formal parameter `f` must have. Crucially, however, ϵ_1 is included as an atomic effect in the latent effect associated with ϵ_3: whenever the body of `apply f` is evaluated, the body of `f` may be (in fact: will be) evaluated.

The polymorphism in effects makes it possible to distinguish between the latent effects of different actual arguments to `apply`. For example, the functions `fn n => n+1` and `fn m => m` have different latent effects. Let us take `fn n => n+1` as example. It has function type and place

$$((\texttt{int},\rho_2)\xrightarrow{\epsilon_5.\{\mathbf{get}(\rho_2),\mathbf{put}(\rho_2)\}}(\texttt{int},\rho_2),\rho_5) \tag{13.1}$$

Here we have assumed that integers always live in ρ_2, while ϵ_5 and ρ_5 were chosen arbitrarily. The region inference algorithm discovers that (13.1) is an instance of $((\alpha_1,\rho_1)\xrightarrow{\epsilon_1.\emptyset}(\alpha_2,\rho_2),\rho_3)$ from the type scheme for `apply` under the instantiating substitution

$$S = (\{\alpha_1\mapsto\texttt{int},\alpha_2\mapsto\texttt{int}\},\{\rho_1\mapsto\rho_2,\rho_2\mapsto\rho_2,\rho_3\mapsto\rho_5\},\\ \{\epsilon_1\mapsto\epsilon_5.\{\mathbf{get}(\rho_2),\mathbf{put}(\rho_2)\}\})$$

Formally, a *substitution* is a triple (S^t,S^r,S^e), where S^t is a map from type variables to types, S^r is a finite map from region variables to region variables and S^e is a map from effect variables to arrow effects. Although we shall not define what it means to apply a substitution to a type in this document, let us explain why substitutions map effect variables to arrow effects. One alternative, one might consider, is to let substitutions map effect variables to effect variables. But then substitutions would not be able to account for the idea that effects can "grow", when instantiated. In the `apply` example, for instance, the empty effect associated with ϵ_1 has to grow to $\{\mathbf{get}(\rho_2),\mathbf{put}(\rho_2)\}$

at the concrete application of `apply` (otherwise, as it is easy to demonstrate, the region inference system would become unsound).

Another alternative would be to let substitutions map effect variables to effects. But that would not work well together with the idea of using substitutions to express "growth" of effects either. For example, applying the map $\{\epsilon \mapsto \{\mathbf{get}(\rho_0), \mathbf{put}(\rho_2)\}\}$ to the effect $\{\mathbf{get}(\rho_9), \epsilon\}$, say, would presumably yield the effect $\{\mathbf{get}(\rho_9), \mathbf{get}(\rho_0), \mathbf{put}(\rho_2)\}$ in which the fact that the original effect had to be at least as large as whatever ϵ stands for, is lost. Instead, we define substitution so that applying the effect substitution $\{\epsilon \mapsto \epsilon.\{\mathbf{get}(\rho_2), \mathbf{put}(\rho)\}\}$ to $\{\mathbf{get}(\rho_9), \epsilon\}$ yields $\{\mathbf{get}(\rho_9), \epsilon, \mathbf{get}(\rho_2), \mathbf{put}(\rho)\}$.

We can now give a complete definition of atomic effects. An *atomic effect* is either an effect variable or a term of the form $\mathbf{get}(\rho)$ or $\mathbf{put}(\rho)$, where ρ as usual ranges over region variables. An *effect* is a finite set of atomic effects.

One can get the Kit to print region-annotated types of all binding occurrences of value variables. Also, one can choose to have arrow effects included in the printout: Enable the flags `print types` and `print effects` in the `Layout` menu. Although this gives very verbose output, it is instructive to look at such a term just once, to see how arrow effects are instantiated. We show the full output for the `apply` example in Figure 13.2. In reading the output it is useful to know that the Kit represents effects and arrow effects as graphs, the nodes of which are region variables, effect variables **put**, **get** or `U` (for "union"; `U` by itself means the empty set). Region variables are leaf nodes. A **put** or **get** node has emanating from it precisely one edge; it leads to the region variable in question. An effect variable node (written `e` followed by a sequence number) is always the handle of an arrow effect; there are edges from the effect variable to the atomic effects of that arrow effect, either directly, or via union nodes or other effect variable nodes. For instance, `e14(U)` in the figure denotes an effect variable with an edge to an empty union node. When a term containing arrow effects is printed, shared nodes that have already been printed are marked with a `@`; their children are not printed again. For instance, in the figure, the second occurence of `r2` is printed as `@r2`. The binding occurrence of `apply` has been printed with its region type scheme. Each non-binding occurrence of `apply` has been printed with four square-bracketed lists: The first list is the actual region arguments; the following three are "instantiation lists" that show the range of the substitution by which the bound variables of the type scheme was instantiated, in the same order as the bound variables occurred. For example, in the second use of `apply`, `r8` was instantiated to `r17`.

```
fun apply
    :all
        'a34,'a32,r7,r8,r9,r10,e11,e12,e13.
        ((('a32,r10)-e11->('a34,r9)),r8)
         -e12(put(r7))->
         ((('a32,r10)-e13(U(U,get(r8),e11))->('a34,r9)),r7)
        )
    at r1
    [r7:1]
    (f)=
    fn e13 at r7 x:('a32,r10) => f x

val y:(int,r2) =
    letregion r9:1, r10:1
    in  letregion r11:1
        in apply
            [r9]
            [int,int]
            [r9,r10,r2,r2]
            [e7(U(get(r2),get(r1),put(@r2))),
             e12(put(r9)),
             e8(e7(U(get(r2),get(r1),put(@r2))),get(r10))
            ]
           at r11
           (fn e7 at r10 n:(int,r2) => n + 1)
        end
        5
    end
val z:(int,r2) =
    letregion r16:1, r17:1
    in  letregion r18:1
        in apply
            [r16]
            [int,int]
            [r16,r17,r2,r2]
            [e14(U),e19(put(r16)),e15(e14(U),get(r17))]
           at r18
           (fn e14 at r17 m:(int,r2) => m)
        end
        9
    end
```

Figure 13.2: The instantiation of arrow effects keeps different applications of the same function (here `apply`) apart. (Project: `kitdemo/lambda`, files: `kitdemo/lambda1.sml` and `lambda2.sml`.)

13.4 Region-Polymorphic Recursion and Higher-Order Functions

Unlike identifiers bound by `fun`, lambda-bound function identifiers are never region-polymorphic. So in an expression of the form

`(fn f =>` $\cdots$ `f` $\cdots$ `f` $\cdots$`)`

all the uses of `f` use the same regions. Indeed, since `f` occurs free in the type environment while region inference analyses the body of the lambda abstraction, none of the regions which appear in the type of `f` will be de-allocated inside the body of the lambda abstraction. Also, such a region must be bound outside the lambda abstraction, so any attempt to reset such a region inside the body of the abstraction will cause the storage mode analysis to complain (by Rule (B1) of Section 12.2).

Therefore, when a function f is passed as argument to another function, g,

$$g(f)$$

first regions are allocated for the use of f, then g is called and finally the regions are de-allocated (provided they are not global regions). Whether the `letregion` construct thus introduced encloses the call site immediately

`letregion` $\rho_1, \ldots, \rho_n$ `in` $g(f)$ `end`

or further out

`letregion` $\rho_1, \ldots, \rho_n$ `in` ... $g(f)$ ... `end`

depends on the type and effect of the expression $g(f)$ in the usual way: regions can be de-allocated when they occur free neither in the type of the expression nor in the type environment.

13.5 Examples: `map` and `foldl`

Consider[4]

[4]Project: `kitdemo/lambda`, files `kitdemo/lambda3.sml` and `kitdemo/lambda4.sml`.

```
fun map f [] = []
  | map f (x::xs) = f(x) :: map f xs

val x = map (fn x => x+1) [7,11]
```

The above formulation of `map` is not the most efficient one in the Kit, since it will create one closure for each element in the list, due to currying.[5] However it serves to illustrate the point made in the previous section about allocating regions in connection with higher-order functions. The region-annotated version is seen in Figure 13.3. We see that the regions that appear free in the type and place of the successor function (i.e., `r2` and `r12`) must be allocated prior to the call of `map` and that they stay alive throughout the evaluation of the body of `map`. Note, however, that the closures that are created when `map` is applied do not pile up in `r12`, the region of the successor function. Instead, they are put in local regions bound to `r22`, one closure in each region. Also, if we had given some more complicated argument to `map`, the body of that function could have `letregion` expressions. For each list element, regions would then be allocated, used and then de-allocated before proceeding to the next list element.

So it might appear that higher-order functions are nothing to worry about when programming with regions. That is not so, however. The limitation that lambda-bound functions are never region-polymorphic can lead to space leaks. Here is an example:

```
fun foldl f acc [] = acc
  | foldl f acc (x::xs)  = foldl f (f(x,acc)) xs

val x = foldl (fn (x,acc) => 10*acc+x) 0 [7,2];
```

Since `f` is lambda-bound, all the pairs created by the expression `(x,acc)` will pile up in the same region. The storage mode analysis will infer storage mode `attop` for the allocation of the pair, by rule (B1) of Section 12.2: since `foldl` is curried, there are several lambdas between the formal region parameter of `foldl` which indicates where the pair should be put, and the allocation point of the pair.

[5] When `map` and the application of `map` appear in the same compilation unit, the Kit will automatically specialise `map` to a recursive function which does not have this defect. (This it the result of a general optimisation of curried, closed functions that have a constant argument.) The output we present in this section was obtained by putting `map` in a compilation unit of its own.

```
fun map at r1 [r7:1, r8:0, r9:0] (var932)=
    fn at r7 var933 =>
    (case var933
     of nil => nil at r8
     |  :: =>
        let val v10434 = decon_:: var933;
            val x = #0 v10434;
            val xs = #1 v10434;
            val v10439 =
                (var932 x,
                 letregion r21:1
                 in  letregion r22:1
                     in map[r21,r8,r9] at r22 var932
                     end
                     xs
                 end
                ) at r9
        in  :: at r8 v10439
        end
    ) (*case*)

val x =
    letregion r9:1, r10:INF, r11:INF, r12:1
    in  letregion r13:1
        in map[r9,r1,r1] at r13 (fn at r12 x => x + 1)
        end
        let val v10465 =
                (7,
                 let val v10466 = (11, nil at r10) at r11
                 in  :: at r10 v10466
                 end
                ) at r11
        in  :: at r10 v10465
        end
    end
```

Figure 13.3: Although this version of `map` creates a closure for each list element, the region-polymorphic recursion (of `map`) ensures that that closure is put in a region local to `map`. Thus these closures do not pile up in `r12`, the region of the initial argument.

It does not help to uncurry `foldl` and turn `foldl` into a region endomorphism:

```
fun foldl(p as (f,[],_)) = p
  | foldl(f,x::xs,acc) = foldl(f,xs,f(x,acc))

val x = #3(foldl(fn(x,acc) => 10*acc+x,[7,2],0));
```

The storage mode analysis will still give `attop` for the allocation of the pair `(x,acc)`, for the region of the pair is free in the type of `f`, which is locally live at that point.

The solution is to require that `f` be curried, to avoid the creation of the pair altogether, i.e, going to higher order rather than lower:

```
fun foldl f b xs =
  let fun loop(p as ([], b))= p
        | loop(x::xs, b) = loop(xs,f x b)
  in
      #2(loop(xs,b))
  end
```

The region-annotated version appears in Figure 14.2 (page 137).

Chapter 14

The function call

Standard ML allows function applications of the form

$$exp_1\, exp_2$$

where exp_1 is the operator and exp_2 is the operand. The syntax for function application is overloaded, in that it is used for three different purposes in ML:

1. application of built-in operations such as `+`, `=`, `:=`;
2. application of a value constructor (including `ref`) or an exception constructor;
3. application of user-defined functions, i.e., functions that are introduced by `fn` or `fun`;

This chapter is about the last kind of function application; in this chapter, we use the term function application to stand for application of user-defined functions only.

Function applications are ubiquitous in Standard ML programs; in particular, iteration is often achieved by function calls. Not surprisingly, careful compilation of function calls is essential for obtaining good performance.

The Kit partitions function calls into four kinds, which are implemented in different ways. At best, a function call is simply realised by a jump in the target code.

The resource conscious programmer will want to know the special cases; for example, when doing an iterative computation, it is important to know

whether the space usage is going to be independent of the number of iterations.

In this section we enumerate the cases recognized by the Kit and show how one can check whether specific function calls in the code turn out the way one intended.

The Kit performs a backwards flow analysis, called *call conversion*, to determine which function calls are tail calls and, more generally, which function calls fall into the special cases listed below. We say that expressions produced by this analysis are *call-explicit*.

One can inspect call-explicit programs by enabling the flag

```
print call-explicit expression
```

in the menu `Printing of intermediate forms`. Call-explicit expressions are produced after regions have been dropped (page 56) but before generation of KAM code.

We shall first give a brief description of the parameter passing mechanism in general and then discuss the different kinds of function calls provided, working our way from the most specialised (and most efficient) cases towards the default cases.

14.1 Parameter Passing

There is one (and so far only one) register which is used for passing arguments to functions. It is called `stardardArg`. In addition, region-polymorphic functions use another fixed register, called `standardArg1`[1], which points to the record of region parameters which the caller has allocated prior to the call.

14.2 Tail Calls

A call which is the last action of a function is referred to as a *tail call*. After region inference, the Kit performs a tail call analysis (in one backwards scan through the program). It is significant that the tail call analysis happens after region inference: as we saw in Section 12.3, a function call that looks

[1] Admittedly, not terribly good nomenclature.

like a tail call in the source program may end up as a non-tail call in the region-annotated program, because the function has to return in order to free memory.

14.3 Simple Jump (jmp)

In this section we shall consider conditions under which one can implement a function call as a simple jump. A call of a region-polymorphic function takes the form f `[`ρ_1`, ..., `ρ_n`]` `at` ρ_0 *exp* where ρ_0 is the region which holds the region vector containing the actual region parameters ρ_1, ..., ρ_n. During K-normalisation, the Kit tries to bring the creation of ρ_0 close to the point of the call. Therefore, an important case to consider is a call of the form

$$\texttt{letregion}\ \rho_0\ \texttt{in}\ f\ \texttt{[}\rho_1\texttt{, ..., }\rho_n\texttt{]}\ \texttt{at}\ \rho_0\ \mathit{exp}\ \texttt{end}\quad (n \geq 0) \tag{14.1}$$

where f is the name of a region-polymorphic function.

The Kit simplifies this expression to a simple jump

$$\texttt{jmp}\ f\ \mathit{exp}$$

if the following conditions are met:

1. the call is a tail call; and
2. one has
   (a) $n = 0$; or
   (b) the call occurs inside the body of some region-polymorphic function g and
      i. the actual region parameters ρ_1, ..., ρ_n are a prefix of the formal region parameters of g, i.e., the list of formal region parameters of g is `[`ρ_1, ..., ρ_n, ρ_{n+1}, ..., ρ_{n+k}`]`, for some ρ_{n+1}, ..., ρ_{n+k}; and
      ii. the closest surrounding λ of the call is the λ that starts the right-hand side of g.

The start address of f is known during compilation (since f is region polymorphic). Thus such a function call is as efficient as an assembly language `goto` to a constant label.

To understand the above requirements, note that if the region ρ_0 really has to be created (be it on the stack or as an infinite region) then the call f cannot be treated as a tail call, for f has to return to de-allocate ρ_0. Now (2a) is one way of ensuring that there is no need to allocate ρ_0. A different way is given by (2b). The idea is to re-use the region vector of the function g in which the call of f occurs (a common special case is that g is f). Condition (2(b)i) ensures that the actual region parameters of f coincide with (a prefix of) the formal parameters of g. Finally, (2(b)ii) is necessary in order to ensure that the region vector of g really is available when f is called.

To understand (2(b)ii) in more detail, consider the example

```
fun g[r](x) =
  h[r1] (fn y => letregion r2 in f[r] at r2 y end),
```

which one might think of as sugar for

```
val rec g[r] = fn x =>
  h[r1] (fn y => letregion r2 in f[r]at r2 y end).
```

Here the call to `f` will not be implemented by a `jmp`, for there is a `fn` between the start of the body of `g` and the call of `f`. Indeed, we must not implement the call of `f` by a `jmp`, for in the call `f[r] at r2`, a region vector containing `r` has to be constructed, since, at the point of the call, `r`, is available only from the closure of `fn y => letregion r2 in f[r]at r2 y end`.

Note that (14.1) requires that the `letregion` bind only one region variable (the region used for the region record). The way to avoid that `letregion` binds more than one region variable is to turn the calling function into a region endomorphism, when possible.

The following is an example of how one obtains simple jumps:[2]

```
local
  fun f'(p as (0,b)) = p
    | f'(n,b) = f'(n-1,n*b)
in
  fun f(a,b) = #2(f'(a,b))
end;
```

The call-explicit version of `f'` appears in Figure 14.1. Another example of a `jmp` tail call will be shown in Section 14.8.

[2]Project: `kitdemo/tail`, file: `kitdemo/tail2.sml`.

```
fun f' attop r1 [r7:inf] (var1024)=
    (case #0 var1024
     of 0 => var1024
     |  _ =>
       let val n = #0 var1024; val b = #1 var1024
       in  jmp f' (n - 1, n * b) sat r7
       end
    ) (*case*) ;
```

Figure 14.1: An example where a function call turns into a simple jump.

14.4 Non-Tail Call of Region-Polymorphic Function (`funcall`)

Still referring to the form (14.1), let us consider the case where (1) or (2) is not satisfied. Then the Kit will allocate ρ_0 before the call of f and de-allocate it afterwards.[3] The region bound to ρ_0 will always be finite and be on the stack. Due to this allocation, the call cannot be a tail call. The mnemonic used for a non-tail call of a region-polymorphic function is `funcall`. Thus (14.1) is simplified to

`letregion` ρ_0 `in funcall` f `[`ρ_1`, ..., `ρ_n`] at` ρ_0 exp `end`.

Now let us turn to calls of region-polymorphic functions which do not fit the pattern (14.1). One special case is:

`letregion` $\rho_0, \rho_1, \ldots \rho_k$ `in` f `[] at` ρ_0 exp `end`

where $k > 0$. Here ρ_0 is not needed; the Kit therefore replaces the expression by

`letregion` $\rho_1, \ldots \rho_k$ `in funcall` f exp `end`

(For reasons of presentation, we have assumed that the `letregion`-bound region variables have been rearranged, if necessary, to bring ρ_0 to the front.)

[3] One could avoid this allocation in the case $n = 1$ or, more generally, if one allowed unboxed representation of region vectors, but for simplicity, we choose to forego this opportunity for optimisation.

Every remaining case of an application of a region-polymorphic function

$$f \texttt{ [}\rho_1\texttt{, ..., }\rho_n\texttt{] at } \rho_0 \ exp$$

is replaced by

$$(\texttt{funcall } f \texttt{ [}\rho_1\texttt{, ..., }\rho_n\texttt{] at } \rho_0) \ exp$$

This completes all possible cases of applications of region-polymorphic functions. We now turn to function applications where the operator is not the name of a region-polymorphic function.

14.5 Tail Call of Unknown Function (`fnjmp`)

Consider the case:

$$exp_1 \ exp_2$$

where (a) the call is a tail call and (b) exp_1 is not the name of a region-polymorphic function.

Here exp_1 will be evaluated to a closure, pointed to by a standard register, `standardClos`. Then exp_2 will be evaluated and the result put in the standard register `standardArg`. The first word in the closure always contains the address of the code of the function. This address is fetched into a register and a jump to the address is made. Since the call is a tail call, it induces no allocation, neither on the stack nor in regions. It is thus as efficient as an indirect `goto` in assembly language.

The mnemonic used in call-explicit expressions for this special case is

$$\texttt{fnjmp } exp_1 \ exp_2$$

14.6 Non-Tail call of Unknown Function (`fncall`)

Consider the case

$$exp_1 \ exp_2$$

where (a) the call is not a tail call and (b) exp_1 is not the name of a region-polymorphic function.

This is implemented as follows: first exp_1 is evaluated and the result, a pointer to a closure, is stored in `standardClos`. Then exp_2 is evaluated and

stored in `standardArg`. Then live registers and a return address are pushed onto the stack and a jump is made to the code address which is stored in the first word of the closure pointed to by `standardClos`. Upon return, registers are restored from the stack.

The mnemonic used in call-explicit expressions for this special case is

$$\texttt{fncall}\ \ exp_1\ \ exp_2$$

14.7 Example: Function Composition

The prelude defines function composition as follows:

```
fun (f o g) x = f(g x)
```

The resulting call-explicit expression produced by the Kit is[4]

```
fun o attop r1[r7:2] (var1026) =
    fn attop r7 x =>
       let val f = #0 var1026; val g = #1 var1026
       in  fnjmp f (fncall g x)
       end
```

Note that `f o g` first creates a closure in `r7` and then returns. When called, the created function first performs a non-tail call of `g` and then a tail call to `f`.

14.8 Example: `foldl` Revisited

Consider

```
fun foldl f b xs =
  case xs of
    [] => b
  | x::xs' => foldl f (f x b) xs'
```

Note that the recursive call of `foldl` is a call of a known function, but not a tail call: `foldl` returns a closure, which is subsequently applied to the value of `(f x b)`. This too returns a closure which in turn is applied to `xs'`. The resulting call-explicit expression is[5]

[4] Project `kitdemo/compose`, file `kitdemo/compose.sml`.

[5] Project `kitdemo/fold`, file `kitdemo/fold1.sml`.

```
fun foldl attop r1 [r7:4, r8:4] (f)=
    fn attop r7 b =>
       fn attop r8 xs =>
          (case xs
           of nil => b
           |  :: =>
             let val v11876 = decon_:: xs;
                 val x = #0 v11876;
                 val xs' = #1 v11876
             in  letregion r22:4
                 in fncall
                    letregion r24:4
                    in fncall
                       letregion r25:2
                       in funcall foldl[atbot r24,atbot r22] atbot r25 f
                       end
                       (fncall (fncall f x) b)
                    end
                    xs'
                 end
             end
          ) (*case*)
```

Note that upon each iteration, fresh regions for holding two closures are being allocated for the duration of the recursive call. Thus space usage is linear in the length of the list (4 words for each list cell, to be precise).

An efficient version of `foldl` is written thus:

```
fun foldl f b xs =
  let fun loop(p as ([], b))= p
        | loop(x::xs, b) = loop(xs,f x b))
  in
      #2(loop(xs,b))
  end
```

It is compiled into the call-explicit expression in Figure 14.2.[6] There are two reasons why this is much better: the loop is implemented as a jump and,

[6]Project `kitdemo/fold`, file `kitdemo/fold2.sml`.

```
fun foldl attop r1 [r7:3, r8:3] (f)=
    fn attop r7 b =>
       fn attop r8 xs =>
          letregion r20:1
          in let fun loop atbot r20 [r21:inf] (var1074)=
                     (case #0 var1074
                      of nil => var1074
                      |  :: =>
                         let val v11919 = #0 var1074;
                             val v11921 = decon_:: v11919;
                             val b = #1 var1074;
                             val x = #0 v11921;
                             val xs = #1 v11921
  (* note jmp *)          in  jmp loop (xs,
                                         fncall (fncall f x) b
                                        ) sat r21
                         end
                     ) (*case*)
             in  letregion r28:inf
                 in let val v11926 =
                          letregion r29:1
                          in funcall loop[atbot r28] atbot r29
                                     (xs, b) atbot r28
                          end
                    in  #1 v11926
                    end
                 end
             end
          end
```

Figure 14.2: The result of compiling `foldl` is an iterative function which avoids argument pairs piling up in one region.

more importantly, there is no new allocation in each iteration, except, of course, for the allocation which `f` might make.[7]

As an exercise, consider the following variant of `foldl` which assumes that `f` takes a pair as an argument:

```
fun foldl' f b xs =
  let fun loop(p as ([], b))= p
        | loop(x::xs, b) = loop(xs,f(x,b)))
  in
      #2(loop(xs,b))
  end
```

Interestingly, this program contains a potential space leak. Can you detect it? If not, the Kit will tell you when you compile the program.[8]

[7] We repeat that because `f` is a formal function parameter, all the allocations made by the calls to `f` (one call for each element of the list) are put in the same regions. If the list is very long or the values produced large, it may be a good idea to copy the final result to separate regions.

[8] Project `kitdemo/fold`, file `kitdemo/fold3.sml`.

Part III

System Reference

Chapter 15

Using the Profiler

We have already seen several examples of the use of the region profiler. We shall now explain how to profile in more detail. For example, we shall see how one can find out precisely what allocation points in the program contribute to a particular region.

The region profiler consists of several tools which can be used to analyse the dynamic memory behaviour of the target program. First of all, there are the *profiles* which are graphs showing the dynamic memory usage of the executed program. There are three different graphs:

- A *region profile* is a graph which gives a "global" view of the memory usage by showing the total number of words allocated in regions and on the stack as a function of time. In the graph, regions that arise from the same

  `letregion` ρ `in` e `end`

  expression are collected into one coloured band, labelled ρ. The region variables that label bands are always global or letregion-bound, never formal region parameters.

- An *object profile* is a graph which gives a "local" view into a particular region, as a function of time. The graph shows the objects allocated into a chosen region, with one coloured band for each allocation point in the region-annotated lambda program[1]. Each allocation point is

[1] Every occurrence of an "`at`" in the region-annotated lambda program is an allocation point.

annotated with a *program point* which is a unique number identifying the allocation.[2]

If you have an object profile showing that program point 42 (written `pp42`) contributes with a lot of allocations you can search for `pp42` in the region-annotated lambda program.

- A *stack profile* is a graph which shows the stack memory usage, as a function of time.

As described above the region profiler can give you a region-annotated lambda program annotated with program points.

During compilation, it is also possible to generate a *region flow graph* which shows how regions may be passed around at runtime when region polymorphic functions are applied. The region flow graph is very handy when profiling larger programs when one wants to find out why a formal region variable has been instantiated to a certain `letregion`-bound region variable.

An example should clarify this. Suppose the region profile shows that `r5` grows very big. Further, suppose an object profile of `r5` shows that program point `pp345` is responsible. Searching for `pp345` in the region-annotated program, you may find that the allocation at `pp345` is into some other region variable, `r34`, say. Here `r34` will be a formal region parameter of a region-polymorphic function which at runtime has been instantiated to `r5` by one or more calls of region-polymorphic functions.

You can now use the region flow graph to find the "cascade" of region polymorphic applications that ends up instantiating `r34` to `r5`.

Profiling is sketched in Figure 15.1.

We will now show an example on how to profile a concrete program containing a space leak and then show how the profiler can be used to fix it. After that, we explain in more detail how to specify the profiling strategies and how the profiles are generated.

[2]Program points are unique local to a project, e.g. with a project containing two source files, the program points in the region-annotated lambda programs for the two files will be distinct.

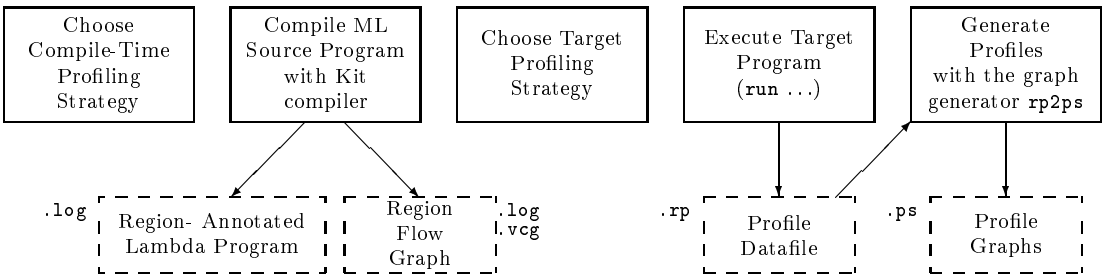


Figure 15.1: Overview of the ML Kit profiler. Dotted boxes represent output from the profiler. The file containing the output is also shown, e.g. a profile goes into a `.ps` file.

15.1 Profiling project `scan_rev1`

In this section, we concentrate on the general principles of profiling. We use the revised scan project (project `/kitdemo/scan_rev1`) as an example. Instead of asking for an input file to scan (as project `scan` does) the program scans the same file (`../../kitdemo/life.sml`) 11 times.

The first thing to do is to get an overview of the memory usage of the program. The region profile does that, see Figure 15.2

The graph shows that region `r1` holds the largest amount of memory, but it does not get bigger over time. Region `r2797`, however, accumulates more memory for each time it scans the life program.

To see what happens in region `r2797`, we make an object profile of that region, see Figure 15.3.

The object profile shows that program point `pp33` produces a lot of allocations which are first freed again when the program stops. We now search for `pp33` in file `prelude.log` and find:

```
fun implode attop r1 pp32 [r110:inf] (strs)=
    ccall(implodeStringProfiling,  attop r110 pp33, strs);
```

Formal region variable `r110` is instantiated with `letregion`-bound region variable `r2797` in a call to function `implode`. We now search after `r2797` in file `scan_rev1.log` and find the following fragment of the region flow graph.

```
        toString    [r1545:inf]
```

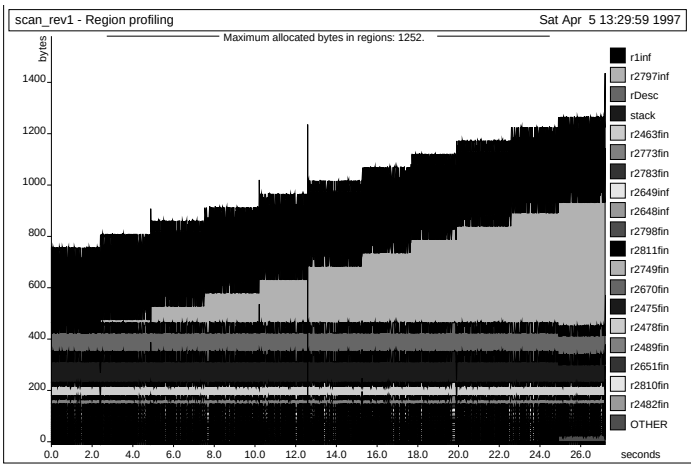


Figure 15.2: It is obvious that memory is accumulated in the two top bands. The global region `r1` and region `r2797` hold the largets amount of memory. The graph has been generated by executing `run -microsec 10000` on the HPUX (with the C backend) and then typing `rp2ps -sampleMax 1200 -region`.

```
--r1545 attop--> LETREGION[r2395:inf];
 --r1545 attop--> driver[r2468:inf]
--r2468 attop--> LETREGION[r2797:inf];   (15.1)
```

This is read as follows: the formal region variable `r2468` is instantiated to `letregion`-bound region variable `r2797` when function `driver` is called. Formal region variable `r1545` is then instantiated to region variable `r2468` when calling function `toString`.[3] Searching after `r1545` in file `lib.log` shows that `toString` calls function `implode` which is found in file `prelude.log`.[4]

```
fun toString attop r1 pp200 [r1545:inf] (n)=
    letregion r1542:inf, r1543:inf, r1544:inf, r1547:1
    in implode[sat r1545 pp208] atbot r1547 pp207
```

[3]Region flow graphs are local to each program in a project. Calling a non local region polymorphic function will then introduce an edge in the region flow graph, but we do not know in which module the called function is located. It may be necessary to look in several log files to find the path from a formal region variable to an actual.

[4]We use ... to indicate that we have deleted text.

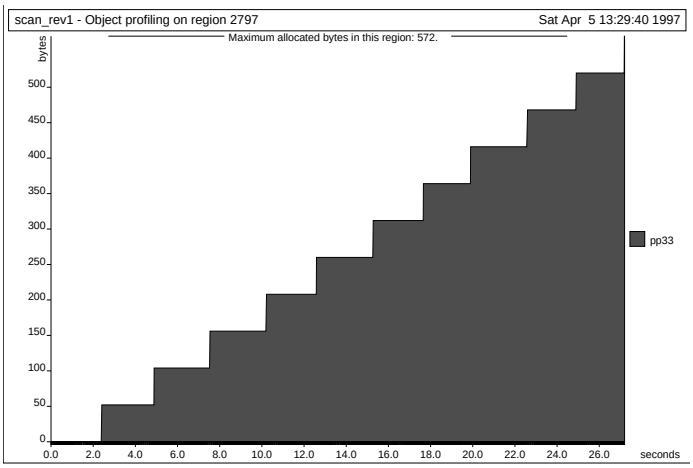


Figure 15.3: There seems to be a space leak at program point `pp33`. The graph has been generated by typing `rp2ps -sampleMax 1200 -object 2797`.

...

We see that region `r2797` is passed with storage mode attop (15.1, above) to formal region variable `r2468` when function `driver` is called for the first time. Region `r2797` contains the result string which is printed after scanning the file. This can be seen from the lambda program in file `scan_rev1.log`. Searching after allocation points allocating into region `r2468` gives among others the following fragments: `": size = "attop r2468` and `" comments: "attop r2468`.

The result string is not needed after a file has been scanned and the result string printed, so the memory holding the result string can be de-allocated. The `attop` storage mode explains why the region holding the result string is not deallocated between scans. So, why is the storage mode attop? To answer this we have to see where `driver` is called the first time, which is in function `do_it`:

```
fun main(is: instream):unit =
let
  fun driver(None,n,inside) =
        report_totals(n, inside)
```

```
    | driver(Some filename,n:int,inside:int) =
        case scan_file filename of
          Some(n,inside) => report_totals(n,inside)
        | None => ()

  val filename = "../../kitdemo/life.sml"
  fun do_it 0 = driver(Some filename, 0, 0)
    | do_it n = (driver(Some filename, 0, 0); do_it (n-1))
in
  do_it 10;
  ()
end
```

The following fragment of the corresponding lambda program (file `scan_rev1.log`) shows that the file name `"life.sml"` is also put into region `r2797` which, of course, has to stay allocated between the scans:

```
...
letregion r2797:inf
in let val filename = "../../kitdemo/life.sml"attop r2797 pp501
   in  letregion r2798:3
     in let fun do_it atbot r2798 pp502 [] (var70)=
...
```

The reason that the file name and the other strings mentioned above must stay in the same region is that they are all made part of the same list of strings, namely the argument to `implode` in `report_totals`.

The region `r2797` contains both local and non-local data to the driver function which is why the region cannot be reset in the driver function. A general solution to this problem is to delay the creation of the file name, so that the file name is created at each call to `driver`. The newly created file name will then be put into a region local to the application point to `driver`. The revision is found in project `kitdemo/scan_rev2`:

```
fun filename() = "../../kitdemo/life.sml"
fun do_it 0 = driver(Some (filename()), 0, 0)
  | do_it n = (driver(Some (filename()), 0, 0); do_it (n-1))
```

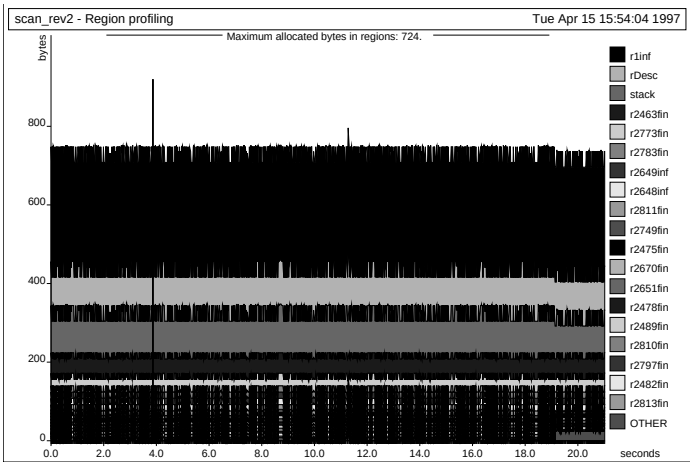


Figure 15.4: There is no space leak: no matter how many times we scan the file, the project will use the same number of words. The graph has been generated by executing `run -microsec 10000` and `rp2ps -sampleMax 1200 -region`.

Figure 15.4 shows a region profile of the `scan_rev2` project.

To see the effect of the modifications above consider the following lambda fragment (found in file `scan_rev2.log`):

```
fun do_it atbot r2797 pp501 [] (var142)=
  (case var142
     of 0 =>
        letregion r2799:inf, r2800:2, r2801:3, r2803:1
        in driver[atbot r2799 pp506] atbot r2803 pp505
           (Some atbot r2800 pp503 "../../kitdemo/life.sml"
                                       attop r2799 pp502,
            0,
            0
           ) atbot r2801 pp504
        end (*r2799:inf, r2800:2, r2801:3, r2803:1*)
```

The copy of the file name is put into region `r2799` which is deallocated after the call to `driver`.

15.2 Compile-Time Profiling Strategy

We will now show some examples on how the profiling tools can be used. As shown in Figure 15.1 we have to choose a *Compile-Time Profiling Strategy*. The Compile-Time Profiling Strategy directs how the region-annotated lambda code with program points and the region flow graph are generated.

The Compile-Time Profiling Strategy is set up in the Profiling sub-menu in the ML Kit menu.[5]

```
Profiling

0       region profiling.............................. off >>>
1       generate lambda program with program points. off
2       generate region flow graph (.vcg file)...... off
3       paths between two nodes in region flow graph [] >>>
4       Instruction Count Profiling................. off
```

Region profiling is enabled by choosing the first item: `region profiling`.

If you want the region-annotated lambda code with program points, toggle the second menu item: `generate lambda program with program points`. The lambda program is written on the log.

To generate a region flow graph, choose `generate region flow graph (.vcg file)`. The region flow graph will be written on the log in text layout which may be hard to read. A more readable graph is exported to the target directory in file *f*`.vcg` where *f*`.sml` is the source program. The file *f*`.vcg` contains the region flow graph in a format that can be read by the VCG tool (Visualization of Compiler Graphs[6]).

As a running example we use the project `life` in the `kitdemo` directory. The project contains one file: `life.sml`. We toggle the first three options on (above, 0-2) and compile the project from inside the `Project` sub-menu.

[5] The `Instruction Count Profiling` option is only available in the HP-UX backend and has nothing to do with region profiling. It simply counts the number of executed instructions in the target program excluding runtime calls and the link file. It should only be used when region profiling is not enabled. If the number of instructions executed gets too large, the Overflow exception is raised.

[6] The VCG tool can be obtained from

`http://www.cs.uni-sb.de/RW/users/sander/html/gsvcg1.html`.

We use version 1.30 found in file `vcg.1.30.r3.17.tar`.

The ML Kit now generates several files of which we have `life.log` (containing, among other things, the lambda program with program points and the region flow graph in text layout), `life.vcg` (the region flow graph ready to use with the VCG tool) and the executable `run`.

Lambda program with program points

In the log file (`life.log`) you find the lambda program by searching for `LAMBDA CODE WITH PROGRAM POINTS`:

```
Report: LAMBDA CODE WITH PROGRAM POINTS::
   let exception Div : (exn,r1)
                  (* exn value or name attop r1 pp2 *);
...
```

In the first line we have an allocation with storage mode `attop` into region `r1`. The allocation point has program point 2 (`pp2`).

Region flow graphs

The region flow graph is found by searching after `REGION FLOW GRAPH FOR PROFILING`:

```
Report: REGION FLOW GRAPH FOR PROFILING::
   Begin layout of region flow graph and SCC-graph.
...
       cp_list[r314:inf]
          --r314 sat-->   [*r314*] ;
          --r314 atbot-->   LETREGION[r1539:inf];
          --r314 sat-->   nthgen'[r944:inf]
                        --r944 sat-->   [*r944*] ;
                        --r944 atbot-->   LETREGION[r1588:inf];
...
```

The region flow graph is almost equivalent to the graph used by the storage mode analysis (p. 100) where region variables are nodes and an edge between two nodes ρ and ρ' is inserted if ρ is a formal region parameter of a function f which is applied to actual region parameter ρ'. This implies that `letregion`-bound region variables are always leaf nodes.

Nodes in the graph are written in square brackets, where for example `cp_list[r314:inf]` means that `r314` is a formal region parameter in function `cp_list`. An asterisk inside a square bracket means that the node has been written earlier. Only the node identifier (i.e. the region variable) will then be printed. The size of the region is printed after the region variable: we use `inf` for an infinite region and :*size* for a finite region where *size* is the region size in words.

Edges are written with the *from node* identifier inside the edge. The edge points to the *to node*. The text `cp_list[r314:inf] --r314 sat--> [*r314*] ;` is read: there is an edge from node `r314` to node `r314`, and node `r314` has been written earlier. We have a cycle, so `cp_list` must call itself recursively; if you look in file `life.sml` you will find something like:

```
fun cp_list[] = []
  | cp_list((x,y)::rest) =
        let val l = cp_list rest
        in (x,y):: l
        end.
```

It is important to look inside the edge for the from node. Consider for example:

```
...
 LETREGION[r3621:2];   --r3480 atbot-->   LETREGION[r3627:2];
...
```

We do not have an edge from the `letregion`-bound region variable (`r3621`) to the other `letregion`-bound variable (`r3627`).

The strongly connected components graph

The region flow graph can get very complicated to read because we may have mutually recursive functions giving a bunch of edges and cycles. If the graphs get too complicated you may find help in the *strongly connected component* (scc) version of the graph.

The scc graph is found by searching for `[sccNo` in the log file. Each scc is identified by a unique *scc number*. The region variables contained in each scc is written as info on the scc-node.

Consider for example:

```
[sccNo 206: r1181,]   --sccNo 206-->   [sccNo 205: r1486,];
```

We have a scc node (id 206) containing region variable `r1181` and an edge to scc node (id 205) containing region variable `r1486`.

Region flow paths

If you are interested in the possible paths from one region variable to another, the ML Kit can find them for you.

This often happens when you have an object profile (for example of region ρ_1) showing that a certain allocation point is responsible for the allocations of interest but the region they are allocated in is not the same as the one written at the allocation point, say ρ_2, in the region-annotated lambda program.

The region written at the allocation point (ρ_2) must then be a formal region variable and it is now interesting to find out how ρ_2 has been instantiated to ρ_1.

You can specify the from and to nodes that you want the paths for in the fourth menu item (`paths between two nodes in region flow graph`) in the Profiling sub-menu:

```
Profiling

0       region profiling.......................... off >>>
1       generate lambda program with program points. off
2       generate region flow graph (.vcg file)..... off
3       paths between two nodes in region flow graph [] >>>
4       Instruction Count Profiling................ off

Toggle line (t <number>), Activate line (a <number>), Up (u), or Quit(q):

>3
<type an int pair list of region variables,
e.g. [(formal reg. var. at pp.,\texttt{letregion}-bound reg. var.)]> or up (u): >
[(314,1588)]
```

You may type in a list of integer pairs, i.e. you can specify several pairs of nodes that you want the paths for.

Compiling the source program again gives a new log file where you can search for `[Starting layout of paths...`:[7]

[7] Because region variables may change when re-compiling a source program in a project

```
[Starting layout of paths...
    [Start path: [sccNo 59: r314,]--->
                 [sccNo 58: r944,]--->[sccNo 57: r1588,]]
...Finishing layout of paths]
```

If you look at the region flow graph on page 149 you see that the only path from region `r314` to region `r1588` goes through function `nthgen'`, i.e. `nthgen'` calls `cp_list`. If you look in the file `life.sml` you may notice that `nthgen'` actually calls a function `copy` and not `cp_list`. The function `copy` is declared as

```
copy (GEN l) = GEN(cp_list l)
```

If you see in the lambda program (file `life.log`) you may notice that `cp_list` has been inlined instead of `copy` by the optimizer.

Using the VCG tool

The VCG tool can be used to visualize the exported graphs (file *source*`.vcg`). We assume that you have installed the tool and it is started by typing `xvcg` at the command prompt. We use file `life.vcg` as the running example. Typing `xvcg life.vcg` at the command prompt gives the window shown in Figure 15.5.

The two graphs are exported *folded*. To unfold a graph choose **Unfold Subgraph** from the pull-down menu inside the `xvcg` window. The pull-down menu is activated by pressing one of the mouse buttons. After activating **Unfold Subgraph** you have to pick the node representing the graph to unfold. This is done by clicking on the node with the left mouse button. Pressing the right mouse button will then unfold the chosen graph. Figure 15.6 shows a small fraction of the unfolded region flow graph.

The graph is read in the same way as the text-based version in the log file. It can be printed out, scaled etc. from the pull-down menu. The graph is folded again by choosing **Fold Subgraph** and clicking on one of the nodes. All nodes in the graph then turn black, and clicking on the right mouse button folds the graph.

---

it may be necessary to start all over by starting the Kit again and compile the whole project again to make sure that the regions you have specified will match the regions in a region flow graph of a previous compilation.

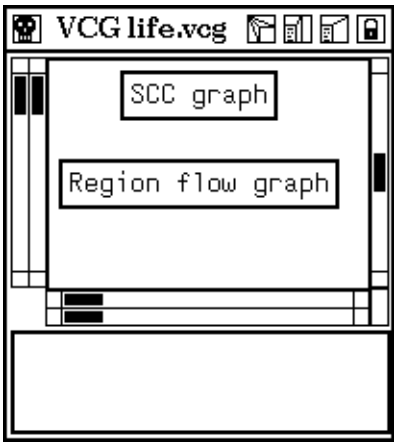


Figure 15.5: The VCG graph contains two nodes. The node "Region flow graph" represents the folded region flow graph and the node "SCC graph" represents the folded strongly connected componemt graph.

Region flow paths are also exported together with the region flow graph. Each path is numbered, and can be viewed by the **Expose/Hide edges** facility in the VCG pull-down menu, see Figure 15.7. Each path is numbered because there can be several paths between the same two nodes. Clicking on the edge class "Graph" will hide the edges in the region flow graph so that edges in the generated path are the only edges shown, see Figure 15.8.

15.3 Target Profiling Strategy

When the source program has been compiled and linked you have an executable, `run`. Typing `run` at the command prompt will execute the program with a predefined Target Profiling Strategy. The profiling strategy is printed on the first four lines of output:

```
Profiling is turned on with options:
     profile timer (unix virtual timer) is turned on.
     a profile tick occurs every 1th second.
     profiling data is written on file profile.rp.
```

You can change the profiling strategy by passing command line arguments directly to the executable. The second line says that a virtual timer is used.

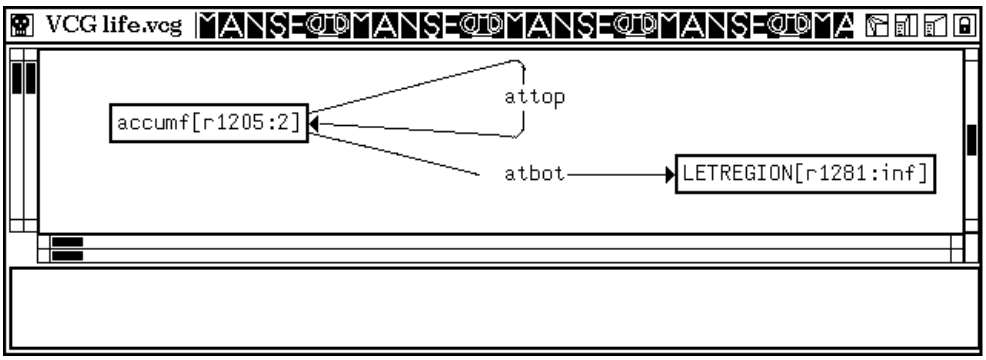


Figure 15.6: The figure shows a small fragment of the region flow graph.

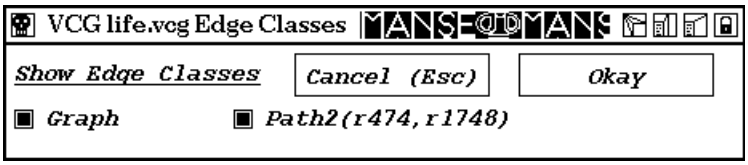


Figure 15.7: After choosing the **Expose/Hide edges** facility you get this window. The window shows that there are two *edge classes* in the graph; one for the region flow graph and one for the path from node `r474` to node `r1748`. If you have generated the path from section 15.2 you have the option `Path2(r314,r1588)`.

There are three possible timers, but in general it may be system dependent. On the HP-UX operating system you have the following timers:[8]

REAL which is real time.

VIRTUAL which is the process virtual time. It runs only when the process is executing.

PROF which is the process virtual time together with the time used in the operating system on behalf of the process.

You specify the timer to use by passing `-realtime`, `-virtualtime` or `-profiletime` to the executable.

[8] A complete description can be found in the manual page for `getitimer`.

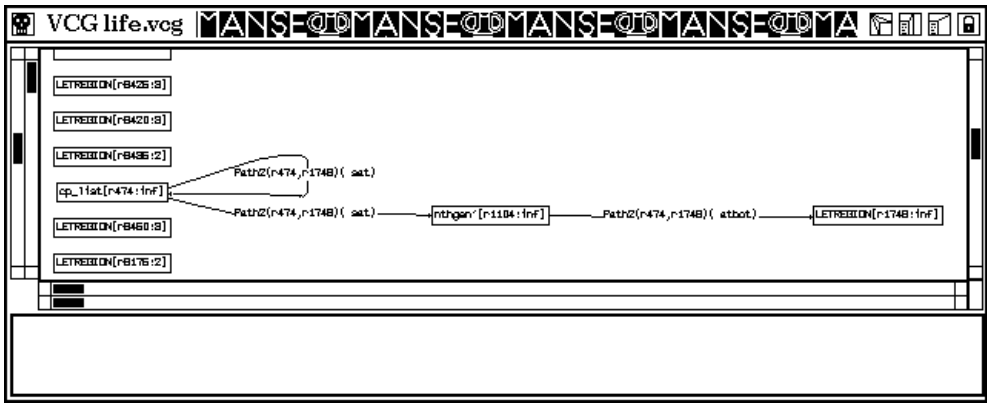


Figure 15.8: The Figure shows the path between node `r474` and `r1748`. If you look on page 149 you may notice that it is the same path as in the log file; the numbers have changed however, because they where generated in two different compilations.

The third line says that a *profile tick* occurs every 1 second. A profile tick is when the program stops normal execution, and memory is traversed to collect profile data. The more often a profile tick occurs the more detailed you profile. The *time slot* (the time between to succeeding profile ticks) to use is specified by the `-sec n` and `-microsec n` options. A time slot of half a second is specified by `-microsec 500000` and not by `-sec 0.5`.[9]

The fourth line tells you that the collected profile data is exported to file `profile.rp`. This can be changed by the `-file outFileName` option.

There are several other command line arguments which can be seen by the `-h` or `-help` options.

15.4 Executing `run`

After executing `run` some *region statistics* are printed on stdout. The region statistics are collected independently of the Target Profiling Strategy above and are exact values for the program.

[9]The lowest possible time slot to use is system dependent. It is also system dependent how long time that passes before the time wraps. This will not in practice happen on a HP-UX but it will happen after about 40 minutes on SUN OS4.

```
*************Region statistics***************


SBRK.
  Number of calls to sbrk                      :          3
  Number of bytes allocated in each SBRK call :      24240
  Total number of bytes allocated by SBRK     :      72720 (0.1Mb)

REGIONPAGES.
  Size of one page              :        800 bytes

  Max. no. of simultaneously allocated pages :         78
  Number of allocated pages now              :          3

REGIONS.
  Size of infinite region descriptor (incl. profiling information) :         28 bytes
  Size of infinite region descriptor (excl. profiling information) :         16 bytes

  Size of finite region descriptor   :          8 bytes

  Number of calls to allocateRegionInf   :     157771
  Number of calls to deallocateRegionInf :     157768

  Number of calls to allocateRegionFin   :    3457870
  Number of calls to deallocateRegionFin :    3457870

  Number of calls to alloc                  :    1446811
  Number of calls to resetRegion            :     139776
  Number of calls to deallocateRegionsUntil :          0

  Max. no. of co-existing regions (finite plus infinite) :        242
  Number of regions now                                  :          3

  Live data in infinite regions :         84 bytes ( 0.0Mb)
  Live data in finite regions   :          0 bytes ( 0.0Mb)
  ------------------------------------------------------
  Total live data               :         84 bytes ( 0.0Mb)

  Maximum space used for region pages                :      62400 bytes ( 0.1Mb)
  Maximum space used on data in region pages         :      27488 bytes ( 0.0Mb)
      Space in regions at that time used on profiling :      27576 bytes ( 0.0Mb)
  ------------------------------------------------------------------------------
  Maximum allocated space in region pages            :      55064 bytes ( 0.1Mb)

  Memory utilisation for infinite regions (     55064/     62400) : 88%

  Maximum space used on the stack for infinite region descriptors   :        400 bytes ( 0.0Mb)
      Additional space used on profiling information at that time   :        300 bytes ( 0.0Mb)
  ----------------------------------------------------------------------------------------------
  Maximum space used on infinite region descriptors on the stack    :        700 bytes ( 0.0Mb)

  Maximum space used on the stack for finite regions             :       6604 bytes ( 0.0Mb)
      Additional space used on profiling information at that time :       3584 bytes ( 0.0Mb)
  ----------------------------------------------------------------------------------------
  Maximum space used on finite regions on the stack              :      10188 bytes ( 0.0Mb)

  Max. size of stack when program was executed        :      11256 bytes ( 0.0Mb)
```

```
   Space used on profiling information at that time  :       3800 bytes ( 0.0Mb)
  ------------------------------------------------------------------------------
  Max. stack use excl. profiling information         :       7456 bytes ( 0.0Mb)

  Max. size of stack in a profile tick               :       5596 bytes ( 0.0Mb)

*********End of region statistics*********
```

The SBRK part above shows how memory is allocated from the operating system.

Each region consists of several *region pages* whose size is found in the REGIONPAGES part. The value

```
Max. no. of simultaneously allocated pages      :       78
```

multiplied by

```
Size of one page          :        800 bytes
```

gives the maximal memory use in infinite regions (62400 bytes).

In the REGIONS part, we see the number of calls to finite and infinite region operations, respectively. The target program has allocated 157771 infinite regions and deallocated 157768; hence three global regions were alive when the program finished; i.e. global regions are not necessarily deallocated explicitly before the program terminates.

No finite regions are alive (3457870 allocations and deallocations). We have allocated 1446811 objects in infinite regions. It has been possible to reset an infinite region 139776 times. The `deallocateRegionsUntil` operation is only used when raising exceptions, i.e. no exceptions have been raised.

Because objects allocated in infinite regions are not split across different region pages it is not always possible to fill out all region pages. The value

```
Memory utilisation for infinite regions (     55064/     62400) : 88%
```

shows memory utilisation at the moment where the program had allocated the largest amount of memory. The size of objects in finite regions allocated on the stack is shown together with the overhead produced by the profiler. The values

```
Max. size of stack when program was executed     :   11256 bytes ( 0.0Mb)
```

and

```
Max. size of stack in a profile tick             :    5596 bytes ( 0.0Mb)
```

can be used to see if it is necessary to profile more detailed. If the difference between the two figures is large you can profile with a smaller time slot.

After execution of the target program we have a profile data file named `profile.rp`.

15.5 Processing the profile data file

The profile datafile `profile.rp` can be processed by the graph generator `rp2ps` (read RegionProfiler2PostScript) found in the `bin` directory for the ML Kit version you use.[10] The graph generator is controlled by command line options.

A region profile is produced by the `-region` switch. Typing the UNIX command `rp2ps -region` produces a postscript file in file `region.ps`. The file `profile.rp` is used as profile data file. Figure 15.9 shows the region profile for the example program `life.sml`. The regions are sorted by size

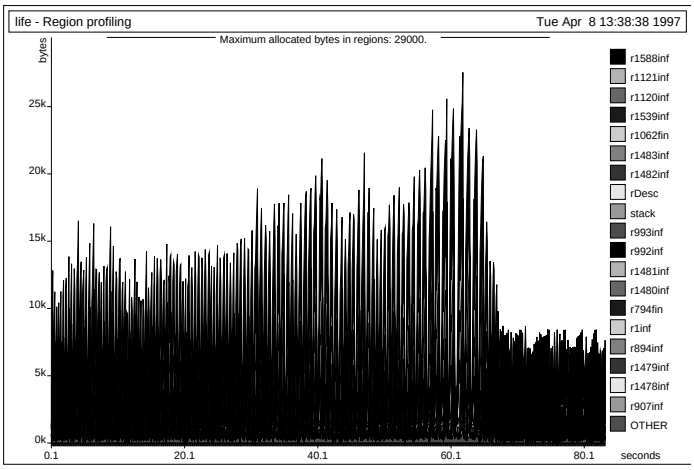


Figure 15.9: The region profile shows all regions and the stack with the region (or stack) having the largest area at the top. Executing the life program with `run -microsec 100000` and typing `rp2ps -sampleMax 1200 -region` produces this graph.

(area) with the largest at the top and the smallest at the bottom. If there are

[10] The `rp2ps` program is based on a profiler by Colin Runciman, David Wakeling and Niklas Röjemo.

more regions than can be shown in different shades, the smallest are collected in an *other band* at the bottom.

Each region is identified with a number that matches a `letregion`-bound region variable in the region-annotated lambda program. Infinite regions end with "`inf`" and finite regions with "`fin`". We also have a band `rDesc` and `stack`. The `rDesc` band shows the memory used on infinite "region descriptors" on the stack. The stack band shows stack usage excluding the region descriptors for the infinite regions.

The *max. allocation line* "Maximum allocated bytes in regions: ..." at top of Figure 15.9 shows the maximum number of bytes allocated in regions when the target program was executed. Because we also show the stack use on the graph (as the `rDesc` and `stack` band), we offset the max. allocation line upwards by the maximum stack use shown. The space between the max. allocation line and the top band shows the inaccuracy of the profiling strategy used. Having a large gap indicates that a smaller time slot should be used or maybe another Compile-Time Profiling Strategy.

An object profile is produced by the `-object` switch. If we want to examine the largest region shown in Figure 15.9, we type `rp2ps -sampleMax 1200 -object 1588` and get the object profile shown in Figure 15.10. We

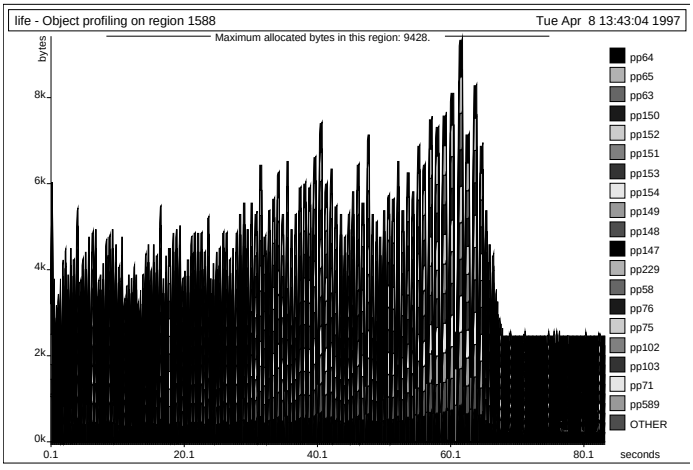


Figure 15.10: The object profile shows all allocation points allocating into this region.

see that allocation point `pp64` is responsible for the largest amount of alloca-

tions in the program. The allocation point may be found by searching after program point `pp64` in the region-annotated lambda program.

A stack profile (Figure 15.11) shows memory usage in the stack. A stack profile is generated with the `-stack` option to `rp2ps`.

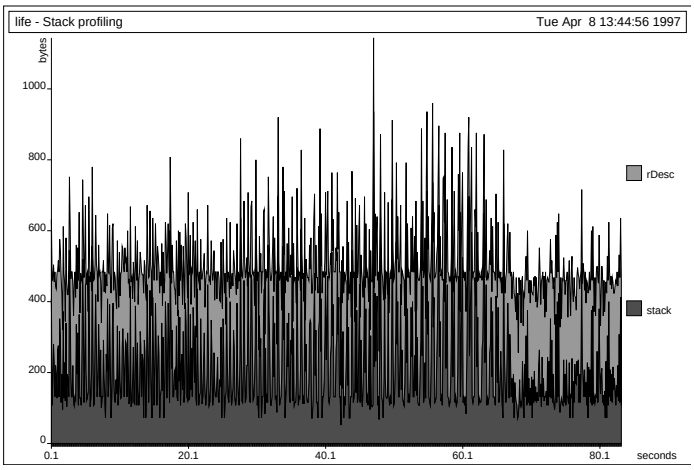


Figure 15.11: Memory usage on the stack.

15.6 More complicated graphs with `rp2ps`

This section gives a fast overview of the more advanced options which can be passed to `rp2ps`. First of all, it is possible to name the profiles with the `-name` option. Comments are inserted in the x-axis with the `-comment` option.

The profile data file may contain an large number of *samples* (the data collected by a profile tick is called a sample). By default, `rp2ps` only uses 64 samples. This may be changed with the `-sampleMax` option. The following two algorithms are used to sort out samples:

`-sortBySize` where the n (specified by `-sampleMax`) largest samples are kept.

`-sortByTime` is used by default and makes a binary deletion of samples by time such that the n samples shown will be equally distributed on the x-axis.

The `-sortBySize` option is handy if you get some profiles with a large gap between the top band and the *max. allocation line*. If there is a large gap when using option `-sortBySize`, then you have to profile with smaller time slots. You can use the `-stat` option to get the number of samples in the profile data file. It is printed as `Number of ticks:`.

Figure 15.12 shows the profile for the following command line:

```
rp2ps -region -sampleMax 50 -name life.sml
      -comment 9 "A comment at time 9" -sortByTime
```

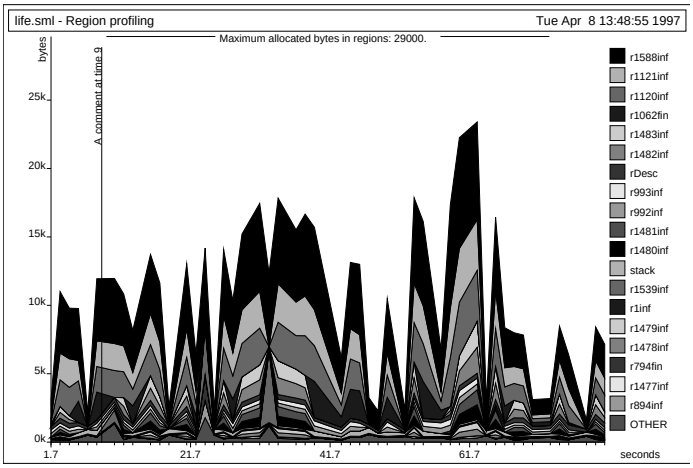


Figure 15.12: It is possible to insert comments in the profiles.

The graph generator recognize several options not shown above. They are printed on stdout when typing `rp2ps -h`.

Chapter 16

Interacting with the Kit

Starting the Kit was described in Section 3.6. To leave the Kit, type `q` followed by a return character.

In the following, we give an overview over the most important sub-menus. Section 16.4 explains how to set up a personal script file.

16.1 `Project` Menu for Separate Compilation

In Section 3.6 we described how software projects with multiple program units (source files) are compiled in the Kit. Once the project has been compiled and linked, the Kit manages what program units should be recompiled upon modification of source code. The system guarantees that the result of first touching one or more program units and then using the separate compilation system to re-build the system is the same as if all program units were recompiled.

A project file contains a sequence of names of all program units in the project. Source files must have extension `.sml` but this extension is omitted in the project file. Every source file must contain a top-level Standard ML declaration; the scope of the declaration is all the subsequent source files in the project file. Hence, a program unit may depend on program units mentioned earlier in the project file, but not vice versa. The meaning of an entire project is the meaning of the top-level declaration that would arise by concatenating all the source files listed in the project file, in the order they are listed. Thus, the separate compilation system is a way of avoiding recompiling parts of a (possibly) long sequence of declarations, while ensuring

that the result is always the same as if one had recompiled the entire program.

As an example, consider the project file below for the text scanning example.[1]

```
prelude
lib
scan
```

The actual source files for the project are `prelude.sml`, `lib.sml` and `scan.sml`. These files must be located in the source directory together with the project file `scan`. Whereas both the program units `lib` and `scan` depend on the program unit `prelude`, `scan` also depends on `lib`.

The `Project` sub-menu provides the user with operations for setting a project file name, reading a project file, showing the status of a project, compiling and linking a project, and operations for touching a program unit, upon modification of source code (See Section 3.6).

After a project has been successfully compiled and linked, it can be executed by running the command

```
run
```

in the target directory.

The Kit compiles each program unit of a project one at a time, in the order mentioned in the project file. A program unit is compiled under a given set of assumptions, providing for instance, region type schemes for free variables of the program unit. Also, compilation of a program unit gives rise to exported information about declared identifiers. Exported information may occur in assumptions for later program units.

A program unit is recompiled if either

1. the user has "touched" the program unit. One *touches* a program unit by selecting `Touch a program unit` or `Touch it again` from the `Project` menu. Typically, one does this after having modified the source file; the Kit does not keep track of file modification dates;

2. the assumptions under which the program unit was previously compiled, have changed.

[1] Project: `kitdemo/scan`.

To avoid unnecessary recompilation, compilation assumptions for a program unit only depend on the free identifiers of the unit. Further, if a program unit has been compiled earlier, the system will seek to *match* the new exported information to the old exported information, by renaming generated names to names generated when the program unit was first compiled. This allows the compiler to use fresh names (stamps) for implementing generative data types, e.g., and still achieve that a program unit is not necessarily recompiled even though a program unit, on which it depends, is modified.

In the text scanning example above, let us assume we modify and touch `lib`. Selecting "`Compile and link project`" will cause `lib` to be recompiled; then the Kit checks whether the assumptions under which the program unit `scan` was compiled have changed and if so, recompiles `scan`.

Modifying only comments or string constants inside `lib` or extending its set of declared identifiers does not trigger recompilation of `scan`.

However, more information is needed to compile a program unit than the ML type schemes for its free variables. Hence, it might be the case that a program unit must be recompiled even if the ML type assumptions about free variables have not changed. For instance, the region type scheme for a free variable might have changed, even if the underlying ML type scheme has not.

As an example, consider modifying the function `readWord` in unit `lib` to put its result in a global region. This will trigger recompilation of the program unit `scan`, since the assumptions under which it was previously compiled, have changed. Besides changes in region type schemes, changes in multiplicities and physical sizes of formal region variables of functions may also trigger recompilation.

16.2 Printing of Intermediate Forms Menu

The menu `Printing of intermediate forms` controls which intermediate forms are output on the log file. A summary of the major phases that produce printable intermediate forms is shown in Figure 16.1. The phases are listed in the order they take place in the Kit. The optimiser (which rewrites a *Lambda* program), collects statistics about the optimisation which can be printed out by turning on the flag `statistics after optimisation`.

The storage mode analysis (Chapter 12) outputs a *MulExp* expression that can be printed by turning on the flag `print atbot expression`.

Menu: Printing of Intermediate Forms

Phase	**Type of Result**	**Flag(s) which Print Result**
Elaboration	*Lambda*	(∗)
Elim. of Polym. Eq.	*Lambda*	(∗)
Lambda Optimiser	*Lambda*	(∗)
	report	`statistics after optimisation`
Spreading	*RegionExp*	(∗)
Region Inference	*RegionExp*	(∗)
Multiplicity Inference	*MulExp*	(∗)
K-normalisation	*MulExp*	
Storage Mode Analysis	*MulExp*	`print atbot expression`(∗)
Dropping of Regions	*MulExp*	`print drop regions expression`(∗)
Physical Size Inference	*MulExp*	`print physical size inference expression`(∗)
Call Conversion	*MulExp*	`print call-explicit expression`(∗)
Code Generation	KAM-code	`print KAM code before register allocation`(∗)
Register Allocation	KAM-code	`print KAM code after register allocation`(∗)

Figure 16.1: The table shows how the menu items in the "Printing of Intermediate Forms" correspond to the phases in the Kit. Enabling `debug compiler` from the `Debug Kit` menu causes all intermediate forms marked (∗) to be printed. Thus one can select phases individually or ask to have all printed. The phases that follow K-normalisation all work on K-normal forms, but, for readablity, terms are printed as though they had not been normalised (unless `Print in K-normal Form` from the `Layout` menu is enabled).

After that, regions with only **get** effects are removed from the *MulExp* expression (page 56). To see the result of that, turn on `print drop regions expression`.

After that, the *physical size inference* determines the size in words of finite region variables. For instance, a finite region that will contain a pair will have physical size two words. To see the result of the physical size inference, turn on `print physical size inference expression`.

After that, call conversion converts the *MulExp* to a call-explicit expression (page 130). To see the result, enable `print call-explicit expression`.

After that, KAM code is generated. The KAM code before the register allocation can be inspected by enabling `print KAM code before register allocation`, and the result of the register allocation can be viewed by enabling `print KAM code after register allocation`.

16.3 Layout Menu

While the switches described in the previous section concern which intermediate forms to print, the switches in the sub-menu `Layout` control how these forms are printed.

The flags `print types`, `print effects`, and `print regions` control the printing of types and places, effects and region allocation points ("`at` ρ"). All eight combinations of these three flags are possible, but if `print_effect` is turned on it is best also to turn the two others on so that one can see where the effect variables and the region variables which appear in arrow effects are bound..

Enabling `print in K-Normal Form` causes expressions to be output in K-Normal Form instead of the simplified form in which they are normally presented.

16.4 Creating your own Script File

If you have built the Kit yourself on an HP or a SUN using the distribution accessible from our web site, the Kit has already produced a script file for you and you do not have to modify it to get started. If somebody else has built the Kit locally, they should be able to refer you to a script file which you can take as a starting point; the only things you will have to change in

the script will be the constants listed in Section 16.4.1. Likewise, if you have downloaded an executable Kit from the Kit web site, you will have received with it a script file for the architecture in question and you only have to modify the constants listed in Section 16.4.1. If you are porting the Kit to a completely different platform, you need to take the script constants listed in Section 16.4.2 into account too.

16.4.1 Script constants concerning paths

Under the header `(* File *)` in the script file you should find the following constants

`source_directory` The directory where the Kit will look for your ML project and source files (including the prelude).

`target_directory` The directory where you want the Kit to put the target files it produces.

`log_to_file` True if log information should be written to a file rather than onto the screen.

`log_directory` The directory where the Kit will write log files.

`path_to_kit_script` The full file name of your script file.

`path_to_runtime` The full file name of the non-profiling runtime system of the Kit.

`path_to_runtime_prof` The full file name of the profiling runtime system of the Kit.

16.4.2 Platform-dependent Settings

The string constant `target_file_extension` is set to `".c"` if you generate C target code and to `".s"` if you generate HP target code.

The following settings

```
val kit_architecture : string = "HPUX"
val c_compiler : string = "cc -Aa"
val c_libs : string = "-lm"
```

may all have to be changed, depending on your platform and the C-compiler you use. See the `readme` and `roadmap` files in the distribution.

Chapter 17

Calling C Functions

In this chapter we describe how the ML Kit programmer can call C functions. C functions may be passed ML values and may return ML values. Not all ML values are represented as if they were C values. For instance, C strings are null-terminated arrays of characters, and this is not how the Kit represents ML strings. For this reason, a small number of conversion functions and macros are provided for converting C values to ML values and vice versa.

When a C function has to return boxed values of known size, finite regions are allocated by the Kit and pointers to them passed to the C function as extra parameters. When a C function has to return values of unbounded size, pointers to infinite regions are passed to the C function; in this case, the C function can itself allocate space in these infinite regions, using the primitives described below. In both cases, the Kit uses region inference to infer the lifetime of regions that are passed to the C-function. The region inference algorithm does not analyse C functions; it assumes that C functions that are called from ML code are region exomorphisms.

For every C function that is called from the ML code, the order of the region arguments (if any) is uniquely determined by the ML result type of the function. This type must be a monotype constructed from lists, records, booleans, reals, strings and integers.

Examples of existing libraries which one can access in this way are the X Window System and standard UNIX libraries containing functions like `time`, `cp` and `fork`. There are limitations to the scheme, however. First, since C and the ML Kit do not share value representations, transmitting large data structures between C and ML will involve significant copying, which might be a problem in practice. Second, some C libraries requires the user to set up

"call-back functions" to be executed when specific events occur. However, it is not currently possible to have a C function call an ML Kit function.

17.1 Declaring Primitives and C Functions

The Kit comes with a prelude (see the file `kitdemo/prelude.sml`) defining some of the initial basis of the 1990 Definition. The declarations in the prelude use a special built-in identifier called `prim` which is declared to have type scheme $\forall\alpha\beta.(\texttt{int} * \alpha) \rightarrow \beta$ in the initial environment. A primitive function is then declared in the prelude by passing a number to `prim`. For example

```
fun op = (x: ''a, y: ''a): bool = prim(0, (x, y))
```

declares polymorphic equality. The argument and result types are explicitly stated in order to give the primitive the correct type scheme.

Primitive number 31 is used for calling functions written in C. The second argument to `prim` is a tuple holding as the first component a string containing the name of the C function. The second component of the tuple is another string containing the name of the C function to be used when profiling is enabled. The remaining components of the tuple are arguments to the C function:

$$\begin{array}{l}\texttt{fun}\ \mathit{ml\_fun}\ (x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau = \\ \quad \texttt{prim}(31, (\mathit{c\_func}, \mathit{c\_funcProf}, x_1, \ldots, x_n))\end{array}$$

The result type τ must be of the following form (no type variables are allowed):

$$\tau ::= \texttt{int} \mid \texttt{bool} \mid \tau_1 * \ldots * \tau_n \mid \tau\ \texttt{list} \mid \texttt{real} \mid \texttt{string} \mid \texttt{unit}$$

If the result type is one of `int`, `bool` or `unit` then the result value can be returned unboxed. If the result type represents a boxed value, the C function must be told where to store the value. For any type which is either `real` or a non-empty tuple type, and does not occur in a list type of the result type τ, the Kit allocates space for the value and passes a pointer to the allocated space as an additional argument to the C function. For any type representing a boxed value which is either `string` or occurs in a list type of the result type τ, the Kit cannot statically determine the amount of space needed to

store the value. Instead, regions are passed to the C function as additional arguments and the C function must then explicitly allocate space in these regions as needed, using a C function provided by the runtime system. The order in which these additional arguments are passed to the C function is determined by a pre-order traversal of the result type τ. For a list type, regions are given in the order:

1. Region for cons-cells;
2. Region for auxiliary pairs;
3. Regions for elements, if necessary.

Below we give an example to show what extra arguments are passed to a C function, given the result type. In the example, we use the following (optional) naming convention: names of arguments holding addresses of pre-allocated space in regions start with `vAddr`, while names of arguments holding addresses of region descriptors (to be used for allocation in an infinite region) start with `rAddr`.

Example 1 Given the result type (`int` $*$ `string`) `list` $*$ `real`, the following extra arguments are passed to the C function (in order): `vAddrPair`, `rAddrLCons`, `rAddrLPairs`, `rAddrEPairs`, `rAddrEStrings` and `vAddrReal`, see Figure 17.1.

Here `vAddrPair` holds an address pointing to pre-allocated storage in which the tuple of the list and the (pointer to the) real should reside. The arguments `rAddrLCons` and `rAddrLPairs` hold region addresses for the spine and the auxiliary pairs of the list, respectively. Similarly, `rAddrEPairs` and `rAddrEStrings` hold region addresses for element pairs and strings, respectively. The argument `vAddrReal` holds the address for pre-allocated storage for the real.

Additional arguments holding pointers to pre-allocated space and infinite regions are passed to the C function prior to the original ML arguments. Consider again the ML declaration

$$\begin{array}{l}\texttt{fun}\ \mathit{ml\_fun}\ (x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau = \\ \quad \texttt{prim}(31, (\mathit{c\_func}, \mathit{c\_funcProf}, x_1, \ldots, x_n)).\end{array}$$

The C function *c_func* is then declared as

$$\texttt{int}\ \mathit{c\_func}\ (\texttt{int}\ \mathit{addr}_1, \ldots, \texttt{int}\ \mathit{addr}_m,\ \texttt{int}\ x_1, \ldots, \texttt{int}\ x_n)$$

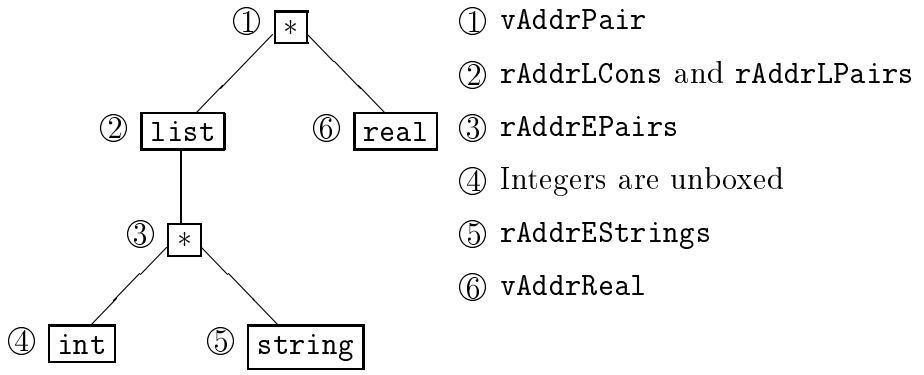


Figure 17.1: The order of pointers to allocated space and infinite regions is determined from a pre-order traversal of the result type (`int*string`) `list*real`.

where $addr_1$, ..., $addr_m$ are pointers to pre-allocated space and infinite regions as described above.

To support profiling the programmer must provide special profiling versions of some C functions. When profiling is enabled and at least one pointer to pre-allocated space or to an infinite region is passed to the C function, then also a single program point representing the call of the C function is passed. The program point has to be used when allocating into infinite regions. This is explained in Section 17.4. The program point is passed as the last argument:

`int` *c_funcProf* (`int` $addr_1, \ldots,$ `int` $addr_m,$
`int` $x_1, \ldots,$ `int` $x_n,$ `int` *pPoint*)

C functions that do not allocate into infinite regions can be used unchanged when profiling.[1]

[1]For simplicity, we have chosen to pass the program point even though the C function only uses pre-allocated space. Because we pass the program point as the last argument to the C function we need not have the program point as a formal parameter in the C function. The program point is passed but not used.

17.2 Conversion Macros and Functions

We provide a small conversion library of macros and functions for use by C functions that need to convert between ML values and C values. Using the library whenever values are passed between C and ML will protect you against any future change in the representation of ML values. The interface to the library is provided through the include file `src/Runtime/Version17/MlConvert.h`.[2]

17.2.1 Integers

There are two macros for converting between the ML representation of integers and the C representation of integers:[3]

```
#define convertIntToC(i)
#define convertIntToML(i)
```

To convert an ML integer (ML_{int}) to a C integer (C_{int}) write

C_{int} = `convertIntToC`(ML_{int});,

To convert a C integer (C_{int}) to an ML integer (ML_{int}) write

ML_{int} = `convertIntToML`(C_{int});.

The above macros are used in the examples 2, 3 and 6 in Section 17.7.

17.2.2 Units

The following constant in the conversion library denotes the ML representation of `()`:

```
#define mlUNIT
```

17.2.3 Reals

An ML real is represented as a pointer into a region containing the real. To convert an ML real to a C real we dereference the pointer. To convert a C real to an ML real, we update the memory to contain the ML real. The following two macros are provided:

[2] There is also a symbolic link to this file in the `kitdemo` directory.

[3] In this release of the Kit, these macros are the identity maps, but that may change.

```
#define convertRealToC(mlReal)
#define convertRealToML(cReal, mlReal)
```

Converting from an ML real to a C real can be done with the first macro:

C_{real} = `convertRealToC`(ML_{real});.

Converting from a C real to an ML real (being part of the result value of the C function) can be done in one or two steps depending on whether the real is part of a list or not. If the real is not in a list the memory containing the real has been allocated before the C call (See Section 17.1):

`convertRealToML`(C_{real}, ML_{real});.

If the ML real is in a list element, then space must be allocated for the real before converting it. If ρ_{real} is the region for the real you write:

`allocReal`(ρ_{real}, ML_{real});
`convertRealToML`(C_{real}, ML_{real});

The above macros are used in the examples 3, 6 and 8 in Section 17.7.

17.2.4 Booleans

Four constants provide the values of true and false in ML and in C. These are defined by the following macros:[4]

```
#define mlTRUE  3
#define mlFALSE 1
#define cTRUE   1
#define cFALSE  0
```

Two macros are provided for converting booleans:

```
#define convertBoolToC(i)
#define convertBoolToML(i)
```

Converting booleans are similar to converting integers:

C_{bool} = `convertBoolToC`(ML_{bool});
ML_{bool}=`convertBoolToML`(C_{bool});.

[4]Booleans in the Kit are tagged for historical reasons.

17.2.5 Records

Records are boxed. One macro is provided for storing and retrieving elements:

```
#define elemRecordML(recAddr, offset)
```

An element can be retrieved by writing

ML_{elem} = `elemRecordML`(ML_{rec}, *offset*);

where the first element has offset 0. An element is stored by

`elemRecordML`(ML_{rec}, *offset*) = ML_{elem};

Two specialized versions of the above macros are provided for pairs:

```
#define first(x)
#define second(x)
```

If the record is in a list element then it is necessary to allocate the record before using it. This is done with the macro

```
#define allocRecordML(rhoRec, size, recAddr)
```

where `rhoRec` is a pointer to a region descriptor, `size` is the size of the record (i.e., the number of components), and `recAddr` is a variable in which `allocRecordML` returns a pointer to storage for the record. The record is then stored, component by component, by repeatedly calling `elemRecordML` with the pointer as argument.

The above macros are used in examples 8, 9 and 7 in Section 17.7.

17.2.6 Strings

Strings are boxed and always allocated in infinite regions. It is possible to print an ML string by using the C function

```
void printString(StringDesc *str);
```

Strings are converted from ML to C and vice versa using the two C functions

```
void convertStringToC(StringDesc *mlStr, char *cStr, int cStrLen);
StringDesc *convertStringToML(int rAddr, char *cStr);.
```

An ML string is converted to a C string by writing

`convertStringToC`(ML_{str},C_{str}, $cStrLen$);

and a C string is converted to an ML string by writing

ML_{str} = `convertStringToML`($rhoStr$, C_{str});

When using `convertStringToC`, the C string (C_{str}) has to be allocated in advance. The length of the pre-allocated C string is also passed to `convertStringToC`. If the ML string is larger than the C string an error message is written on `stdout` and the program will terminate. The following function returns the size of an ML string:

```
int sizeString(StringDesc *str);
```

The above macros are used in the examples 7 and 5 in Section 17.7.

17.2.7 Lists

Lists are always allocated in infinite regions. A list uses, as a minimum, two regions. One region for the constructors (`NIL` and `CONS`) and one region for the auxiliary pairs (Figure 17.2).

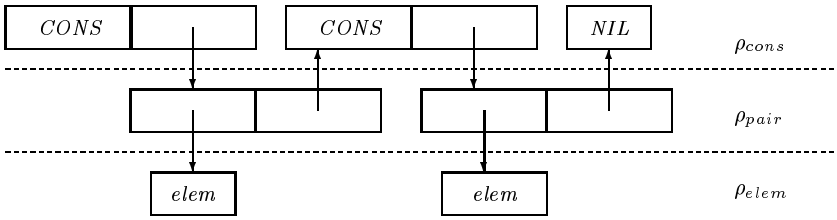


Figure 17.2: A list is constructed with constructors (`NIL` and `CONS`) and pairs. The constructors are allocated in region ρ_{cons} and the pairs in region ρ_{pair}. If the elements are boxed then they are allocated in one or more infinite regions. In this Figure we assume one infinite region for the elements.

We will now show three examples of manipulating lists. The first example runs through a list. Consider the following C function template:

```
int run_through_a_list(int list) {
  int ls;
  int elemML;
  for (ls=list; isCONS(ls); ls=tl(ls)) {
    elemML = hd(ls);
    /*do something with the element*/
  }
  return;
}
```

The ML list is passed to the C function in parameter `list`. The example uses a simple loop to run through the list. The parameter `list` points at the first constructor in the list. Each time we have a `CONS` constructor we also have an element, see Figure 17.2. The element can be retrieved with the `hd` macro. One obtains the tail of the list by using the `tl` macro.

The following four macros are provided in the conversion library.

```
#define isNIL(x)
#define isCONS(x)
#define hd(x)
#define tl(x)
```

The next example explains how to construct a list backwards. Consider the following C function template.

```
int construct_list_backwards(int consRho, int pairRho) {
  int *resList, *pair;
  makeNIL(consRho,resList);
  while (/*more elements*/) {
    ml_elem = ...;
    allocRecordML(pairRho, 2, pair);
    first(pair) = (int) ml_elem;
    second(pair) = (int) resList;
    makeCONS(consRho, pair, resList);
  }
  return (int) resList;
}
```

First we make the `NIL` constructor which marks the end of the list. Each time we have an element, we allocate a pair. We store the element in the first cell of the pair. A pointer to the list (constructed so far) is put in the second cell of the pair. We then allocate a new `CONS` constructor, now being the first constructor in the list. The pair is the argument given to the `CONS` constructor. We have assumed that the elements are unboxed, so that no regions are necessary for the elements.

The last example shows how a list can be constructed forwards. It is more clumsy to construct the list forwards because we have to return a pointer to the first element. Consider the following C function template.

```
int construct_list_forwards(int consRho, int pairRho) {
  int *pair, *cons, *temp_pair, res;

  /* The first element is special because we have to    */
  /* return a pointer to it.                            */
  ml_elem = ...
  allocRecordML(pairRho, 2, pair);
  first(pair) = (int) ml_elem;
  makeCONS(consRho, pair, cons);
  res = (int) cons;

  while (/*more elements*/) {
    ml_elem = ...
    allocRecordML(pairRho, 2, temp_pair);
    first(temp_pair) = (int) ml_elem;
    makeCONS(consRho, temp_pair, cons);
    second(pair) = (int) cons;
    pair = temp_pair;
  }
  makeNIL(consRho, cons);
  second(pair) = (int)cons;
  return res;
}
```

We construct the `CONS` constructor and pair for the first element and return a pointer to the `CONS` constructor as the result. We then construct the rest of the list by constructing a `CONS` constructor and a pair for each element. It is

necessary to use a temporary variable for the pair (in the example `temp_pair`) because we have to update the pair for the previous element. We let the last pair point at a `NIL` constructor to denote the end of the list.

The two macros `makeCONS` and `makeNIL` are provided in the conversion library:

```
#define makeNIL(rAddr, ptr)
#define makeCONS(rAddr, pair, ptr)
```

17.3 Exceptions

C functions are allowed to raise exceptions and it is possible for the ML code to handle these exceptions. A C function cannot declare exceptions locally. As an example, consider the following ML declaration.

```
exception EXN
fun raiseif0 (arg : int) : unit =
     prim(31, ("raiseif0", "raiseif0", arg, EXN))
```

If we want the function `raiseif0` to raise exception `EXN` if the argument (`arg`) is 0 then we use the function `raise_exn` provided by the conversion library. The C function may be declared thus:

```
int raiseif0(int arg, int exn) {
  int c_int;
  c_int = convertIntToC(arg);
  if (c_int = 0) {
    raise_exn(exn);
    return;
  }
  return mlUnit;
}
```

Exceptions are used in examples 6 and 7 in Section 17.7.

17.4 Profiling

It is necessary to make special versions of those C functions that allocate into infinite regions if the Kit profiler is used.

When profiling is enabled, an extra argument is passed to some of the C functions. The argument is an integer identifying the allocation point representing the C call in the lambda program (Chapter 15).

The conversion library contains special versions of various allocation macros and functions presented earlier in this chapter:

```
#define allocRealProf(realRho, realPtr, pPoint)
#define allocRecordMLProf(rhoRec, ssize, recAddr, pPoint)
StringDesc *convertStringToMLProfiling(int rhoString,
                                       char *cStr,
                                       int pPoint);
#define makeNILProf(rAddr, ptr, pPoint)
#define makeCONSProf(rAddr, pair, ptr, pPoint)
```

As an example, we show the profiling version of the C function `construct_list_backwards`, presented earlier.

```
int construct_list_backwardsProf(int consRho,
                                  int pairRho,
                                  int pPoint) {
  int *resList, *pair;
  makeNILProf(consRho,resList, pPoint);
  while (/*more elements*/) {
    ml_elem = ...;
    allocRecordMLProf(pairRho, 2, pair, pPoint);
    first(pair) = (int) ml_elem;
    second(pair) = (int) resList;
    makeCONSProf(consRho, pair, resList, pPoint);
  }
  return (int) resList;
}
```

The above example shows that it is not difficult to make the profiling version of a C function; use the "`Prof`" versions of the macros and use the extra argument `pPoint`, appropriately. The same program point is used for all allocations in the C function, perceiving the C function as one entity.

17.5 Storage Modes

As described in Chapter 12 (page 95), actual region parameters contain a storage mode at runtime, if the region is infinite. A C function may check the storage mode of an infinite region to see whether it is possible to reset the region before allocating space in it. The conversion library provides a macro, `is_inf_and_atbot(x)`, which can be used to test whether resetting is safe, assuming that the arguments to the C function are dead.

The C function `resetRegion`, which is provided by the conversion library, can be used to reset a region as in the C function template below.

```
int construct_list_backwards(int consRho, int pairRho) {
  int *resList, *pair;

  if (is_inf_and_atbot(consRho))
    resetRegion(consRho);
  if (is_inf_and_atbot(pairRho))
    resetRegion(pairRho);

  makeNIL(consRho,resList);
  ...
```

The C programmer should be careful not to reset regions that could contain live values. In particular, the C programmer must be conservative and take into acount possible region aliasing between regions holding arguments and regions holding the result. Clearly, if a region which the C function is supposed to return a result in contains part of the value argument(s) of the function, then the function should not first reset the region and then access the argument(s).

17.6 Compiling and Linking

To use a set of C functions in the ML code, one must first compile the C functions into an object file. (Remember to include the conversion library.)

As an example, the file `kitdemo/my_lib.c` holds a set of example C functions. On the HPUX system this file is compiled by typing[5]

[5] On the SUN OS4 system type `gcc -ansi -o my_lib.o -c my_lib.c`.

```
cc -Aa -D_HPUX_SOURCE -o my_lib.o -c my_lib.c
```

in the `kitdemo` directory. If the C functions are used with profiling type[6]

```
cc -Aa -D_HPUX_SOURCE -DPROFILING -o my_lib_prof.o -c my_lib.c
```

The project `ccalls` located in the `kitdemo` directory demonstrates calls to C functions. The project must be compiled and linked with the file `my_lib.o` (or the file `my_lib_prof.o` if profiling is enabled.) This is done by first modifying the `link with library` string in the `Control` menu so that it contains the full name of `kitdemo/my_lib.o`, for example

```
"/home/thor1/jane/kitdemo/my_lib.o -lm"
```

and then compiling the project as usual. If profiling is enabled, the link string must instead refer to `kitdemo/my_lib_prof.o`, e.g.,

```
"/home/thor1/jane/kitdemo/my_lib_prof.o -lm"
```

The project is then executed as usual.

17.6.1 Auto Conversion

For C functions that are simple, in a sense which is defined below, the Kit can generate code which automatically converts arguments from ML to C and results from C back to ML.

Auto conversion is enabled by prepending a `@`-character to the name of the C function, as in the following example:

```
fun power (base : int, n : int) : int =
     prim(31, ("@power", base, n))
```

The power function may then be implemented in C as follows:

```
int power(int base, int n) {
  int p;
  for (p = 1; n > 0; --n)
    p = p * base;
  return p;
}
```

[6] On the SUN OS4 system type `gcc -ansi -o my_lib_prof.o -DPROFILING -c my_lib.c`.

No explicit conversion is needed in the C code. Auto conversion is only supported when the arguments of the ML function are of type `int` or `bool` and when the result has type `unit`, `int` or `bool`. It also works when profiling is enabled.

Auto conversion is used in example 4 in Section 17.7.

17.7 Examples

Several example C functions are located in the file `kitdemo/my_lib.c` and the project `kitdemo/ccalls` makes use of these functions.

The following ML declarations are located in the `ccalls.sml` file which is part of the `ccalls` project, search after

```
(*-----------------------------------------------*)
(* Interface functions that call prim(31, ...)  *)
(*-----------------------------------------------*)

fun power(base : int, n : int) : int =
     prim(31, ("power", "power", base, n))
fun power_auto(base : int, n : int) : int =
     prim(31, ("@power_auto", "@power_auto", base, n))
fun power_real (base : real, n : int) : real =
     prim(31, ("power_real", "power_real", base, n))
fun print_string_list (string_list) : unit =
     prim(31, ("print_string_list","print_string_list",
               string_list))
exception Power of string
fun power_exn (base : real, n : int) : real =
     prim(31, ("power_exn", "power_exn",
               base, n, Power "This is power"))
exception DIR of string
fun dir (directory : string) : string list =
     prim(31, ("dir", "dirProf", directory,
               DIR "Cannot open directory"))
fun real_list () : real list =
     prim(31, ("real_list", "real_listProf"))
fun change_elem (p : int*string) : string*int =
      prim(31, ("change_elem", "change_elem", p))
```

The implementation of each of the C functions is summarized below (see the file `my_lib.c` for detailed comments.)

Example 2 The `power` function shows how to convert integers. This is done with the macros `convertIntToC` and `convertIntToML`.

Example 3 The `power_real` function shows how to convert reals. This is done with the macros `convertRealToC` and `convertRealToML`.

Example 4 The `power_auto` function shows the use of auto conversion. This is the easiest way of declaring C functions. The same C function may be called from the Kit and other C programs.

Example 5 The `print_string_list` example shows how to run through a list of strings. The method can easily be extended to running through lists of lists of lists, etc.

Example 6 The `power_exn` function shows how an exception can be raised from a C function. Note that it is necessary to `return` from the C function after you have called the `raise_exn` function.

Example 7 The `dir` function shows how a list can be constructed backwards. We use the UNIX system calls `opendir` and `readdir` to read the contents of the specified directory.

Note also that we check the infinite regions for resetting at the start of the C function. The checks must be placed at the start of the function, orelse not inserted at all.

If you compare the C functions `dir` and `dirProf` you may notice how the function `dir` is modified to work with profiling.

Example 8 Function `real_list` constructs a list of reals forwards. The reals are allocated in an infinite region. It may be more convenient to construct the list backwards in the C function and then apply a list reverse function on the result list in the Kit program.

Example 9 Function `change_elem` shows the use of macro `elemRecordML`. The result type is `string*int`. The function swaps the two elements in the pair. The Kit passes an address to pre-allocated space for the result pair, and an infinite region for the result string.

At first thought it should be enough to just swap the two arguments, and not copy the string into the string region, i.e. one could write the following function:

```
int change_elem(int newPair, int stringRho, int pair) {
  int firstElem_ml, secondElem_ml;

  firstElem_ml = elemRecordML(pair, 0);
  secondElem_ml = elemRecordML(pair, 1);

  elemRecordML(newPair, 0) = secondElem_ml;
  elemRecordML(newPair, 1) = firstElem_ml;

  return newPair;
}
```

This function may work sometimes but it is not always safe! Region inference expects the result string to be allocated in `stringRho`, and may therefore de-allocate the region containing the argument string, `secondElem_ml`, while the string in the returned pair is still alive. A safe version of `change_elem` is found in `my_lib.c`.

Index

Global Regions

`r1`	Holds values of type `top`, i.e., records, exceptions and closures;
`r2`	This region does not actually exist; it is used with unboxed values, such as integers, booleans and the 0-tuple.