Technical Report DIKU-TR-98/25
Department of Computer Science
University of Copenhagen
Universitetsparken 1
DK-2100 KBH Ø
DENMARK

Programming with Regions in the ML Kit (for Version 3)

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

December 3, 1998

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 unboxed (i.e., not region allocated). Auxiliary pairs in one region; elements in zero or more regions. Size of auxiliary pairs: two 32-bit words.
α tree	A tree and its subtrees reside in one region. Elements in one region (if not unboxed).
`exn`	Exception values are boxed and are always 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 stack allocated region descripter, 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 compiler for full Standard ML, including Modules and the SML Basis Library. It is intended for the development of stand-alone applications that must be 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 seven 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 that 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 that 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, exceptions, 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 that 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 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 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 ML Kit.

The present report describes the ML Kit Version 3. This version of the ML Kit extends the ML Kit Version 2 with support for the Standard ML Modules language. The ML Kit Version 2 is a further development of the ML Kit Version 1, which was developed at Edinburgh University and Copenhagen University. The ML Kit (after Version 1) is also called the ML Kit with Regions. 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 here, please feel free to write. Further information is available at our web site:

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

August, 1998

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 that 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, because 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, because it is, in general, very hard to make sure that none of the values that reside in the memory that 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 leak space, that is, it might use more memory than expected, perhaps eventually, exhaust 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 that 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 that cannot be checked by the compiler and that 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

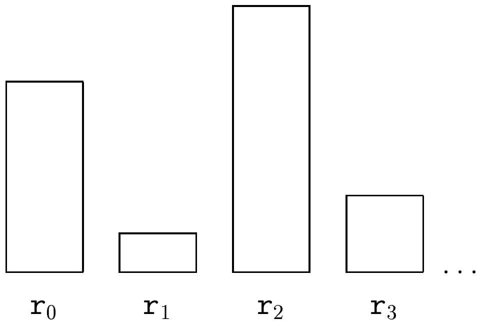


Figure 1.1: The store is a stack of regions; every region is depicted by a box in the picture.

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 that 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 notice 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 case 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 has operations for allocating, de-allocating, and extending regions. But it also has an explicit operation for resetting an existing region, that is, 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; it prints thorough explanations of why it thinks resetting might be unsafe otherwise. The 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. The function `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

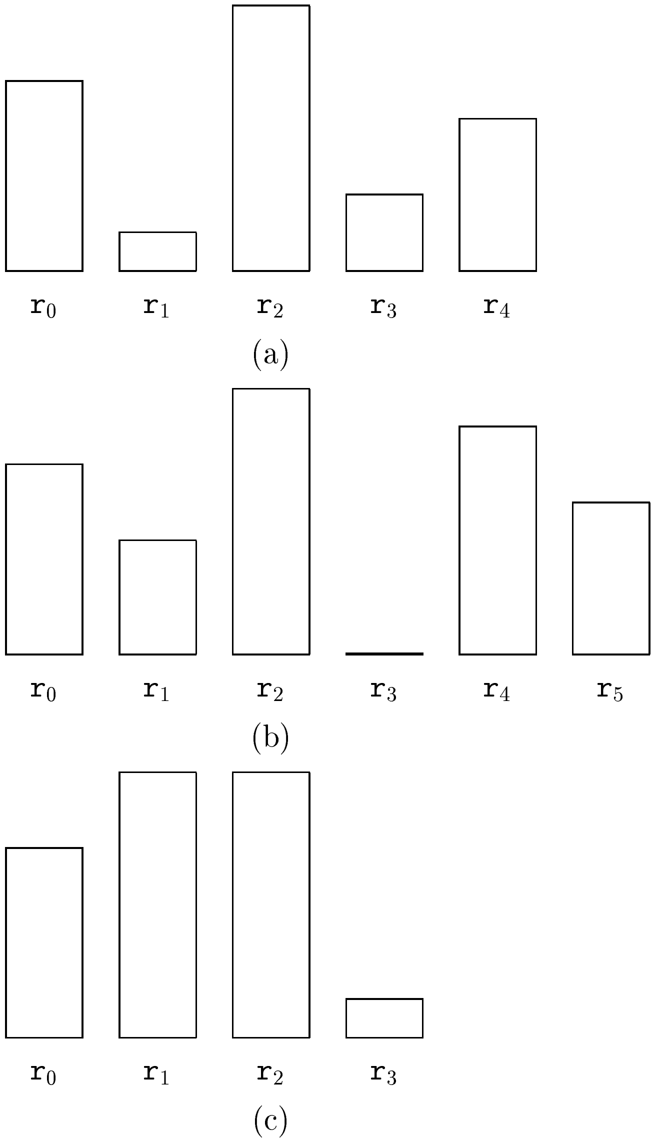


Figure 1.2: Further development of the region stack: (a) after allocation of $\mathtt{r}_4$; (b) after growth of $\mathtt{r}_1$ and $\mathtt{r}_4$, resetting of $\mathtt{r}_3$ and allocation of $\mathtt{r}_5$; (c) after popping of $\mathtt{r}_4$ and $\mathtt{r}_5$ but extension of $\mathtt{r}_1$ and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

that we feel capture natural programming intuitions. Moreover, the Kit implementation 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, that is, 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 tool makes it possible to see what 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:

```
             0
            0 0
           0   00        0
00         0   00     0000   0
00         0   00    0000    0
            0 0      0  0        00
             0       0000        00
                      0000


             0
            0000
           00 0 0      0 0
00        000 0  0   0   0
00         00 0 0    0
            0000    0    0       00
             0       0           00
                     0   0
                       00
```

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

l_n: list of integer pairs representing generation n.

`r0`

The computation of the next generation involves a considerable amount of list computation. Chris Reade 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

```
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) Chris Reade's Game of Life program.

ever needed. Similarly, giving each generation a separate region on the region stack is no good either, because 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 new region into the old region. This effect 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`, because `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 that 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 that 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. Because each of the region management directives that the compiler inserts in the code are constant time and space operations, we have now avoided unpredictable interruptions due to memory management. This avoidance of unpredictable interruptions 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 1.5 and 1.6. The highest amount of memory used for regions during the computation is 26,060 bytes. Figure 1.6, which has data collected from 200 snapshots of the computation, clearly shows that most of the 26,060 bytes are reclaimed between every two generations of the game. It turns out that the game essentially stabilises with a small number of live positions on the board after roughly 150 generations.

[3]The source file for the life program is `kitdemo/life.sml`. Running programs is described in Section 2.8. When run with n=10000 on the HP PA-RISC, the memory consumption (resident memory, measured using `top`) quickly reaches 180Kb and stays there for the remaining generations.

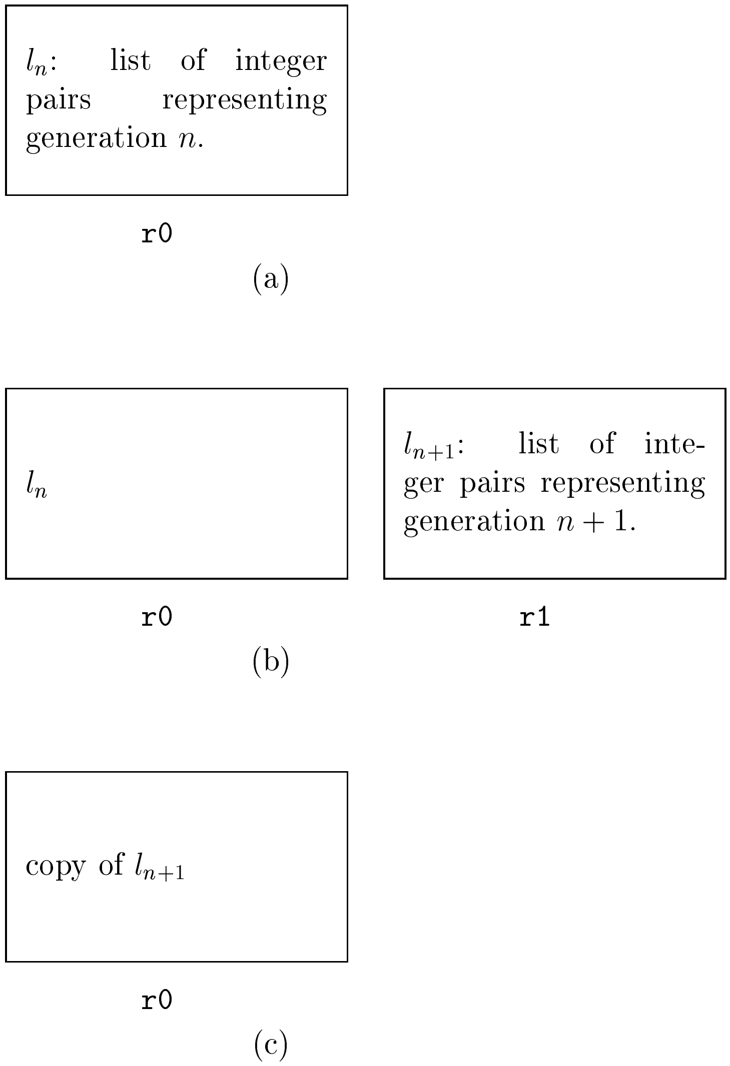


Figure 1.4: Using double-copying in the game of Life: (a) generation number n resides in region `r0`; (b) generation $(n+1)$ has been built in `r1`; (c) region `r0` has been reset, the new generation copied into `r0` and `r1` has been deallocated.

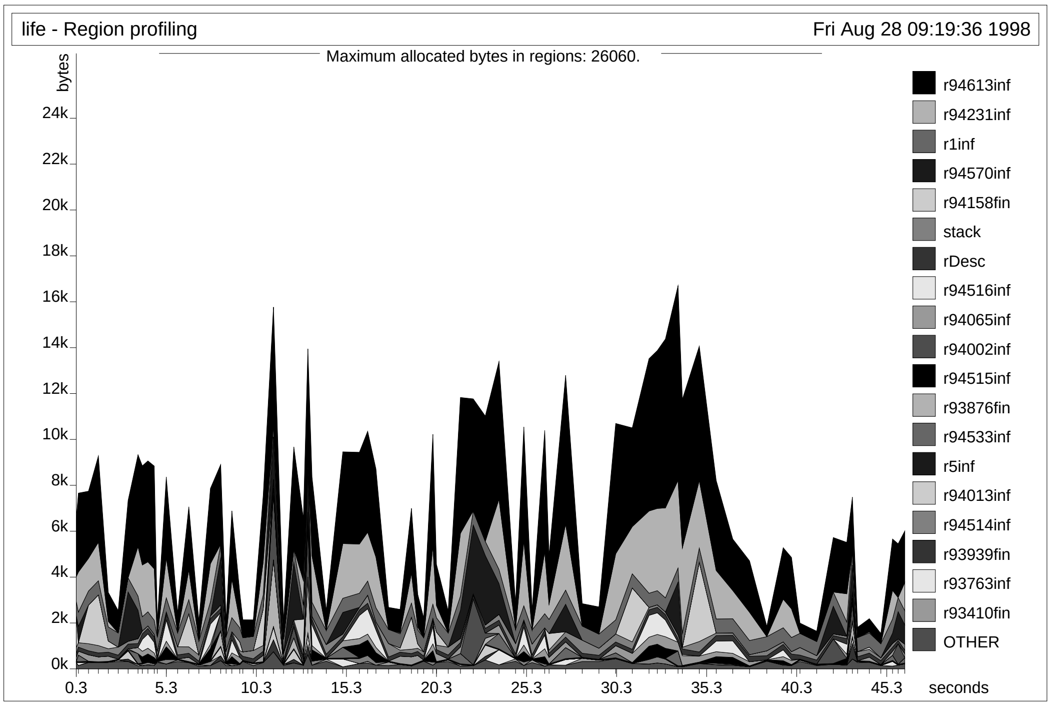


Figure 1.5: A region profile of two hundred generations of the "Game of Life", showing region sizes as a function of time (80 snapshots).

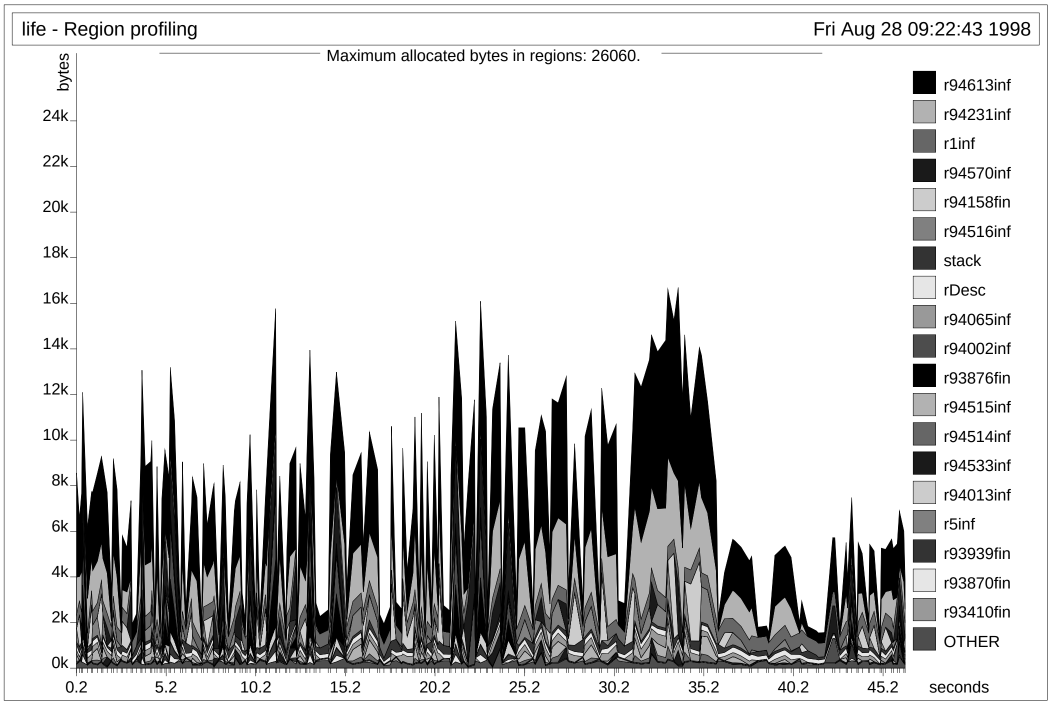


Figure 1.6: Region profile of two hundred generations of the "Game of Life", showing region sizes as a function of time (200 snapshots).

This stabilisation is clearly reflected in the region profile.

Figure 1.5 is from the same computation, but it only includes data from 80 snapshots. This figure makes it easier to see that the largest regions are `r94613` and `r94231`. To find out what these regions contain, however, one needs to know about the methods described in Part II.

1.3.1 Try it!

This section tells you how to repeat the profiling experiment shown above.

Compile the SML program `kitdemo/life.sml` as follows. First, make a personal copy of the `kit/kitdemo` directory, place yourself in it, and start the kit:[4]

```
kit
```

[4]We assume that you have added the `kit/bin` directory to your `PATH` shell variable.

Select `Profiling` from the Kit menu (type 5, then carriage return). Toggle `region profiling` (type 0, then carriage return). Go up one level (type `u`, then carriage return). Select `Compile an sml file`. Type `"life.sml"` (including the quotes, then return). After the Kit has compiled `life.sml`, type `quit`. The executable life program is `kitdemo/run`.

Next, you run `run`, as follows:

```
run -microsec 10000
```

This will make a profiling snapshot every 10,000 microseconds (i.e., every ten milliseconds). If you are satisfied with less fine-grained information, choose a larger number; it will speed up execution. If you just type

```
run
```

there will be one snapshot per second.

Finally, you create a PostScript file and view it as follows:[5]

```
rp2ps -region -name life -sampleMax 80
ghostview -seascape region.ps
```

The option `-sampleMax` n instructs `rp2ps` to show at most n snapshots (evenly distributed over the duration of the computation).

1.4 Including a Profile in a LaTeX Document

Figure 1.5 was produced by first executing the command

```
rp2ps -region -name life -sampleMax 80 -eps 137 mm
```

The option `-eps 137 mm` has the effect that `region.ps` becomes an encapsulated PostScript file. The resulting `region.ps` was renamed `life80.ps` and included in this document as follows:

```
\begin{figure}
\begin{center}
\includegraphics{life80.ps}
\end{center}
\caption{A region profile of two hundred
```

[5] The program `rp2ps` can be found in the `kit/bin` directory.

```
generations of the ``Game of Life'', showing
region sizes as a function of time (80 snapshots).}
\label{lifeprof80.fig}
\end{figure}
```

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 reasons, 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 that 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 the region pages of

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

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, that is, 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 that 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 is partitioned 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 that the user can safely use as a worst-case model of what happens at runtime.

2.5 Boxed and Unboxed Representation of Values

As is common with implementations of programming languages, we distinguish between *boxed* and *unboxed* representation of values in the KAM. An *unboxed* value is one that is stored in a register or a machine word. A *boxed* *value* is one that is represented by a word-size pointer to the value itself, which is stored in one or more regions.

The Kit uses unboxed representation for integers, booleans, words, the unit value, and characters. The Kit uses boxed representation for pairs, records (with at least one element), reals, exception values, function closures, and constructed values (i.e., data types, except lists and booleans).

A boxed value may reside in a finite or an infinite region. Unboxed values are not stored in regions, except when they are part of a boxed value. For example, the integer `3` by itself is stored as the (binary representation) of the value 3 in a KAM register. However, the pair `(3,4)` is represented as a pointer to two consequtive words in a region, the first of which contains the binary representation of 3 and the second of which contains the binary representation of 4.

2.6 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 into HP PA-RISC assembly language. The intermediate languages that 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 that 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 rewrite results in faster programs (as long as the rewrite 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 was not 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.

2.7 Runtime System

The runtime system is written in C. It is small (less than 50Kb 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 for use by the SML Basis Library.

It is possible to call C functions from ML Kit code. The Kit takes care of the memory allocation, by allocating regions for the result of the call before the call and de-allocating the regions at some point after the call. The C functions can build ML data structures such as lists through abstract operations provided by the Kit runtime system. See Chapter 18 for further details.

2.8 Running the Kit

The Kit is a batch compiler. Thus, executing a program consists of first compiling the program and then running the generated target program. Because the Kit stores files in the directories where your source files are located, you should make a personal copy of these directories. Therefore, before you try any of the examples below, make a personal copy of the `kitdemo` directory, which is part of the distribution, and run the kit on your own copy.

The Kit provides two mechanisms for compiling programs. The first mechanism allows you to compile a single SML source file whereas the second mechanism allows you to compile and maintain projects, which are collections of SML source files. To compile a single SML source file or 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`.[2] Compiling a project is very similar to compiling a single SML source file, however, we shall postpone the in-depth discussion of how to compile projects to Chapter 15.

2.8.1 Compiling an SML Source File

As a concrete example, we show how some of the region-annotated programs in Chapter 3 came about.

To compile the SML source file `projection.sml` of Example 3.2, place yourself in your own copy of the `kitdemo` directory and start the Kit with the shell command `kit`.

After the Kit has uttered various greetings, you will find yourself in a rudimentary menu-driven dialogue, see Figure 2.1. 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 Section 3.2 were obtained). Choose `Printing of intermediate forms` (i.e., type `1` followed by carriage 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.

Now, choose `Compile an sml file`; then type `"projection.sml"` (including the quotes) followed by return. The Kit outputs (among other things) the MulExp program shown in Section 3.2.

Go up one on the menu tree. Printing of the region-annotated types can now be enabled by selecting `Layout` from the main menu, and then `print types`. Thereafter, go back to the top-most menu and choose `Compile it again` to compile the source file `projection.sml` again. This time, the Kit outputs the MulExp program shown in Section 3.3.

Next, you can try the example in Section 3.4; select `Compile an sml file` from the top-most menu and enter `"elimpair.sml"`.

[2]The `readme` file in the distribution tells you how to install the Kit.

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

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

>
```

Figure 2.1: The top-most Kit menu

2.8.2 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 the working directory.

Running the target program is done from the UNIX shell by typing

```
run
```

The file will probably be around 350Kb large, even for the trivial examples considered in this chapter. This is because it contains the Kit runtime system and compiled code for the parts of the SML Basis Library that are needed for linking.

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

Part II

The Language Constructs of SML

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 SML 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, because the empty tuple is not allocated; it is just a constant that 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: _|}
end
```

There are several things to notice from this example.

1. The MulExp program contains a free region variable, `r1`. Notice that the construction of the pair `xy` has been annotated by "`at r1`", indicating where the pair should be put;

2. The expression `{|xy: (_,r1), x: _|}` is an example of a *frame expression.* A frame enumerates the components that are exported from a compilation unit. A frame is similar to a record, except that its components are variables, each annotated with a type scheme and a region variable. (In records, the components can only have types, not general type schemes.) In the example, the type of the frame is `{|xy: (unit*unit, r1), x: unit|}`. The type shows that, after the program unit has been evaluated, `xy` will reside in `r1`. In the the above example, printing of types was suppressed. Thus types were abbreviated to _.

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 with place.* We use μ to range over types with places

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

[1] Program `kitdemo/projection.sml`. Running programs is described in Section 2.8.

where τ is a *region-annotated type*, which again can contain other region-annotated types with places. The region-annotated type with place of an expression is the ML type of the expression decorated with extra region information; every type constructor that represents boxed values (e.g., pairs and strings) is paired with a region variable, indicating where the value is to be put at runtime. Type constructors that represents unboxed values (e.g., integers and booleans) are paired with the region variable ρ_{w}, which denotes a non-existing global region. As an abbreviation, we shall often omit the region variable ρ_{w} from region-annotated types and from region-annotated types with places; and so shall the Kit.

Here are some examples of region-annotated types with places:

$\boxed{\texttt{unit}}$ The type of 0-tuples. Integers, booleans, and 0-tuples are represented unboxed at runtime (rather than being stored in regions), see Section 2.5.

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

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

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

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

3.4 Effects and `letregion`

We now describe the general principle that the Kit uses to decide when it is safe to put `letregion` around an expression.

Here is an example of an SML program that first creates a pair and then selects a component of the pair, after which the pair is garbage:[2]

[2]Program `kitdemo/elimpair.sml`.

```
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: _|}
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.

```
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. The compiler compiles the program as it is, without reducing the conditional to its `then` branch. During evaluation, a region (denoted by `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 with place, 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. The atomic effect $\mathbf{put}(\rho)$ 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 `int` and effect $\{\mathbf{put}(\mathtt{r7}), \mathbf{get}(\mathtt{r7})\}$.

Whenever a region variable occurs free in the effect of an expression but occurs free neither in the region-annotated type with place of the expression nor in the type of any program variable that occurs free in the expression then that region variable denotes a region that is used only 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 with place 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 represented unboxed. A record with n components ($n \geq 1$) is represented boxed, as a pointer to precisely n words in a region. Notice that records are not tagged. Avoiding tags is possible, because (1) there is no pointer tracing garbage collection; and (2) 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 that 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.

Chapter 4

Basic Values

In this chapter we describe how basic values such as integers, reals, strings, and booleans are represented in the Kit. The Kit complies to the Definition of Standard ML (Revised) and to large parts of the Standard ML Basis Library;[1] that is, as a programmer, you can refer to components of the Standard ML Basis Library through the *initial basis*, in which all programs are compiled. Throughout this chapter, we introduce some of the top-level bindings that are provided by the initial basis.

4.1 Integers

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

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

Many other useful operations on integers are available in the `Int` structure.[2]

At runtime, integers are represented without any form of boxing or tagging, so all 32 bits are available.

[1] See the Kit web site for a link to the Standard ML Basis Library.

[2] To see what operations are available in the `Int` structure, consult the file `kit/basislib/INTEGER.sml`.

4.2 Reals

The initial basis provides the following top-level operations on reals:

```
infix  4 < > <= >=
infix  6 + -
infix  7  * /
val ~ : real -> real
val abs: real -> real
val real: int -> real
val trunc : real -> int
val floor : real -> int
val ceil : real -> int
val round : real -> int
```

Values of type `real` are implemented as 64-bit floating point numbers. They are always boxed, that is, 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 this reason, regions with runtime type `real` (see 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.

The structures `Real` and `Math` provide other useful operations on reals.

4.3 Characters and Strings

The initial basis provides the following top-level operations on characters and strings:

```
infix  4 =
infix  6 ^
val ord: char -> int
val chr: int -> char
val str: char -> string
val size: string -> int
val explode: string -> char list
val implode: char list -> string
```

```
val ^ : string * string -> string
val concat: string list -> string
val substring: string * int * int -> string
```

Characters are represented as 32-bit words, although only 8 bits are used to store the character. Just like integers, characters are unboxed and untagged.

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 that are linked together. The internal string representation is completely transparent to the programmer, who does not have to worry about the actual size of region pages. Characters of a string takes up only 8 bits of memory each.

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

The string and character operations can raise exceptions, as detailed in the Standard ML Basis Library documentation.

The structures `Char` and `String` provide other useful operations on characters and strings.

4.4 Booleans

The boolean values `true` and `false` are represented as 32-bit words, although only one bit is used to denote the value. Booleans are unboxed. The initial basis provides the following top-level operations on booleans:

```
infix 4 =
val not: bool -> bool
```

The structure `Bool` provides other useful operations on booleans.

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 5.3 introduces region-annotated list types and show how they correspond to the layout of lists in memory. Section 5.4 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 $\texttt{[}exp_1\texttt{,}\cdots\texttt{,}exp_n\texttt{]}$ 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

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

Notice that `::` is always applied to a pair. The construction of the pair and the application of `::` should, in principle, not be confused: the pair and the constructed value are in principle separate values inasmuch as they have different type. 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

```
(op ::(x, op ::(y, _)))
```

which combines decomposition of constructed values with decomposition of pairs.

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

Lambda, RegionExp, or MulExp	
`nil`	create `nil` value
`::` (e)	create `::` (cons) value
`decon_::` (e)	cons decomposition

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

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

where e must have list type.

5.2 Physical Representation

The empty list is represented by an odd, unboxed integer. A non-empty list is represented as a pointer to a pair of two words in a region, the first of which

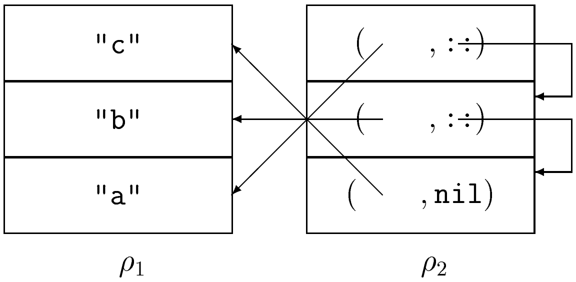


Figure 5.1: Layout of the list `["a","b","c"]` : $((\texttt{string}, \rho_1), [\rho_2])\texttt{list}$ in memory. The auxiliary pairs of the list reside in ρ_2. Each auxiliary pair takes up two words; the constructors `::` (cons) and `nil` are represented unboxed.

contains the head of the list and the second of which contains the representation of the tail of the list. In other words, the physical representation does not distinguish a `::` cell from the auxiliary pair to which `::` is applied. Since `nil` is represented by an odd number and since word addresses are always even, `nil` can be distinguished from the representation of a non-empty list.

As a consequence, there is no cost involved in applying `::` to an auxiliary pair or in applying the decomposition operator `decon_::` to a non-empty list.

5.3 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 element values in the same list must reside in the same region(s) and that all auxiliary pairs of the same list 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 all the auxiliary pairs of a 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 of a list takes the form

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

where μ is the region-annotated type with place of the members of the list and where ρ is the region where the auxiliary pairs of the list are stored. For example, the region-annotated type

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

classifies lists that have their auxiliary pairs in a region ρ_2 and strings in a region ρ_1.

Note that the `list` type constructor is not paired with a region variable. The reason is that the physical representation of lists treats the constructors as unboxed in the sense described in Section 5.2.

Very importantly, not all lists need to live in the same regions. Formally, `nil` and `::` have the following region-annotated type schemes:

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

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 regions (signified by the bound region variables ρ_1 and ρ_2). The ϵ is a so-called *effect variable*. The $\epsilon.\emptyset$ 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 empty, as only unboxed values are manipulated by `::`. 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 a list can, potentially, be in its own regions. But notice that the type of `::` forces the element, which is consed onto the list, to be in the same regions as the already existing elements of the list. Similarly, the type forces the auxiliary pairs to be in one region (ρ_2).

5.4 Example: Basic List Operations

The Kit compiles the program[1]

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

into the RegionExp program shown in Figure 5.2.

[1]Program `kitdemo/onetwothree.sml`.

```
let val it =
        letregion r7:INF
        in let val l =
                    let val v40150 =
                            (1,
                             let val v40151 =
                                     (2,
                                      let val v40152 =
                                              (3, nil) at r7
                                      in :: v40152
                                      end
                                     ) at r7
                             in  :: v40151
                             end
                            ) at r7
                    in  :: v40150
                    end
           in  (case l
                   of :: => #0 decon_:: l
                    | _ => raise Bind
               ) (*case*)
           end
        end (*r7:INF*)
in  {|it: _|}
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 region `r7` holds the auxiliary pairs of the list.

Chapter 6

First-Order Functions

In this chapter, we shall treat functions that are declared with `fun` and that are first-order (i.e., that 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 that the Kit offers to this problem is *region-polymorphic functions*, that is, functions that are passed regions at runtime.

When one declares a function that, when called, produces a fresh list, then 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 that tell 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. Even so, it is useful to understand the general principle, so that one can good use of region polymorphism.

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

$$\texttt{fun}\ f\ \texttt{at}\,\rho_0\ \texttt{[}\rho_1\texttt{,}\ \cdots\texttt{,}\ \rho_k\texttt{]}\ x\ \texttt{=}\ e$$

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. Notice that region parameters are enclosed in brackets (`[ ]`); this should not cause confusion with ML lists, because RegionExp and MulExp do not use brackets for lists..

In the special case $k' = 0$ the record for actual region parameters is empty and is therefore not allocated. We therefore omit printing the "`at` ρ_0" in that case.

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

For an example, consider the following program:

```
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. Notice that `r7` is a formal region parameter of `fromto`. In the last call of `fromto`, a record holding a region descriptor for `r19` is passed to `fromto`; the region record is stored in `r20`. Notice that the regions that hold the two lists generated by this program are distinct. The list that escapes to top level is stored in the global region `r1`, whereas the list that does not escape is stored in the local region `r19`.

6.2 Region-Annotated Type Schemes

A *(region-annotated) type scheme* takes the form

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

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 5.3 are examples of region-annotated type schemes.

```
let fun fromto at r1 [r7:INF] (v40148)=
        let val b = #1 v40148; val a = #0 v40148
        in  (case a > b
               of true => nil
               |  _ =>
                  let val v40153 =
                          (a, letregion r12:1, r14:1
                              in fromto[r7] at r12
                                    (a + 1, b) at r14
                              end (*r12:1, r14:1*)
                          ) at r7
                  in  :: v40153
                  end
            ) (*case*)
        end ;
    val l =
        let val v40175 =
                letregion r16:1, r18:1
                in fromto[r1] at r16 (1, 10) at r18
                end (*r16:1, r18:1*);
            val _not_used =
                letregion r19:INF
                in let val v40176 =
                           letregion r20:1, r22:1
                           in fromto[r19] at r20
                                 (100, 110) at r22
                           end (*r20:1, r22:1*)
                   in  ()
                   end
                end (*r19:INF*)
        in  v40175
        end
in  {|l: _, fromto: (_,r1)|}
end
```

Figure 6.1: The region-annotated version of `fromto` shows that `fromto` is region-polymorphic. (Program: `kitdemo/fromto.sml`, printed by selecting `print drop regions expression` from the `Printing of intermediate forms` menu and then selecting `Compile an sml file` from the main menu.)

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-annotated type schemes that 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-annotated 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-annotated type scheme of `fromto` from Figure 6.1 is

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

At the last call of `fromto` in Figure 6.1, the type scheme is instantiated to the region-annotated type

$$(\texttt{int} * \texttt{int}, \rho_{22}) \xrightarrow{\epsilon'.\{\mathbf{get}(\rho_{22}),\mathbf{put}(\rho_{19})\}} (\texttt{int}, [\rho_{19}])\texttt{list}$$

The instantiation of bound variables of the type scheme that yields this region-annotated type is

$$\{\rho_7 \mapsto \rho_{19}, \rho_8 \mapsto \rho_{22}, \epsilon \mapsto \epsilon'\}$$

In general, the actual region parameters that annotate 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 introduces only 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 what region the value resides in, whereas storing a value in a region uses the name (see Section 2.1) of the region. This omitting of region parameters explains why ρ_8 does not become a formal region parameter of `fromto` and why ρ_{22} 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-annotated) type scheme with place*, that is, a pair (σ, ρ). Indeed, every binding of a variable to a boxed value (whether the binding is done by `fun`, `let`, or `fn`) associates a region-annotated type scheme with place to 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-annotated type (scheme) with place" of some variable, we mean the region-annotated type (scheme) with place that 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.

The region-annotated type scheme with place of a variable bound to an unboxed value is always on the form $(\sigma, \rho_{\mathrm{w}})$, where σ is the region-annotated type scheme associated with the variable and where ρ_{w} denotes a non-existent global region (see Section 3.3). In the following, we shall often abbreviate the region-annotated type scheme with place of a variable bound to an unboxed value by its region-annotated type scheme.

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 that 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 that are determined by the argument to the function. A good example is the list append function from the initial basis:

```
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, the auxiliary pairs of `xs` and `ys` need not be in the same regions, although the elements of `xs` and `ys` clearly must be in the same regions, because they end up in the same list. These properties of append are summarised in the inferred region-annotated type scheme:

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

When one writes a function it is a good idea to consider whether one wants the function to create values in fresh regions or whether one wants it to add values to existing regions. Adding to existing regions can of course make these regions too large and long-lived, because 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 the auxiliary pairs of a list into a fresh region, whereas `cp2` always copies the auxiliary pairs of a list into the same region:

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

As we saw in Section 1.3, there are cases where it is useful to copy a list from one region into another region, so as 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.

Because 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*. Notice that this is not a clear-cut distinction, however. Often, functions have both an endomorphic and an exomorphic side to them. Also notice that even a region exomorphic function can be forced to act as an endomorphism by the calling context. As an example, consider the expression

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

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

6.4 Polymorphic Recursion

A recursive region-polymorphic function

$$\texttt{fun}\ f\ \texttt{at}\,\rho_0\ \texttt{[}\rho_1\texttt{,}\ \cdots\texttt{,}\ \rho_k\texttt{]}\ x\ \texttt{=}\ e$$

may call itself inside its own body (e) with regions that are different from its own formal region parameter ($\texttt{[}\rho_1\texttt{,}\ \cdots\texttt{,}\ \rho_k\texttt{]}$). 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 memory management in connection with recursion. Unfortunately, it is also makes the region inference problem considerably more challenging, but that is a different story [TB98].

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 that 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 and so on. The space consumption 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, that is, 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, at the rightmost merge of two lists of length at most one, approximately $4n$ list cells are live. Then a series of final merges occur. Code that uses these ideas is listed in Figure 6.2. [1] The

[1] Project `kitdemo/msort.pm`, file `kitdemo/msort.sml`. To compile the project, enter the `Project` sub-menu (from within the `kitdemo` directory) and choose `Set project`

```
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;
```

Figure 6.2: Merge sorting of lists.

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 a region 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`.

`file name`. Then type `"msort.pm"` (including the quotes) followed by return. Finally, choose `Compile and link project`. The Kit places an executable file `run` in the `kitdemo` directory. For an in-depth description of how to compile and run projects, see Chapter 15.

Based on the above analysis we conclude that the space required by `msort xs` is approximately $4nc_1 + c_2\log_2 n$ plus the extra stack space required for the final merges, where n is the length of `xs`, c_1 is the size of a list cell (2 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).

Because `merge` is not tail-recursive, a merge requires space both for its two input lists, for its output list, for finite regions on the stack and for temporaries stored on the stack. More precisely, each recursive call of `merge` allocates two words for the argument to the recursive call in a region on the stack; it also allocates a region closure of size one word, holding the actual region parameter. When one of the lists becomes empty, `merge` calls `cp`, which allocates less for each iterative call than `merge` does. Each return from `merge` allocates a list cell (two words) but deallocates the two regions (of sizes two and one words, respectively) on the stack, so the maximum space usage is reached when the last element of the result of the merge is constructed (which happens when the recursion is deepest). Here the space used is (we show $n = 50,000$ list elements as an example)

data	**size (words)**	$n = 50,000$
input list	$2n$	400,000 bytes
l	n	200,000 bytes
l sorted	n	200,000 bytes
r	n	200,000 bytes
r sorted	n	200,000 bytes
finite regions on stack	$3n$	600,000 bytes
total in regions	$9n$	1,800,000 bytes

It turns out that each iterative call of `merge` pushes three registers on the stack, so that stack size (not including space for finite regions) will be approximately $3 \times 4 \times n$ bytes, which for $n = 50,000$ is 600,000 bytes. The total space consumption for sorting 50,000 integeres should therefore be roughly 2,400,000 bytes.

To check the above analysis, we sorted 50,000 integers with the region profiler enabled. As one sees in Figures 6.3 and 6.4, the space usage found by region profile correspond well to the results of our analysis.

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

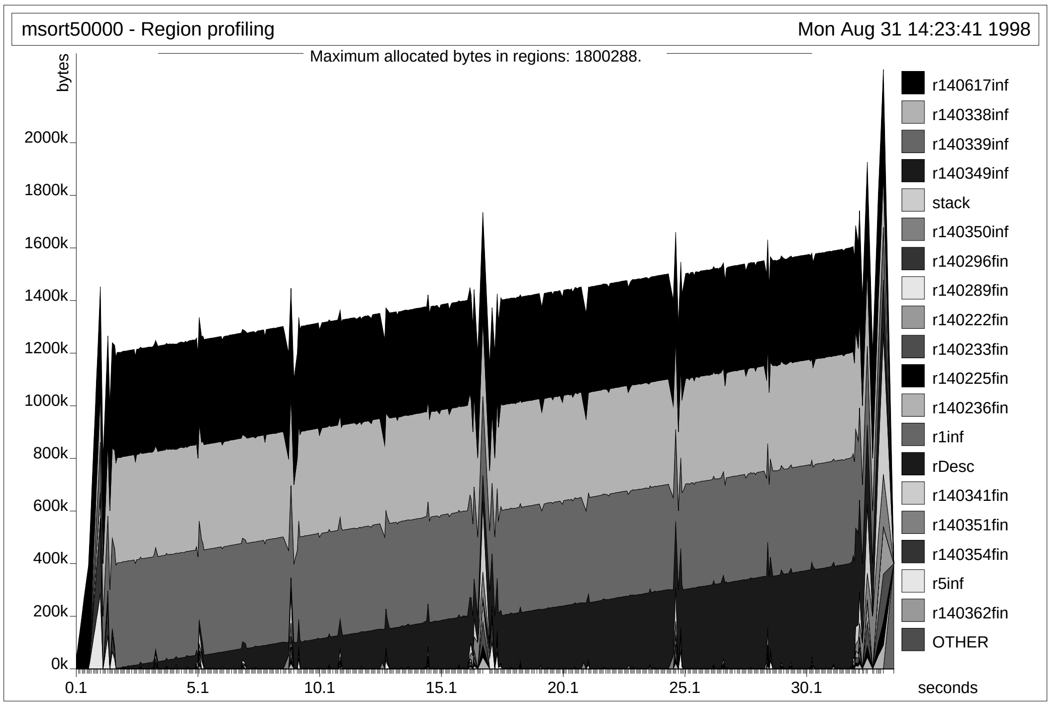


Figure 6.3: Region profiling of `msort` sorting 50,000 integers. The high-level mark of 1,800,288 bytes is exact (i.e., not sampled).

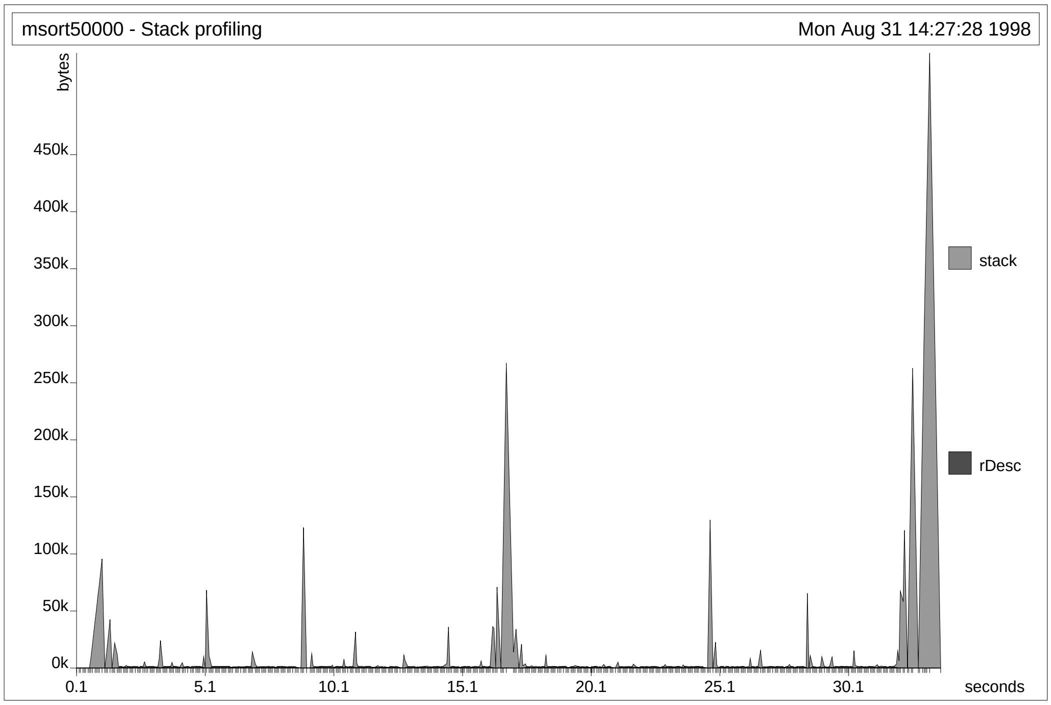


Figure 6.4: Stack profiling of `msort` sorting 50,000 integers.

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 so as 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 $\{\mathtt{x} \mapsto 7\}$. 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 then uses these declarations to perform the declarations 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, because 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 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 that has ρ free in its region-annoated type scheme with 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]

For a more interesting example of the consequences of the Ground Rule, consider the following declarations, taken from a program that 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; because `rest` is in scope at the recursive application of `sieve`, the list that 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
```

That the copying really overwrites the input list relies, in part, on region resetting (Chapter 12). But it also relies on region polymorphism and on the Ground Rule. Rewriting the applicationn of `sieve` 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. Because `cp` is also exomorphic, there will be no sharing between `rest` and the other lists. The region variable that denotes the region that holds the the auxiliary pairs of `rest` appears in the effect of the (revised) `let` expression. However, this region variable does not occur free in the region-annotated type scheme with place of any value variable in scope at that point, not even in the region-annotated type scheme with place of `sieve`, which only has the region that contains `sieve` itself free in its region-annotated type scheme with place. Consequently, region inference wraps the `let` expression by a `letregion` binding of the region variable in question:

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

7.3 Shortening Lifetime

Informally, region inference forces the lifetime of an identifier to be at least its scope. Improving memory performance therefore sometimes requires making scopes of identifiers smaller. Useful program transformations include:

1. **Inwards let floating**: transform

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

$$\texttt{let val } id_2 \texttt{ = let val } id_1 \texttt{ = } exp_1 \texttt{ in } exp_2 \texttt{ end in } exp \texttt{ end}$$

provided id_1 does not occur free in *exp*.

2. **Application extrusion**: transform

$$\texttt{let } dec \texttt{ in } f\texttt{(}exp\texttt{) end}$$

into

$$f \texttt{ let } dec \texttt{ in } exp \texttt{ end}$$

provided f is an identifier that is not declared by *dec*.

Application extrusion is a particularly useful 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-annotated type scheme σ has a put effect on a region variable that 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 that 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 the expression `y>3` evaluates to `true`, an append operation is performed that produces a list in the same region as `x`. 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 that is allocated at the beginning of the body of `g`. The sequence of calls to `f` accumulates copies of `x@x` in that region, although none of these lists are accessible anywhere.

[1] Program `kitdemo/escape.sml`.

In 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-annotated type scheme inferred for `f` is

$$\forall\epsilon.\mathtt{int} \xrightarrow{\epsilon.\{\mathbf{put}(\mathtt{r5})\}} \mathtt{int}$$

where the region-annotated type of `x` is

$$(\mathtt{int}, [\mathtt{r5}])\mathtt{list}$$

Here we see that `r5` is free in the region-annotated type scheme and appears with a put effect.

8.1 Warnings About Space Leaks

The Kit issues a warning each time it meets a function that is declared using `fun` and has a free put effect occurring somewhere in its type scheme. In practice, this warning mechanism is a valuable device for predicting space leaks. The region-annotated version of our example function `g` is listed in Figure 8.1. During compilation of `g`, the Kit issues the following warning:[2]

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

We are told that the program might space leak in region `r8`. Looking at the function `f`, we see that this region is an actual region parameter to `@`. It follows 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 value that causes the space leak. In the above example, we can move the construction of the list into `f`:[3]

[2]To provoke the warning, one has to disable inlining in the Lambda optimiser; this is done by setting the `Maximum inline size`, found in the `Control/Optimiser` sub-menu, to `0`.

[3]Program `kitdemo/escape1.sml`.

```
fun g at r1 [] (v40146)=
  letregion r8:INF
  in let val x =
          let val v40160 =
                (5, let val v40161 = (7, nil) at r8
                    in :: v40161
                    end
                ) at r8
          in  :: v40160
          end
     in letregion r11:1
        in let fun f at r11 [] (y)=
                let val _not_used =
                      let val v40153 =
                            (case y > 3
                               of true =>
                                 letregion r14:1, r16:1
                                 in @[r8] at r14 (x, x) at r16
                                 end (*r14:1, r16:1*)
                                |  _ => x
                            ) (*case*)
                      in  ()
                      end
                in  5
                end ;
              val _not_used =
                let val v40159 = letregion r17:1
                                 in f[] 1
                                 end (*r17:1*)
                in ()
                end
          in letregion r19:1 in f[] 4 end (*r19:1*)
          end
        end (*r11:1*)
     end
  end (*r8:INF*)
```

Figure 8.1: The region-annotated version of g.

```
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 reconstructed 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:[4]

```
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 quantification 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 as `f` can be called).

At other times, there is no clean way of avoiding escaping put effects. One example is found in the `TextIO` structure of the Basis Library:

```
exception CannotOpen
fun raiseIo fcn nam exn =
  raise IO.Io {function = fcn^"", name = nam^"", cause = exn}
```

[4]Program `kitdemo/escape2.sml`.

```
fun openIn (f: string) : instream =
  {ic=prim("openInStream","openInStream",(f,CannotOpen)),
   name=f} handle exn => raiseIo "openIn" f exn
fun openOut(f: string): outstream =
  {oc=prim("openOutStream","openOutStream",(f,CannotOpen)),
   name=f} handle exn => raiseIo "openOut" f exn
```

As explained in Chapter 11, when a unary exception constructor is applied to a value, both the argument value and the resulting constructed value are forced into a particular global region. Thus, the application

```
IO.Io {function = fcn^"", name = nam^"", cause = exn}
```

has a potential space leak in it; every time we apply the exception constructor, the resulting exception value will be put into a global region. This particular space leak is perhaps not something that would keep one awake at night, because 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. Notice, however, that the string arguments to `raiseIo` are copied inside the body of `raiseIo`, so that they are not forced to be placed in the global string region.

Chapter 9

References

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

9.1 References in Standard ML

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

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

If the type of a reference r is $\tau\,\texttt{ref}$ 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 `val` declaration. While the value stored at a reference may change, the binding between variable and reference does not change. We show an example, because 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;
```

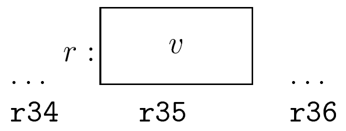


Figure 9.1: Creating a reference allocates one word in a region on the region stack. Here, the region is drawn as a finite region, but it could equally well be infinite.

```
        !x
    end
```

Because `!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. Because `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 (assuming *exp* translates into e)

$$\texttt{ref at}\,\rho\;\;e$$

which is evaluated as follows. First e is evaluated. Assume that this evaluation 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.

The situation is depicted in Figure 9.1. The value v can be unboxed as shown in Figure 9.2. Or it may be boxed, in which case v is an address.

Notice that a reference really is a pointer in the implementation. In particular, a reference is not tagged, so it may be stored in a KAM register. The contents of the reference is also always 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. Notice that dereferencing does not require knowledge of what region the word with address r resides in.

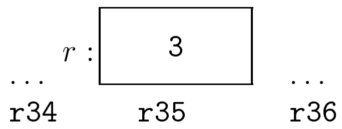


Figure 9.2: Creating a reference allocates one word in a region on the region stack. Here, the region is drawn as a finite region, but it could equally well be infinite.

Assigning a value v to a reference r simply stores v in the memory at address r. When v is an unboxed value, the assignment 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 is:

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

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 region-annotated type with place of the contents of that word. For example, assume ρ is bound to some region name, say `r35`; then the evaluation of the declaration `val x = ref at` ρ `3` results in the environment $\{\mathtt{x} \mapsto r\}$, where r is the address of a word with contents 3 residing in region `r35`, see Figure 9.2. The type of `x` is `((int,`ρ_{w}`) ref,` ρ`)`, which, as usual, we shorten to `(int ref,` ρ`)`.

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

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

Although the type scheme for `:=` has in it a put effect on the region holding the reference, assignment does not actually allocate any values in this region.

Instead, it manipulates an already existing value in the region. 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, then they are forced to be in the same regions (cf. the region-annotated type schemes given above).

If we compile the example from Section 9.1, we get the program shown in Figure 9.3.[1] The region denoted by `r7` contains the memory word whose 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 notice that the word containing 5 is designated `r7`, because the `then` and `else` branches must be given the same region-annotated type with place. Finally, notice that all references will be reclaimed automatically at the end of the `letregion` constructs that bind `r7` and `r8`.

9.4 Local References

References that are created locally within a function and that do not escape the function naturally reside in regions that 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 (*r10:1*)
           end
        end (*r9:1*)
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

[1]Program `kitdemo/refs3.sml`.
[2]Program `kitdemo/refs1.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 v40143 = true
                          in ref at r8 v40143 end ;
                      val z =
                          (case letregion r9:1
                                in ![] y
                                end (*r9:1*)
                             of true => x
                              | _ => ref at r7 5
                          ) (*case*) ;
                      val v40137 =
                          letregion r11:1, r13:1
                          in :=[r7] at r11 (z, 6) at r13
                          end (*r11:1, r13:1*)
                  in  letregion r14:1 in ![] x end (*r14:1*)
                  end
               end (*r8:1*)
           end
        end (*r7:INF*)
in  {|it: _|}
end
```

Figure 9.3: Region-annotated reference creation.

(`standardArg`) to that word on the stack. At the end of the `letregion r9 in` $\cdots$ `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)
end
val y = memo_id "abc"
val z = memo_id "efg";
```

Provided that inlining by the optimiser is restricted to inline only those functions that are applied once,[4] this example compiles into

```
let val r = let val v40271 = nil in ref at r1 v40271 end ;
    fun memo_id at r1 [] (x)=
        let val v40267 =
                letregion r8:1, r10:1
                in :=[r1] at r8
                   (r,
                    let val v40268 = (x, letregion r12:1
                                         in ![] r
                                         end (*r12:1*)
                                     ) at r1
                    in  :: v40268
                    end
                   ) at r10
                end (*r8:1, r10:1*)
        in  x
        end ;
    val y = letregion r14:1
            in memo_id[] "abc"at r4
            end (*r14:1*);
```

[3]Program `kitdemo/refs2.sml`.

[4]To restrict the optimiser accordingly, set the menu entry `Control/Compiler/maximum inline size` to `0`.

```
       val z = letregion r16:1
               in memo_id[] "efg"at r4
               end (*r16:1*)
   in  {|z: (_,r4), y: (_,r4),
         memo_id: (_,r1), r: (_,r1)
       |}
   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 with places of :
  less_int minus_int := ! r Div Mod 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 that this region with all its values will be alive for as long as the reference is live. If the contents type is unboxed (e.g., `int`), there is no problem, for in that case, no region for the contents is allocated. But one should avoid creating long-lived references that are assigned many different large values.

Chapter 10

Recursive Data Types

This chapter describes how the Kit treats recursive data types. We have already seen how one recursive datatype, namely lists, is handled. This chapter deals with the general case.

10.1 Spreading Data Types

The Kit performs an analysis called "spreading of data types". Spreading of datatypes analyses `datatype` declarations. This analysis of a `datatype` declaration uses information about the type constructors that appear in the types of the constructors of the data type(s) introduced by the declaration, but it does not use information about the use of the data type.

Spreading determines (a) a so-called arity of every type name that the data type declaration introduces and (b) a region-annotated type scheme for every value constructor introduced by the data type declaration.

In the Definition of Standard ML every type name has an attribute, called its arity [MTHM97]. The arity of a type name is the number of type arguments it requires. For example, `int` has arity 0 while the type name introduced by the following declaration of binary trees has arity 1:

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

The Kit extends the notion of arity (in it's internal languages) to account for regions and effects. For lists, for example, we need 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). Region variables that are induced by the types of constructors and that 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 that describes where the pairs of type `'a * 'a list` (i.e., the auxiliary pairs), reside.

Besides auxiliary regions, one sometimes needs auxiliary effects. For an example, consider:

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

Here one needs 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 arities 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).

Because of the limit on the number of auxiliary region variables, spreading of data type declarations sometimes unifies 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 applying a constructor to a value v sometimes forces identification of regions of v that hold otherwise unrelated parts of v.

The automatic memory management that 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 except lists (Chapter 5) are always boxed.

10.2 Example: Balanced Trees

Consider the program in Figure 10.1.[1] 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 r75:1, r77:INF
    in print[]
       letregion r78:1, r80:INF
       in implode[r77] at r78
          letregion r81:1, r83:1, r84:INF, r85:INF
          in preord[r80] at r81
             (letregion r86:1, r88:INF
              in balpre[r84,r85] at r86
                 letregion r89:1, r91:1
                 in explode[r88] at r89
                    "Greetings from the Kit\n"at r91
                 end (*r89:1, r91:1*)
              end (*r86:1, r88:INF*),
              nil
             ) at r83
          end (*r81:1, r83:1, r84:INF, r85:INF*)
```

[1]Project: `kitdemo/trees.pm`, 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 = print(implode(preord(balpre(explode
    "Greetings from the Kit\n"),[])));
```

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

```
          end (*r78:1, r80:INF*)
      end (*r75:1, r77:INF*)
```

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.

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 Standard 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 top level. The memory 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 a *nullary* exception value or a *constructed* exception value. A nullary exception value is a pointer to a word that points to an exception name. A constructed exception value is a pair (en, v) of an exception name en and a value v; we refer to v as the *argument* of en. This representation of exception values allows for the exception name of an exception value to be fetched in the same way irrespective of whether the exception value is nullary or constructed.

Referring to a nullary exception constructor allocates no memory. By contrast, applying a unary exception constructor to an argument constructs a constructed exception value. The memory cost of such an application is two words for holding the pair (en, v).

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 nullary exception values are put into a certain global region and thus never reclaimed automatically. A constructed exception value is put in a region that is live at least as long as the exception constructor is in scope.

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 raised 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 that can handle the exception. The KAM has a register that 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 that 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 that is popped before the exception is handled! This is why we put all exception names 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 at ρ *match* in e_1 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.

Handling of exceptions is the only operation that takes time that 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];
```

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] Program `kitdemo/exceptions.sml`.

Chapter 12

Resetting Regions

The idea of region resetting was introduced in Section 1.2.

This chapter gives an informal explanation of the rules that govern resetting. Knowing these rules is useful, 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. Resetting a region is a constant-time operation. Since the same region variable can be bound sometimes to a finite region and sometimes to an infinite region a runtime, resetting a region can involve a test at runtime.

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

1. inserting automatic resetting of infinite regions, when possible;

2. checking applications of `resetRegions` (and `forceResetting`) so as 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 to indicate 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, so as 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 what 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 transforms it into `atbot` ρ if it can prove that the region will contain no 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 at bottom 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` creates the empty list, it can potentially reset 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 region as `l1`. Because `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]

```
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` creates the empty list, it is welcome to reset the region that holds `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 used.) 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).

Because the Kit allows for 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 that has ρ as a formal region parameter, one cannot know at compile time whether to use `attop` or `atbot` as storage mode. Instead, the storage mode analysis operates with a third kind of storage mode named `sat`, read: "somewhere at". Consider an application of f for 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 point `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]

---

[1] Program `kitdemo/sma1.sml`.

[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.

The relevant parts of the result of compiling the last example are shown in Figure 12.1. To see the storage modes, switch on the flag

```
print drop regions expression with storage modes
```

in the menu `Printing of intermediate forms`.

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, because 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 `r18` with storage mode `atbot` because 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 does not reside in `r18`.

Within every lambda abstraction, the Kit performs a backwards flow analysis that determines, for every allocation point, a set of *locally live variables*, that is, 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 that 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 that 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 those allocation points that do not occur inside functions.

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

$$\mathtt{at}\,\rho \qquad (12.1)$$

[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.

```
fun f attop r1 [r7:INF] (var553)=
    (case var553
       of nil => nil
       |  _ => let val xs = #1 decon_:: var553;
                   val x = #0 decon_:: var553;
                   val v41096 =
                       (x + 1,
                        letregion r14:1
                        in f[sat r7] atbot r14 xs
                        end (*r14:1*)
                       ) attop r7
          in  :: v41096
          end
    ) (*case*) ;
val n =
    letregion r16:1, r18:INF
    in length[]
       let val l1 =
               let val v41103 =
                       (1,
                        let val v41104 =
                                (2,
                                 let val v41105 =
                                         (3, nil) attop r18
                                 in :: v41105
                                 end
                                ) attop r18
                        in  :: v41104
                        end
                       ) attop r18
               in  :: v41103
               end
       in  (case true
              of true => letregion r22:1
(*1*)                    in f[atbot r18] atbot r22 l1
                         end (*r22:1*)
              |  _ => l1
           ) (*case*)
       end
    end (*r16:1, r18:INF*)
```

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

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

```
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, that is, inside an expression of the form `fn` *pat* `=>` e. Here we regard every expression of the form

`let fun f(x) =` e `in` e' `end`

as an abbreviation for

`let val rec f = fn(x) =>` e `in` e' `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 that 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-annotated type scheme with place (Section 6.2), and use `attop` otherwise (see Figure 12.3);

```
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 considered in B2. If no locally live variable l has ρ occurring in its region-annotated type scheme with place, replace `at` ρ by `atbot` ρ, otherwise by `attop` ρ.

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-annotated type scheme with place, and use `attop` otherwise (see Figure 12.4).

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 of a variable always contains free in it region variables for all the regions that the value bound to the 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

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

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

point. Thus, if no locally live variable at the allocation point has ρ free in its region-annotated type scheme with 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 that does occur free in the region-annotated type scheme with 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*. To determine whether to use `sat` or `attop` in case (B3), the Kit builds a *region flow graph* for the entire compilation unit. (This construction 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 for 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, so as to cater for applications of f in subsequent compilation units.) 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 that 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 that is locally live at the allocation point and for every region variable ρ' that occurs free in the region-annotated type scheme with 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-annotated type scheme with 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 that 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`. A region profile resulting from running the program is shown in Figure 12.6. The diagram shows how much space is used in regions (both finite and infinite regions) and on the stack. 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.

[5] Program `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.

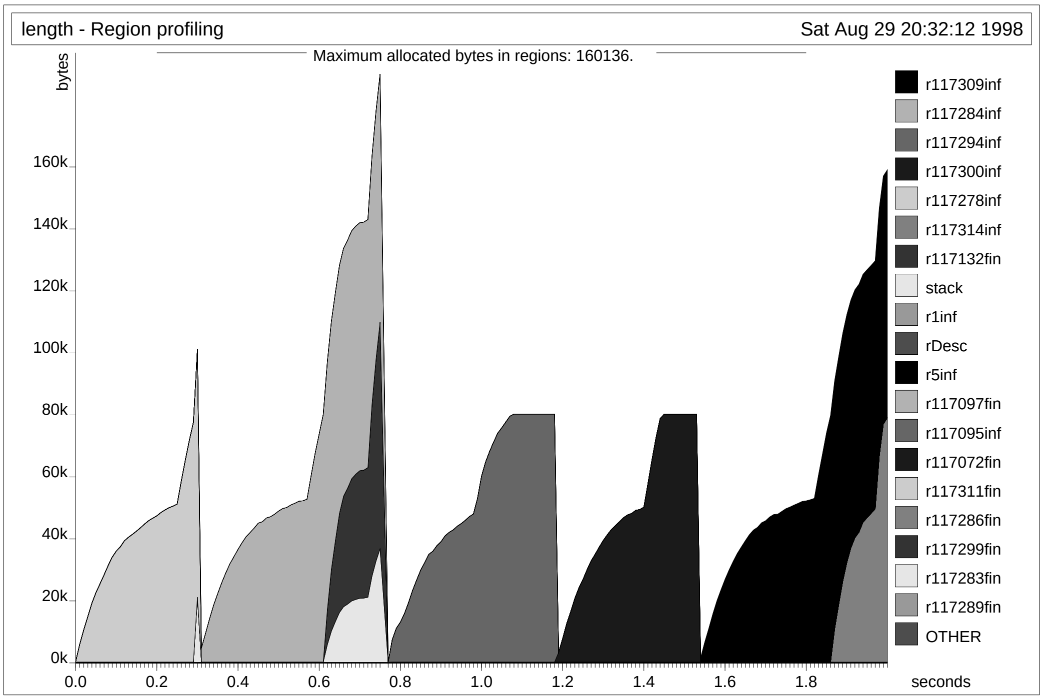


Figure 12.6: Region profiling of five different ways of computing the length of a list, namely, from left to right: `nlength`, `tlength`, `klength`, `llength`, and `glength`.

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 a region; then follows a shorter computation of the length of the list. The space behaviour of the five ways of computing the length vary considerably. We shall have more to say about the time behaviour in what follows.

As one would expect, `nlength` leads to a peak in stack size; 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`! Notice 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, because it is a prototype of a particular schema that 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. Because 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 because `loop` is an endomorphism, `(xs, acc+1)` is forced 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. Because `loop` is a region endomorphism, the result of `loop(l, 0)` will be in the same region as `(l, 0)`. Moreover, because we then immediately take the second projection of that pair, that region is clearly local to the body of `klength`. Call the region ρ. Because 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) of 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! The region that `loop` lies in is clearly different from ρ.

Therefore, every iteration of `loop` resets the infinite region ρ so that it will 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`. 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, because `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 that is initially applied to a freshly constructed pair. This pair can reside in a region that 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 what 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 earlier, `length'` lives in a global region, which is clearly different from the region inside `llength` that 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. Because `global` may be called from a different compilation unit, then, for all we know, `global` may be applied to a pair that 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 that 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 that is declared as local to the function that uses it; we refer (informally) to

[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.	0.38	0.61	0.74	0.69	0.73	0.67

Figure 12.7: User time in seconds for building a list of 1 million 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.

this auxiliary function as the *iterator*.

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

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 1 million 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. Because `upto` alone takes 0.38 seconds to build the list, the differences in times are clear: the versions of the length function that take pairs as arguments are slower than the version that stores values on the stack (i.e., `nlength`); this difference would presumably be reduced significantly if the Kit allowed functions to take argument values in more than one register.

12.4 `resetRegions` and `forceResetting`

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. Therefore, the

[7] When profiling is turned on, every resetting of a region involves resetting of values in the first region page of the region.

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).

Kit provides two operations that the programmer can use to encourage (or force) the Kit to perform resetting at particular places in the program. The two operations are

resetRegions *vid*

and

forceResetting *vid*

In both cases, the argument has to be a value identifier. 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 that occurs free in the region-annotated type scheme with place of *vid*. Let m be the storage mode determined for ρ at a program point according to the rules of the previous section. Whether resetting of *vid* 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 that 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 variant of `msort` listed in Figure 12.9.[8]

Unfortunately, the storage mode analysis complains:

```
 *** Warnings ***
resetRegions(xs):
   You have suggested resetting the regions that appear free
   in the type scheme with place of 'xs', i.e., in
   (int, [r58]) list
   (1)
        'r58': there is a conflict with the locally
        live variable
        l :(int, [r65]) list
        from which the following region variables can be reached
        in the region flow graph:
             {r65}
        Amongst these, 'r65' can also be reached from 'r58'.
        Thus I have given 'r58' storage mode "attop".
```

There is one complaint concerning the first `resetRegions`, but none concerning the two remaining ones. By inspecting the region-annotated term one sees that `r58` is a formal parameter of `msort`. Due to the recursive call `msort l`, the region graph contains an edge from `r58` to `r65`. Thus the analysis decides on `attop`, using rule (B3). This choice 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 decide to put `forceResetting` to work, see Figure 12.10.[9] The region profile of the improved merge sort appears in Figure 12.11. As expected, we have

[8] Project: `kitdemo/msortreset1.pm`, file `kitdemo/msortreset1.sml`.
[9] Project: `kitdemo/msortreset2.pm`, file `kitdemo/msortreset2.sml`.

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

  (* exormorphic 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::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 = print "Really done\n"
end
```

Figure 12.9: Variant of `msort` that uses `resetRegions` to improve memory usage. The Kit fails to infer that the region holding the argument list `xs` can be reset after `xs` is split.

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

  (* exormorphic 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::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 = print "Really done\n"
end
```

Figure 12.10: Using `forceResetting` to reset regions.

now brought space consumption down from four times to two times the size of the input. Figure 12.11 may be compared to Figure 6.3 on page 66.

12.6 Example: Scanning Text Files

In this section we present a program that can scan a sequence of Standard ML source files so as 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 that 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.[10] 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 `TextIO.inputN` 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 that repeatedly reads a file 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

[10] Project: `kitdemo/scan.pm`, file: `kitdemo/scan.sml`.

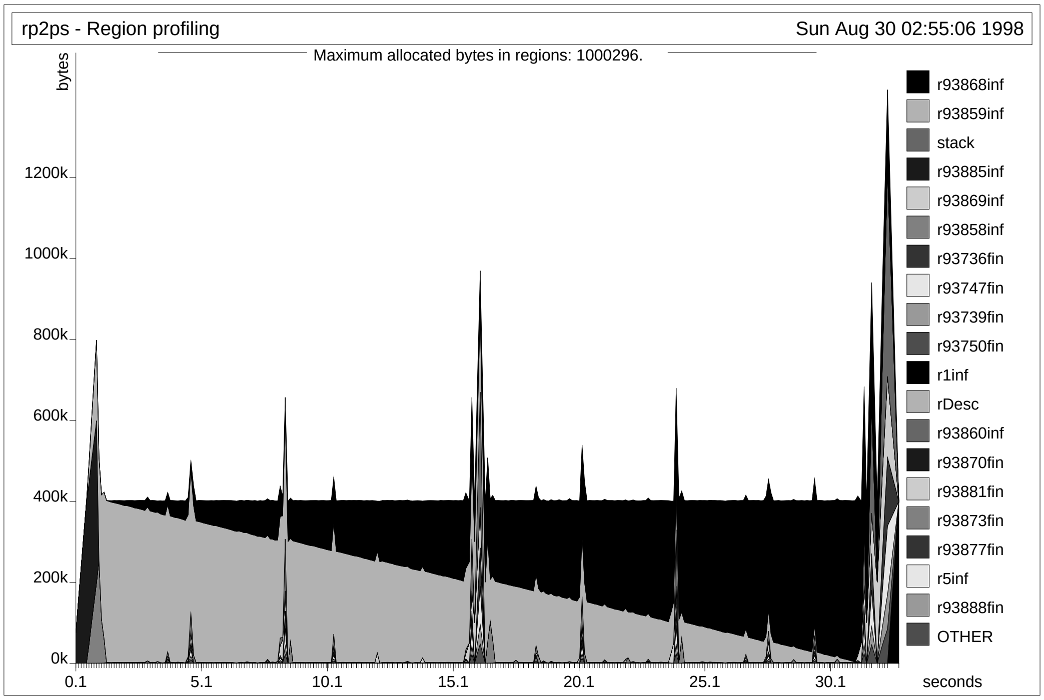


Figure 12.11: Region profiling of the improved mergesort. The lower triangle contains unsorted elements, while the upper triangle contains sorted elements. The program was compiled with profiling enabled and then run with the command `run -microsec 100000`. The PostScript picture `region.ps` was generated with the command `rp2ps -region -sampleMax 1000 -eps 137 mm` and then previewed using the command `ghostview region.ps` .

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, `readWord` will always try to store the string it creates at the bottom of the region in question.

In general however, 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. This extra resetting may be necessary because formal region parameters of exported functions are connected to global regions in the region flow graph (cf., rule B3).

```
local
  exception NotBalanced
  fun scan(is: TextIO.instream) : int*int =
    let
      fun next() = TextIO.inputN(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 *)*)

      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(concat[filename, ": size = ", Int.toString n,
                  " comments: ", Int.toString inside, " (",
                  (Int.toString(percent(inside, n))
                   handle _ => "-"), "%)"])

 (* 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  = TextIO.openIn filename
  in let val (n,inside) = scan is
     in TextIO.closeIn is;
        report_file(filename, n, inside);
        SOME(n,inside)
     end handle NotBalanced =>
         (writeln(filename ^ ": not balanced");
          TextIO.closeIn is;
          NONE)
  end handle IO.Io {name,...}  =>
       (writeln(name^" failed."); NONE)

 fun report_totals(n,inside) =
   writeln(concat["\nTotal sizes: ", Int.toString n,
                  " comments: ", Int.toString inside,
                  " (", (Int.toString(percent(inside,n))
                         handle _ => "-"), "%)"])

 (* 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: TextIO.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') =>
                     (readWord(is), n+n',inside+inside')
                    | NONE =>
                     (readWord(is),n,inside))
  in
    driver(readWord(is),0,0);
    ()
  end
in
  val result = main(TextIO.stdIn)
end
```

The program was compiled both with and without profiling turned on. The output from running the program on 10 of the source files for the Kit is shown here:

```
Parsing/INFIX_STACK.sml: size = 487 comments: 321 (65%)
Parsing/InfixStack.sml: size = 7544 comments: 3025 (40%)
Parsing/Infixing.sml: size = 32262 comments: 5295 (16%)
Parsing/LEX_BASICS.sml: size = 2102 comments: 1257 (59%)
Parsing/LEX_UTILS.sml: size = 1305 comments: 291 (22%)
Parsing/LexBasics.sml: size = 12677 comments: 2967 (23%)
Parsing/LexUtils.sml: size = 7643 comments: 717 (9%)
Parsing/MyBase.sml: size = 33933 comments: 11140 (32%)
Parsing/PARSE.sml: size = 1078 comments: 572 (53%)
Parsing/Parse.sml: size = 7040 comments: 870 (12%)

Total sizes: 106071 comments: 26455 (24%)
```

A region profile for that run is shown in Figure 12.12. The almost-constant space usage is evident. The occasional disturbances are due to the noniterative functions that read a file name from input by first reading one line and then extracting the name.

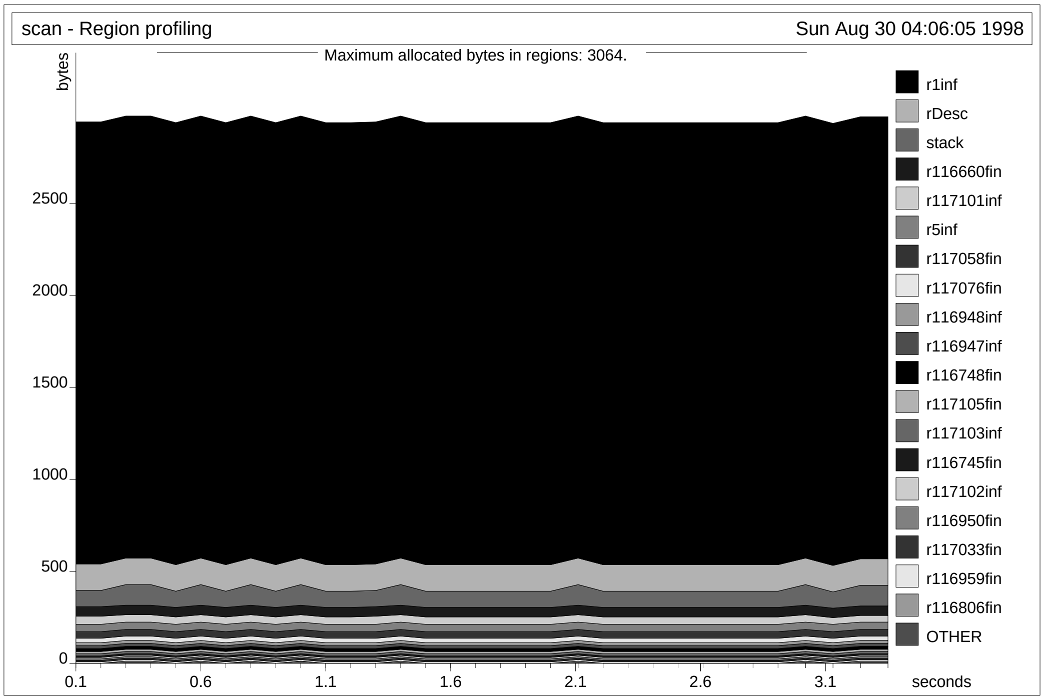


Figure 12.12: Region profile of the scanner. The unit of measure on the y-axis is bytes, not kilobytes. The occasional increase is due to the functions that read a file name from an instream. The program was compiled with profiling enabled, then run with the command `run -microsec 100000 < ../../kitdemo/scanfiles`. A PostScript file `region.ps` can be generated with the command `rp2ps -region -sampleMax 1000 -eps 137 mm`.

Chapter 13

Higher-Order Functions

13.1 Lambda Abstractions (`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 abstractions 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`, for example,

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

declares the value identifier `h` to be the composition of `g` and `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. Functions declared using this abbreviation are said to be *Curried.*

13.2 Region-Annotated Function Types

The general form of a region-annotated function type is

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

where μ_1 is the type with place of the argument, μ_2 is the type with place of the result, and ρ is the region containing the closure for the function. As mentioned in Section 5.3, 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

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

[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] Program `kitdemo/lambda.sml`.

```
let val n =
     letregion r7:1, r9:1, r10:INF
     in let val f =
             let val xs =
                  let val v40299 =
                       (1,
                        let val v40300 = (2, nil) at r10
                        in :: v40300
                        end
                       ) at r10
                  in  :: v40299
                  end
             in fn at r7 ys =>
                 letregion r17:1 in length[] xs end (*r17:1*) +
                 letregion r20:1 in length[] ys end (*r20:1*)
             end
        in  f let val v40294 = (7, nil) at r9 in :: v40294 end
        end
     end (*r7:1, r9:1, r10:INF*)
in  {|n: _|}
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.

Notice that `xs` has to be kept alive for as long as the function `(fn ys => ···)` may be called, for this function will access `xs`, when called. The region-annotated version of the example appears in Figure 13.1.[3] We see that `xs` is put in `r10`, that the function closure for `(fn ys => ···)` is put in `r7` and indeed, `r7` and `r10` have the same lifetime. To understand how the region inference system figured that out, let us consider the effect and the region-annotated types of particular sub-expressions. Looking at the lambda abstraction, it must have a functional type of the form $(\tau \xrightarrow{\epsilon.\varphi} \tau', \texttt{r7})$ where

[3]To see the output programs discussed in this section, enable the flag `print drop regions expression`.

φ is the effect

$$\{\mathbf{get}(\mathtt{r1}), \mathbf{get}(\mathtt{r10}), \mathbf{get}(\mathtt{r9})\}$$

Notice that `r10` occurs free in the type of the lambda abstraction. But, as pointed out in Section 3.4, the criterion for putting a `letregion` binding of ρ around an expression e is that ρ occurs free neither in the type with place of e nor in the type scheme with place of any variable in the domain of the type environment. The smallest sub-expression of the program for which `r10` does not occur free in the type with place of the expression is the right-hand side of the `val` binding of `n`, for that expression simply has type with place `int`. 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 `r10`.

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:[4]

```
fun apply f x = f x
val y = apply (fn n => n + 1.0) 5.0
val z = apply (fn m => m) 6
```

Here is the region-annotated type scheme of `apply`:

$$\forall\alpha_4\alpha_2\rho_7\rho_8\rho_9\rho_{10}\epsilon_{11}\epsilon_{12}\epsilon_{13}.((\alpha_2,\rho_{10}) \xrightarrow{\epsilon_{11}.\emptyset} (\alpha_4,\rho_9),\rho_8) \xrightarrow{\epsilon_{12}.\{\mathbf{put}(\rho_7)\}} ((\alpha_2,\rho_{10}) \xrightarrow{\epsilon_{13}.\{\mathbf{get}(\rho_8),\epsilon_{11}\}} (\alpha_4,\rho_9),\rho_7)$$

The latent effect associated with ϵ_{12} shows that when `apply` is applied to a function, it may create (in fact: will create) a function closure in ρ_7. The latent effect associated with ϵ_{11} is empty, because the declaration of `apply` does not tell us anything about what effect its formal parameter `f` must have. Crucially, however, ϵ_{11} is included as an atomic effect in the latent effect associated with ϵ_{13}; whenever the body of `apply f` is evaluated, the body of `f` may be (in fact: will be) evaluated.

[4]Program `kitdemo/apply.sml`.

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 + 1.0)` and `(fn m => m)` have different latent effects. Let us take the function `(fn n => n + 1.0)` as an example. It has region-annotated type with place

$$((\mathtt{real}, \rho_{18}) \xrightarrow{\epsilon_{14}.\{\mathbf{get}(\rho_{18}),\mathbf{put}(\rho_5)\}} (\mathtt{real}, \rho_5), \rho_{17}) \tag{13.1}$$

Here, the effect variable ϵ_{14} and the region variables ρ_{18} and ρ_5 were chosen arbitrarily. (Actually, the region variable ρ_5 denotes the global region for reals.) The region inference algorithm discovers that (13.1) can be derived from the argument type

$$((\alpha_2, \rho_{10}) \xrightarrow{\epsilon_{11}.\emptyset} (\alpha_4, \rho_9), \rho_8)$$

of the type scheme for `apply` by the instantiating substitution

$$S = (\{\alpha_2 \mapsto \mathtt{real}, \alpha_4 \mapsto \mathtt{real}\}, \{\rho_{10} \mapsto \rho_{18}, \rho_9 \mapsto \rho_5, \rho_8 \mapsto \rho_{17}\},$$
$$\{\epsilon_{11} \mapsto \epsilon_{14}.\{\mathbf{get}(\rho_{18}), \mathbf{put}(\rho_5)\}\})$$

Formally, a *substitution* is a triple (S^t, S^r, S^e), where S^t is a finite map from type variables to region-annotated types, S^r is a finite map from region variables to region variables, and S^e is a finite map from effect variables to arrow effects. 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_{11} has to grow to $\{\mathbf{get}(\rho_{18}), \mathbf{put}(\rho_5)\}$ 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 nor that would work well together with the idea of using substitutions to express growth of effects. 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 type schemes with places of all binding occurrences of value variables. Also, one can choose to have arrow effects included in the printout by enabling the flags `print types` and `print effects` in the `Layout` menu. Although enabling these flags gives very verbose output, it is instructive to look at such a term at least 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, `e13(U(U,get(r8),e11))` in the figure denotes an effect variable with an edge to a union node that has edges to an empty union node, a **get** node, and an effect variable 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. In the figure, the binding occurrence of `apply` has been printed with its region-annotated 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 `r26`.

13.4 Region Polymorphism 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 => ··· f ··· f ···)
```

all the uses of `f` use the same regions. Indeed, because `f` occurs free in the type environment while region inference analyses the body of the lambda

```
fun apply
    :all
        'a4,'a2,r7,r8,r9,r10,e11,e12,e13.
         (('a2,r10)-e11->('a4,r9),r8)
         -e12(put(r7))->
         (('a2,r10)-e13(U(U,get(r8),e11))->('a4,r9),r7)
    at r1
    [r7:1]
    [r8:0, r9:0, r10:0]
    (f)=
    fn e13 at r7 x:('a2,r10) => f x;
val y:(real,r5) =
    letregion r16:1, r17:1, r18:1
    in  letregion r19:1
        in apply
            [r16] [real,real] [r16,r17,r5,r18]
            [e14(get(r1),get(r18),put(r5)),
             e20(put(r16)),
             e15(e14(get(r1),get(r18),put(r5)),get(r17))]
           at r19
           (fn e14 at r17 n:(real,r18) =>
            letregion r22:1 in (n + 1.0at r22) at r5 end
           )
        end (*r19:1*)
        5.0at r18
    end (*r16:1, r17:1, r18:1*);
val z:int =
    letregion r25:1, r26:1
    in  letregion r27:1
        in apply
            [r25] [int,int] [r25,r26,r2,r2]
            [e23,e28(put(r25)),e24(e23,get(r26))]
           at r27
           (fn e23 at r26 m:int => m)
        end (*r27:1*)
        6
    end (*r25:1, r26:1*)
```

Figure 13.2: The instantiation of arrow effects keeps different applications of the same function (here `apply`) apart. The output was obtained by compiling the program `kitdemo/apply.sml` with `Control/Optimiser/maximum inline size` menu entry set to `0` and with the flags `print types` and `print effects` enabled.

abstraction, none of the regions that 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, as in the expression 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, as in

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

or further out, as in

`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 with place of the expression nor in the type environment.

13.5 Examples: `map` and `foldl`

Consider the program[5]

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

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

This formulation of `map` is not the most efficient one in the Kit, because it will create one closure for each element in the list, due to currying.[6] 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 listed in Figure 13.3. We see that the region that appears free in the

[5]Program `kitdemo/map.sml`.

[6]When `map` and the application of `map` appear in the same compilation unit, the Kit will automatically specialise `map` to a recursive function that does not have this defect. This specialisation is the result of a general optimisation of curried functions that are invariant in their first argument. The output we present in this section was obtained by setting the menu entry `Control/Optimiser/maximum specialise size` to `0`.

```
fun map at r1 [r7:1, r8:0] (var256)=
     fn at r7 var257 =>
     (case var257
        of nil => nil
        |  _ =>
           let val xs = #1 decon_:: var257;
               val x = #0 decon_:: var257;
               val v20315 =
                   (var256 x,
                    letregion r20:1
                    in  letregion r21:1
                        in map[r20,r8] at r21 var256
                        end (*r21:1*)
                        xs
                    end (*r20:1*)
                   ) at r8
           in  :: v20315
           end
     ) (*case*) ;
val x =
    letregion r25:1, r26:INF, r27:1
    in  letregion r28:1
        in map[r25,r1] at r28 (fn at r27 x => x + 1)
        end (*r28:1*)
        let val v20320 =
                (7,
                 let val v20321 = (11, nil) at r26
                 in :: v20321
                 end
                ) at r26
        in  :: v20320
        end
    end (*r25:1, r26:INF, r27:1*)
```

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 `r27`, the region of the initial argument.

type with place of the successor function (i.e., `r27`) is allocated prior to the call of `map` and that it stays alive throughout the evaluation of the body of `map`. Notice, however, that the closures that are created when `map` is applied do not pile up in `r27`, the region of the successor function. Instead, they are put in local regions bound to `r20`, one closure in each region. Also, if we had given some more complicated argument to `map`, the body of that function could include `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];
```

Because `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; because `foldl` is curried, there are several lambdas between the formal region parameter of `foldl` that indicates where the pair should be put and the allocation point of the pair.

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 region-annotated type of `f`, which is locally live at that point.

What if require that `f` be curried, to avoid the creation of the pair altogether?[7]

[7] Program `kitdemo/fold2.sml`.

```
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 of this program appears in Figure 14.3 on page 146. This saves the allocation of a pair inside loop, although the saving is lost if the the evaluation of `f x` creates a closure.

In short, folding a function over a list may leak two words of memory for each list element.

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. applications of built-in operations such as `+`, `=`, and `:=`;
2. applications of unary value constructors (including `ref`) and unary exception constructors;
3. applications of user-defined functions, that is, functions introduced by `fn` or `fun`.

This chapter is about the last kind of function applications; in the following, we use the term function application to stand for applications 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.

The Kit performs a backwards flow analysis, called *call conversion*, to determine what function calls are tail calls and, more generally, what function calls fall into the four special cases. 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`, and thus check whether specific function calls in the code turn out the way one intended. Call-explicit expressions are produced after regions have been dropped (page 60) 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 that 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 that 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 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 so as to free memory.

[1] Admittedly, not terribly good nomenclature.

14.3 Simple Jump (jmp)

In this section, we consider conditions under which the Kit implements a function call as a simple jump. A call of a region-polymorphic function takes the form

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

where ρ_0 is the region that holds the region vector containing the actual region parameters $\rho_1, \ldots, \rho_n$. 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

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

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

The Kit simplifies this expression to a simple jump

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

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 $\rho_1, \ldots, \rho_n$ are a prefix of the formal region parameters of g, that is, the list of formal region parameters of g is $\texttt{[}\rho_1, \ldots, \rho_n, \rho_{n+1}, \ldots, \rho_{n+k}\texttt{]}$, for some $\rho_{n+1}, \ldots, \rho_{n+k}$; 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 (because f is region-polymorphic). Thus, such a function call is as efficient as an assembly language jump to a constant label.

To understand the above requirements, notice 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 reuse 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 so as 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, because, 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)`.

Notice that (14.1) requires that the `letregion` around the call binds only one region variable (the region used for the region record). The way to avoid that the `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]Program `kitdemo/tail.sml`.

```
fun f' attop r1 [r7:inf] (var512)=
    (case #0 var512
       of 0 => var512
       |  _ =>
          let val b = #1 var512; val n = #0 var512
          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

$$\texttt{letregion}\ \rho_0\ \texttt{in funcall}\ f\ \texttt{[}\rho_1\texttt{, ..., }\rho_n\texttt{]}\ \texttt{at}\,\rho_0\ \ exp\ \texttt{end}.$$

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

$$\texttt{letregion}\ \ \rho_0, \rho_1, \ldots, \rho_k\ \ \texttt{in}\ \ f\ \ \texttt{[]}\ \ \texttt{at}\,\rho_0\ \ exp\ \ \texttt{end}$$

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

$$\texttt{letregion}\ \ \rho_1, \ldots, \rho_k\ \ \texttt{in funcall}\ \ f\ \ exp\ \ \texttt{end}$$

(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

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

is replaced by

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

This case 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 called `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. Because 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 jump 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.

Applications of this form are 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 that 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

`fncall` exp_1 exp_2

14.7 Example: Function Composition

The Basis Library declares function composition as follows[4]

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

The resulting call-explicit expression produced by the Kit is

```
fun o attop r1 [r7:3] (v40378)=
    let val g = #1 v40378; val f = #0 v40378
    in  fn attop r7 x => fnjmp f (fncall g x)
    end
```

Notice 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 the following declaration of folding over lists:[5]

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

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 shown in Figure 14.2. Notice that upon

[4]Program `kitdemo/compose.sml`.
[5]Program `kitdemo/foldl.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 xs' = #1 decon_:: xs;
                val x = #0 decon_:: xs
            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 (*r25:2*)
                        (fncall fncall f x b)
                    end (*r24:4*)
                    xs'
                end (*r22:4*)
            end
      ) (*case*)
```

Figure 14.2: The straightforward implementation of `foldl` uses space linear in the length of the list. (Program `kitdemo/fold1.sml`.)

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 alternative version of `foldl` assumes that `f` is curried:[6]

```
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 14.3. Here the loop is implemented as a jump and there is no new allocation in each iteration, except, of course, for the allocation that `f` might make.[7]

As an exercise, consider the following variant of `foldl`, which assumes that `f` takes a pair as an argument:[8]

```
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.

---

[6]Program `kitdemo/fold2.sml`.

[7]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]Program `kitdemo/fold3.sml`.

```
fun foldl attop r1 [r7:3, r8:3] (f)=
     fn attop r7 b =>
      fn attop r8 xs =>
      letregion r19:2
      in let fun loop atbot r19 [r20:inf] (var514)=
                 (case #0 var514
                    of nil => var514
                    |  _ =>
                       let val b = #1 var514;
                           val xs = #1 decon_:: #0 var514;
                           val x = #0 decon_:: #0 var514
                       in  jmp loop (xs,
                                     fncall fncall f x b
                                    ) sat r20
                       end
                 ) (*case*)
         in  letregion r27:inf
             in let val v40485 =
                        letregion r28:1
                        in funcall loop[atbot r27] atbot r28
                                   (xs, b) atbot r27
                        end (*r28:1*)
                in  #1 v40485
                end
             end (*r27:inf*)
         end
      end (*r19:2*)
```

Figure 14.3: The result of compiling the efficient version of `foldl` (`kitdemo/fold2.sml`) is an iterative function that avoids argument pairs piling up in one region.

Chapter 15

Modules and Projects

In Section 2.8 we described how to compile and run single file programs. In this chapter we describe how to program in the large with the Kit, using Standard ML Modules and the possibility of organising source files into project files. The Kit fully supports Standard ML Modules and it has a sophisticated system for avoiding unnecessary recompilation. In the following section, we describe the notion of projects. We then turn to show how to program with structures, signatures, and functors. To enable the programmer to write efficient programs using the Modules language, we shall also explain how the Kit compiles Modules language constructs.

15.1 Projects

A *project file* is a file that lists the SML source files that make up a project. A project file can also *import* other project files, so one can organise projects in a hierarchical manner. The Kit enforces the restriction that projects may not be cyclic.

Project files have file extension `.pm`. The grammar for project files (*pm*) is given in Figure 15.1. In a project file, one can import source files, other project files, and object files, using absolute or relative paths. Relative paths are relative to the location of the project file. Until now, we have seen a few examples of project files with no imports (see Section 6.4 for such an example). In Section 15.4, we present an example of a project that imports other projects. Object files are `.o` files stemming from compiling C code; in Section 18.7, we shall see an example of a project that imports object

imports	`::=`	*path*.`o` $\langle$:*path*.`o` $\rangle$ *imports*	external object
	|	*path*.`pm` *imports*	project
	|		empty
body	`::=`	*path*.`sml` *body*	source
	|	*path*.`sig` *body*	source
	|	`local` *body* `in` *body* `end` *body*	local
	|		empty
pm	`::=`	`import` *imports* `in` *body* `end`	with import
	|	*body*	basic

Figure 15.1: Grammar for project files, i.e., files with extension `.pm`. Optional phrases are included in angle brackets $\langle\cdots\rangle$. For some file extension *.ext*, *path.ext* denotes either an absolute path or a relative path (relative to the directory in which the project file is located) to a file on the underlying file system.

files. Project files may contain Standard ML style comments. The declared identifiers of a project is the union of the identifiers being declared by a source file of the project, excluding source files that are included using `local`. As an example of the use of `local` to limit what identifiers are declared by a project, consult the project file `kit/basislib/basislib.pm`.

Every source file must contain a Standard ML top-level declaration; the scope of the declaration is all the subsequent source files mentioned in the project file and all other projects that import this project file. Thus, a source file may depend on source files mentioned earlier in the project file and on other imported projects. The meaning of an entire project is the meaning of the top-level declaration that would arise by first expanding all imported projects and then concatenating all the source files listed in the project file (with appropriate renaming of declared identifiers of source files that are included using `local`), in the order they are listed, except that each project is executed only the first time it is imported.

The Kit has a system for managing compilation and recompilation of projects. The system guarantees that the result of first modifying one or more source files of a project and then using the separate compilation system to rebuild the project is the same as if all source files were recompiled. 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 compiled the entire program from scratch. As an example, consider the project file (`kitdemo/scan.pm`) for the text scanning example of Section 12.6. It contains the following two lines:

```
lib.sml
scan.sml
```

The source files for the project are `lib.sml` and and `scan.sml`, which are both located in the directory where `scan.pm` is located. Whereas each of the source files `lib.sml` and `scan.sml` depends on the Basis Library, the source file `scan.sml` also depends on `lib.sml`.

The `Project` sub-menu provides the user with operations for setting a project file name and for reading and compiling the project file. The Kit automatically detects when a source file has been modified. (It uses file modification dates of the operating system for this purpose.) After a project has been successfully compiled and linked, it can be executed by running the command

```
run
```

in the working directory.

The Kit compiles each source file of a project one at a time, in the order mentioned in the project file. A source file is compiled under a given set of assumptions, which provides, for instance, region-annotated type schemes with places for free variables of the source file. Also, compilation of a source file gives rise to exported information about declared identifiers. Exported information may occur in assumptions for source files mentioned later in the project file.

A source file is recompiled if either (1) the user has modified the source file or (2) the assumptions under which the source file was previously compiled have changed. To avoid unnecessary recompilation, assumptions for a source file depend on only the free identifiers. Moreover, if a source file has been compiled earlier, the Kit seeks to *match* the new exported information to the old exported information by renaming generated names to names generated when the source file was first compiled. Matching allows the compiler to use fresh names (stamps) for implementing generative data types, for instance, and still achieve that a source file is not necessarily recompiled even though source files, on which it depends, are modified.

Let us assume that we modify the source file `lib.sml` of the text scanning example. Selecting `Compile and link project` from the `Project` sub-menu causes `lib.sml` to be recompiled. The Kit checks whether the assumptions under which the source file `scan.sml` was compiled have changed, and if so, recompiles `scan.sml`. Modifying only comments or string constants inside `lib.sml` or extending its set of declared identifiers does not trigger recompilation of `scan.sml`.

Some of the information a source file depends on is the ML type schemes of its free variables. It also depends on, for example, the region-annotated type schemes with places of its free variables. Thus it can happen that a source file is recompiled even though the ML type assumptions for free variables have not changed. For instance, the region-annotated type scheme with place for a free variable may have changed, even though the underlying ML type scheme has not.

As an example, consider what happens if we modify the function `readWord` in the source file `lib.sml` so that it puts its result in a global region. This modification will trigger recompilation of the source file `scan.sml`, because the assumptions under which it was previously compiled have changed. Besides changes in region-annotated type schemes with places, changes in multiplicities and in physical sizes of formal region variables of functions may also trigger recompilation.

15.2 Structures

The support for Modules together with the possibility of dividing top-level declarations into different source files provide a mechanism for programming in the large. In the Kit, structures exist only at compile time. Thus one need not worry where structures live at runtime.

We illustrate the compile-time nature of structures with the following example. Consider the project `set.pm`,[1] which mentions the source files `poly_set.sml` and `int_set.sml`. The source file `poly_set.sml` contains the following top-level declaration:

```
structure PolySet =
  struct
    type 'a set = 'a list
```

[1]Project `kitdemo/set.pm`.

```
    val empty = []
    fun singleton x = [x]
    fun mem x l =
      let fun mem' [] = false
            | mem' (y::ys) = x=y orelse mem' ys
      in mem' l
      end
    fun union(s1,[]) = s1
      | union(s1,x::s2) = if mem(x,s1) then union(s1,s2)
                          else x::union(s1,s2)
  end
```

The code generated by the Kit for the `IntSet` structure is exactly as if the declarations were written outside of a structure. As a consequence, when you refer to a component of a structure using qualified identifiers (e.g., `IntSet.mem`), no code is generated for fetching the component from the structure. Moreover, when opening a structure, using the `open` declaration, no code is generated for rebinding the identifiers that become visible.

15.3 Signatures

Signature declarations also exist at compile time only in the Kit. In particular, a signature declaration does not result in code. The source file `int_set.sml` in the project `set.pm`, discussed earlier, contains the signature declaration

```
signature INT_SET =
  sig
    type 'a set
    val empty : int set
    val singleton : int -> int set
    val mem : int * int set -> bool
    val union : int set * int set -> int set
  end
```

Signatures are used in two contexts; for specifying arguments to functors and for providing restricted views of structures using transparent and opaque signature constraints. We defer the discussion of the former use of signatures to Section 15.4.

Transparent signature constraints may both restrict components from a structure and make polymorphic components less polymorphic. Moreover, opaque signature constraints may also make type components of structures abstract. Consider the structure declarations

```
structure IntSet1 : INT_SET = PolySet
structure IntSet2 :> INT_SET = PolySet
```

located in the source file `int_set.sml`. No code is generated for the structure declarations. Instead, the compiler memorises that if you refer to `IntSet1.mem`, for instance, then it is actually `PolySet.mem` that is applied with type instance `int`.

As for the second declaration, opaque signature constraints are eliminated at compile time (after elaboration) and transformed into transparent signature constraints.

15.4 Functors

Functors map structures to structures. The Kit specialises a functor every time it is applied. Thus, types that are abstract for the programmer (inside a functor body) become visible to the compiler. Region-annotated type schemes and other information about identifiers in the actual functor argument are available when the Kit compiles the functor body.

For practical reasons, it is important that not all functor applications are expanded at once, since this could cause intermediate representations of programs to become as large as (or even much larger than) the entire program. Further, non-restricted in-lining could lead to unnecessary recompilation upon modification of source files. Instead, the largest structure declarations not containing functor applications are compiled into separate chunks of machine object code. Assumptions for compiling these structure declarations are memorised, so that the generated code can be reused upon modification of source files if the assumptions do not change.

Consider the following project:[2]

```
import utils/utils.pm
in SET.sml Set.sml SetApp.sml
end
```

[2]Project `kitdemo/Set.pm`.

The project imports the sub-project `utils.pm`,[3] which provides a structure `ListUtils` containing the function `pr_list` with type scheme `('a -> string) -> 'a list -> string`. The source file `Set.sml` is listed in Figure 15.2. It declares the functor `Set`, which takes as arguments the element type for the set, an ordering function on elements, and a function for providing a string representation of elements.

The source file `SetApp.sml` is listed in Figure 15.3. It constructs a structure `IntSet` by applying the functor `Set` to appropriate arguments including an ordering operation on integers and an operation for giving the string representation of an integer. The `IntSet` structure is used for constructing a set `{2,5}`, which the program prints using the built-in `print` function.

The body of the `Set` functor is instantiated to form the code for the `IntSet` structure. The result of instantiating the `Set` functor is first translated into a Lambda program and then translated into a MulExp program. The MulExp call-explicit code for the `mem` function is shown in Figure 15.4.

Notice that the code for the `mem` function refers to compiled code for the `lt` function; the Kit does not currently propagate enough information accross module boundaries that the use of the `lt` function is reduced to a built-in comparison on integers. Instead, for simplicity, the Kit compiles the argument to the `Set` functor in the source file `SetApp.sml` into separate code:

```
let fun lt attop r1 [] (v40355)= #0 v40355 < #1 v40355;
    fun pr attop r1 [r10:inf] (a)= jmp toString a
in  {|pr: (_,r1), lt: (_,r1)|}
end
```

Here, the `toString` function stems from the `Int` structure of the Basis Library and the primitive operation `<` provides a built-in comparison on integers.

[3]Project `kitdemo/utils/utils.pm`.

```
functor Set (eqtype elem (*total order*)
             val lt : elem * elem -> bool
             val pr : elem -> string)
     : SET where type elem = elem =
  struct
    type elem = elem
    type set = elem list
    val empty : set = []
    fun singleton e = [e]
    fun mem x l =
      let fun mem' [] = false
            | mem' (y::ys) = if lt(y,x) then mem' ys
                             else not(lt(x,y))
      in mem' l
      end
    fun union(s1,s2) =
      let fun U (t as ([], [], acc)) = t
            | U ([], y::ys, acc) = U([], ys, y::acc)
            | U (x::xs, [], acc) = U(xs, [], x::acc)
            | U (s1 as x::xs, s2 as y::ys, acc) =
                U(if lt(x,y) then (xs, s2, x::acc)
                  else if lt(y,x) then (s1, ys, y::acc)
                  else (xs, ys, y::acc))
      in rev(#3(U(s1, s2, [])))
      end
    val pr = fn s => ListUtils.pr_list pr s
  end
```

Figure 15.2: The source file `kitdemo/Set.sml`.

```
structure IntSet = Set(type elem = int
                       val lt = op <
                       fun pr a = Int.toString a)
open IntSet
val _ = print (pr (union(singleton 2, singleton 5)))
```

Figure 15.3: The source file `kitdemo/SetApp.sml`.

```
fun mem attop r1 [r11:4] (x)=
  fn attop r11 l =>
  letregion r15:4
  in let fun mem' atbot r15 [] (var508)=
           (case var508
              of nil => false
               |  _ =>
                 let val ys = #1 decon_:: var508;
                     val y = #0 decon_:: var508
                 in  (case letregion r20:0, r22:2
                           in funcall lt[] (y, x) atbot r22
                           end (*r20:0, r22:2*)
                        of true => jmp mem' ys
                        |  _ =>
                           jmp not
                             letregion r27:0, r29:2
                             in funcall lt[] (x, y) atbot r29
                             end (*r27:0, r29:2*)
                     ) (*case*)
                 end
           ) (*case*)
     in  funcall mem'[] l
     end
  end (*r15:4*);
```

Figure 15.4: The MulExp call-explicit code for the `mem` function resulting from instantiating the `Set` functor.

Part III

System Reference

Chapter 16

Using the Profiler

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

The profiler consists of several tools that can be used to analyse the dynamic memory behaviour of a program. First of all, the profiler lets you create graphs of the dynamic memory usage of the program. Three different kinds of graphs may be created:

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

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

  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 that, for a particular region, shows the objects allocated in the region, with one coloured band for each allocation point in the region-annotated program[1]. Each allocation point is annotated with a *program point*, which is a unique number that identifies

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

the allocation.[2] To inspect region-annotated programs with program points, enable the flag `print program points` found in the `Layout` sub-menu in addition to the flag `print call-explicit expression`, say, from the `Printing of intermediate forms` sub-menu.[3]

If you have an object profile showing that program point `pp42`, say, contributes with allocation, you can search for `pp42` in the region-annotated program and thus find the construct that caused the allocation.

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

In addition to the possibility of generating programs with program points, it is also possible, during compilation, 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 comes in handy when profiling large programs and when one wants to find out why a formal region variable is instantiated to a certain `letregion`-bound region variable.

The following example clarifies the use of a region flow graph. Suppose the region profile shows that `r5` is responsible for most of the memory usage. Further, suppose an object profile of `r5` shows that program point `pp345` is responsible for most of the allocation. 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 that 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`.

The profiling process is sketched in Figure 16.1.

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

---

[2]Program points are unique. In particular, for a project with two program units, the program points in the region-annotated programs for the two units will be distinct.

[3]Program points are annotated during physical size inference.

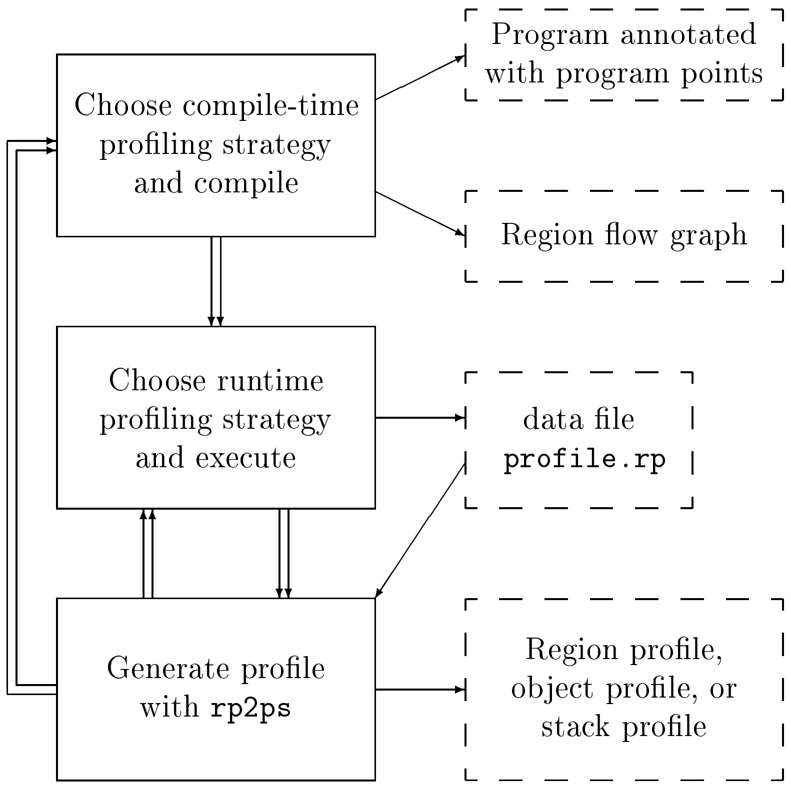


Figure 16.1: Overview of the profile process. The process sometimes requires the programmer to refine the runtime profiling strategy, or even the compile-time profiling strategy. Dotted boxes represent output from the compiler, from executing the program, and from using the tool `rp2ps`, which generates PostScript graphs from the exported data file.

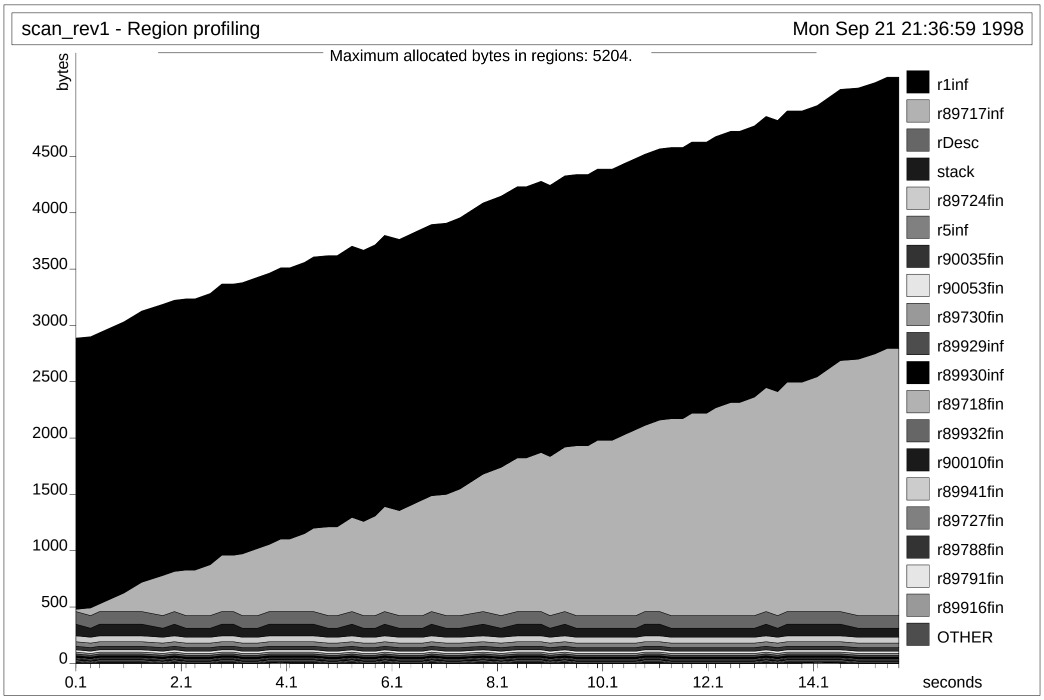


Figure 16.2: Memory is accumulated in the top two bands. The global regions `r1` and `r89717` hold the largets amount of memory. The graph was generated by first compiling the `kitdemo/scan_rev1.pm` project with profiling enabled. Then by executing `echo life.sml | run -microsec 10000` and finally by typing `rp2ps -region -name scan_rev1`.

16.1 Example: Scanning Text Files Again

In this section, we concentrate on the general principles of profiling. As an example, we investigate a revised version of the `kitdemo/scan.pm` project (see Section 12.6). Instead of asking for a list of input files to scan (as project `scan.pm` does), the revised version of the scan project asks for only one input file, which it then scans 50 times.[4]

The first thing to do is to get an overview of the memory usage of the program. A region profile of the program gives you just that. See Figure 16.2.

[4] Project `kitdemo/scan_rev1.pm`.

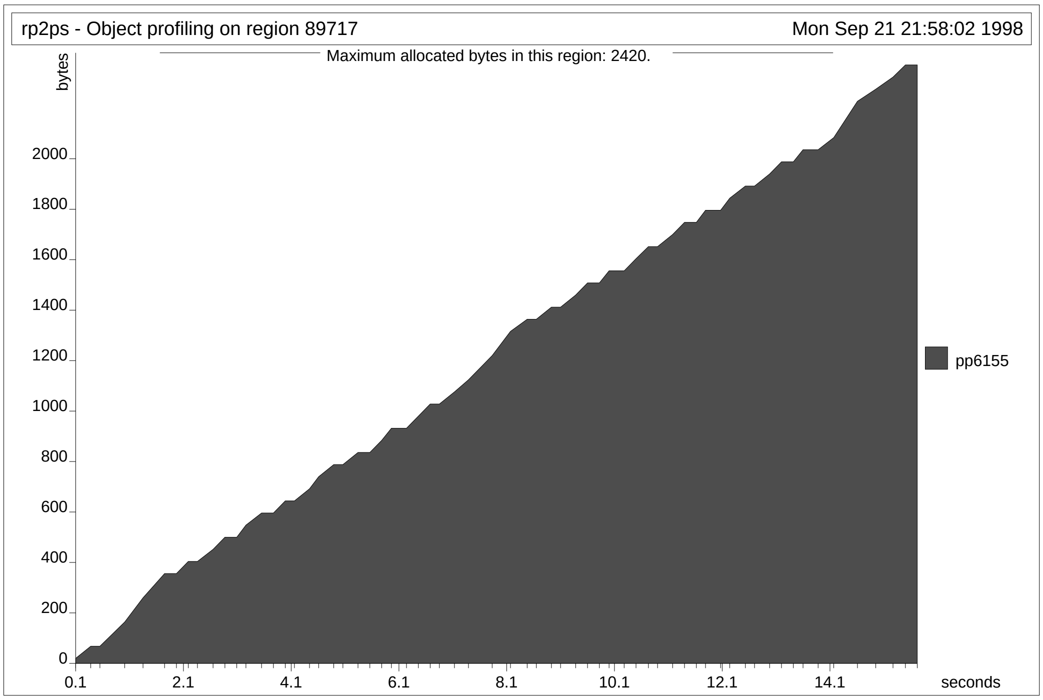


Figure 16.3: There seems to be a space leak at program point `pp6155`. The graph was generated by typing `rp2ps -object 89717`.

The graph shows that region `r1` holds the largest amount of memory, but also that it does not increase over time. Region `r89717`, however, accumulates more memory for each time it scans the file `life.sml`.

To see what happens in region `r89717`, we make an object profile of that region, see Figure 16.3. The object profile shows that program point `pp6155` continually allocates memory that is first freed when the program stops. We now search for `pp6155` in the log files of the basis library, that is, we execute the UNIX command

```
fgrep pp6155 *.log
```

from the directory `kit/basislib`, and find that `pp6155` appears in the file `General.sml.log`, which contains the following fragment:

```
fun implode attop r1 pp6154 [r45074:inf] (chars)=
  ccall(implodeCharsProfiling,  sat r45074 pp6155, chars);
```

So the space leak is caused by function `implode` being called with region `r89717` instantiated for the formal region variable `r45074`.

We now search for `r89717` in file `scan_rev1.sml.log` and find the following fragment of the region flow graph:

```
toString[r75249:inf] --r75249 attop--> LETREGION[r89717:inf]
readWord[r89487:inf] --r89487 atbot--> [*r89717*]
```

The fragment is read as follows. The formal region variable `r75249` is instantiated to the `letregion`-bound region variable `r89717` in a call to `toString`. Moreover, also the formal region variable `r89487` (of function `readWord`) is instantiated to `r89717`. (The asterisks (*) denote that the node has been displayed before.)

Region flow graphs are local to each program in a project. A call to a non-local region-polymorphic function introduces an edge in the region flow graph, but the graph says nothing about in which module the called function is located. Thus, it may be necessary to look in several log files to find the path from a formal region variable to an actual region variable. By inspecting the call-explicit programs found in `basislib/Int.sml.log` and `kitdemo/lib.sml.log` one finds that both `toString` and `readWord` eventually call `implode`. However, `readWord` is called only initially, thus, we conclude that the space leak is caused by function `toString` (from the `Int` structure) being called with region `r89717` instantiated for the formal region variable `r75249`. Indeed, by inspecting the calls to `toString` in the call-explicit program found in `scan_rev1.sml.log`, we see that `toString` is called with actual region `r89717`.

The `concat` function from the initial basis catenates a list of strings. But all the strings in the argument list to `concat` are required to be in the same region. Thus, whenever a file is reported (see Figure 16.4), strings created by the `Int.toString` function are put in the region that also holds the file name for the report (which is read using the function `readWord`); and this region is non-local to the `do_it` function, which implements the main loop of the program.

One way of solving the space leak is to make a copy of `filename` at the call to `report_file` in function `scan_file`:

```
fun scan_file (filename: string) : (int*int)option=
  let val is  = TextIO.openIn filename
  in let val (n,inside)  = scan is
```

```
fun report_file(filename, n, inside) =
  writeln(concat[filename, ": size = ", Int.toString n,
                 " comments: ", Int.toString inside, " (",
                 (Int.toString(percent(inside, n))
                  handle _ => "-"), "%)"])

fun scan_file (filename: string) : (int*int)option=
 let val is  = TextIO.openIn filename
 in let val (n,inside)  = scan is
    in TextIO.closeIn is;
       report_file(filename, n, inside);
       SOME(n,inside)
    end handle NotBalanced =>
        (writeln(filename ^ ": not balanced");
         TextIO.closeIn is;
         NONE)
 end handle IO.Io {name,...} =>
     (writeln(name^" failed."); NONE)

fun main():unit =
  case readWord(TextIO.stdIn)
    of SOME filename =>
      let fun do_it 0 = ()
            | do_it n = (scan_file filename; do_it (n-1))
      in do_it 50
      end
     | NONE => ()
```

Figure 16.4: Fragments of `scan_rev1.sml`. All the strings in the argument list to `concat` are put in the same region.

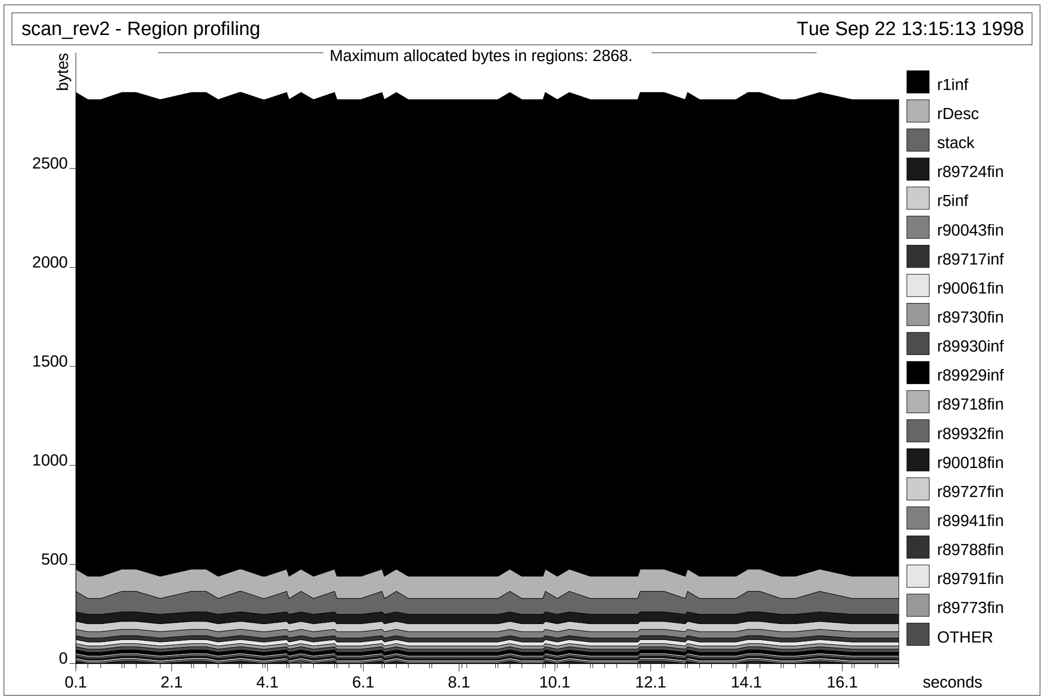


Figure 16.5: There is no space leak: no matter how many times we scan the file, the project will use the same number of words. The graph was generated by executing `echo life.sml | run -microsec 10000` and `rp2ps -region`.

```
      in TextIO.closeIn is;
         report_file(filename^"", n, inside);
         SOME(n,inside)
      end handle NotBalanced =>
          (writeln(filename ^ ": not balanced");
           TextIO.closeIn is;
           NONE)
   end handle IO.Io {name,...} =>
       (writeln(name^" failed."); NONE)
```

Project `kitdemo/scan_rev2.pm` implements the modification. Figure 16.5 shows a region profile of the `scan_rev2.pm` project.

16.2 Compile-Time Profiling Strategy

Before compiling a program for the purpose of profiling, one must decide on a *compile-time profiling strategy*, see Figure 16.1. The compile-time profiling strategy directs the embedding of profiling instructions in the generated code and instructs the compiler whether to report a region flow graph.

The compile-time profiling strategy is set up in the `Profiling` sub-menu:[5]

```
Profiling

0      region profiling............................ off >>>
1      show region flow graph and generate .vcg file off
2      paths between two nodes in region flow graph. [] >>>
3      instruction count profiling.................. off
```

Region profiling is enabled by toggling the item `region profiling`. If you want the Kit to report region-annotated programs with program points, you should enable either `print physical size inference expression` or `print call-explicit expression` from the `Printing of intermediate forms` sub-menu together with the item `print program points` from the `Layout` sub-menu.

To make the compiler report a region flow graph, enable `show region flow graph and generate .vcg file`. The region flow graph is reported both in text layout and in a `.vcg` file, which, when interpreted by the VCG tool, provides a graphical version of the graph.[6]

As a running example, we use the `life` program.[7] We enable the `region profiling` option and the `show region flow graph and generate .vcg file` option from the `Profiling` sub-menu together with the options `print`

[5] The `instruction count profiling` option is available only with the native 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 instructions in the link file. It should be used only when region profiling is disabled. If the number of instructions executed gets too large, the `Overflow` exception is raised.

[6] The VCG tool (Visualization of Compiler Graphs) can be obtained from

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

We use version 1.30, which can be found in file `vcg.1.30.r3.17.tar`.

[7] Program `kitdemo/life.sml`.

`call-explicit expression` and `print program points` from the sub-menus `Printing of intermediate forms` and `Layout`, respectively.

By enabling the option `Log to file` from the `File` sub-menu and by compiling the program using the `Compile an sml file` entry in the main-menu, the Kit now generates several files of which we have `life.log` (containing, among other things, the call-explicit region-annotated program with program points and the region flow graph in text layout), `life.vcg` (the region flow graph to be displayed with the VCG tool) and the executable file `run`.

16.3 The Log File

In the file `life.log` you find the call-explicit region-annotated program with program points and the region flow graph in text layout for the `life.sml` source file. The region flow graph is found by searching for `REGION FLOW GRAPH FOR PROFILING`. The graph contains the following fragment (modified slightly to fit here):

```
cp_list[r89698:inf]
 --r89698 sat-->  [*r89698*] ;
 --r89698 atbot-->   LETREGION[r90769:inf];
 --r89698 sat-->   nthgen'[r90246:inf]
                    --r90246 sat-->   [*r90246*] ;
                    --r90246 atbot-->   LETREGION[r90812:inf];
```

The region flow graph is almost equivalent to the graph used by the storage mode analysis (see page 106). In the graph, region variables are nodes and there is edge between two nodes ρ and ρ' if ρ is a formal region parameter of a function that is applied to actual region parameter ρ'. It follows that `letregion`-bound region variables are always leaf nodes.

Nodes in the graph are written in square brackets, where for example `cp_list[r89698:inf]` means that `r89698` is a formal region parameter of 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 infinite regions and *size* for finite regions of size *size* words.

Edges are written with the *from node* identifier annotated on them. The edge points to the *to node*. The fragment

```
cp_list[r89698:inf]
 --r89698 sat-->  [*r89698*] ;
```

is read: there is an edge from node `r89698` to node `r89698` and node `r89698` has been written earlier. From the cycle in the graph, one can conclude that `cp_list` calls 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
```

The region flow graph can get very complicated to read because we may have mutually recursive functions, which give many 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 annotated on the scc node.

Consider, for example, the following fragment of the scc version of the region flow graph for the `life` program:

```
[sccNo 184: r90473,]   --sccNo 184-->   [sccNo 183: r90717,];
```

Here, we have a scc node (id 184) containing region variable `r90473` and an edge to scc node (id 183) containing region variable `r90717`.

16.4 Region Flow Paths

If you are interested in the possible paths from one region variable to another, the Kit can find them for you. Assume that you have an object profile for some region variable ρ showing that a certain allocation point is responsible for the allocations and that the region variable written at the allocation point is not ρ, but some other region variable ρ'. In this case, ρ' must be a formal region variable of some region-polymorphic function; it is now interesting to find out how ρ' has been instantiated to ρ.

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

```
Profiling

0       region profiling........................... off >>>
1       show region flow graph and generate .vcg file off
2       paths between two nodes in region flow graph. [] >>>
3       instruction count profiling.................. off

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

>2
<type an int pair list of region variables,
e.g. [(formal reg. var. at pp.,letregion bound reg. var.)]>
or up (u): >
```

At this point, you can type in a list of integer pairs, that is, 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`:[8]

```
[Starting layout of paths...
    [Start path:
        [sccNo 63: r89698,]--->
        [sccNo 62: r90246,]--->
        [sccNo 61: r90812,]
    ]
...Finishing layout of paths]
```

If you look at the region flow graph on page 168, you see that the only path from region `r89698` to region `r90812` goes through function `nthgen'`, that is, `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

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

[8] Because region variables may change when re-compiling a source file, it may be necessary to start all over by starting the Kit again and compile the program again to make sure that the regions you have specified will match the regions in a region flow graph of a previous compilation.

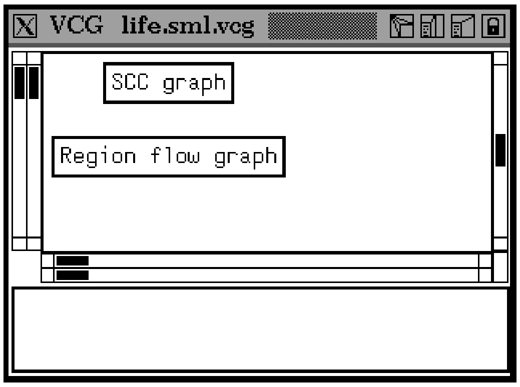


Figure 16.6: The VCG graph contains two nodes. The node "Region flow graph" represents the folded region flow graph and the node "SCC graph" represents the folded strongly connected componemt graph.

If you look at the call-explicit region-annotated program in file `life.log`, you may notice that the function `copy` has been in-lined by the optimiser.

16.5 Using the VCG Tool

The VCG tool can be used to visualise region flow graphs exported in `.vcg` files. We assume that the tool is installed and that it can be started by typing `xvcg` at the command prompt. We use the file `life.vcg` as a running example. Typing `xvcg life.vcg` at the command prompt gives the window shown in Figure 16.6.

The two graphs are exported *folded*, meaning that they are represented in the window as one node each. 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`, choose with the left mouse button the node representing the graph that you want to unfold. Then press the right mouse button to unfold the chosen graph. Figure 16.7 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, and so on from the pull-down menu. The graph is folded again by choosing `Fold Subgraph` and clicking on one of the

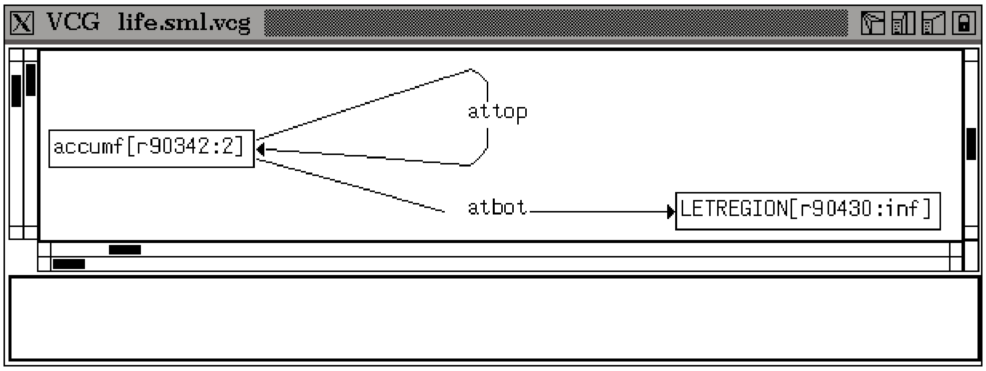


Figure 16.7: A small fragment of the region flow graph.

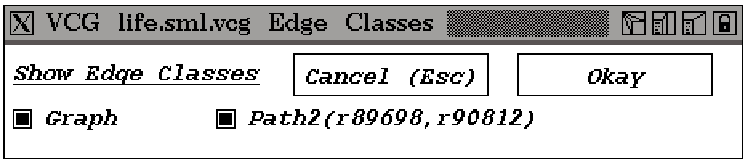


Figure 16.8: After choosing the `Expose/Hide edges` facility you get this window. The window shows that there are two classes of edges in the graph; one for the region flow graph and one for the path from node `r89698` to node `r90812`. If you have generated the path from Section 16.4, the option `Path2(r89698,r90812)` is present.

nodes. All nodes in the graph then turn black; clicking on the right mouse button then folds the 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 16.8.

Each path is numbered because there can be several paths between the same two nodes. Clicking on the `Graph` edge class will hide the edges in the region flow graph so that edges in the generated path are the only edges shown, see Figure 16.9.

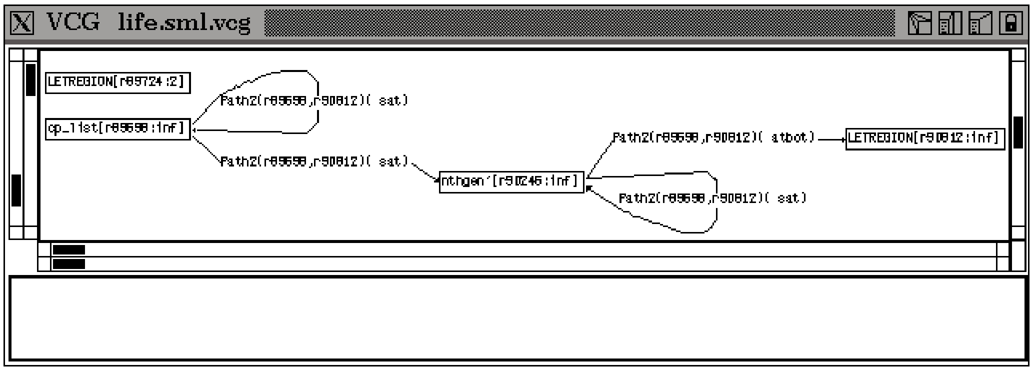


Figure 16.9: The figure shows the path between node `r89698` and `r90812`.

16.6 Runtime Profiling Strategy

When the source program has been compiled and linked, you have an executable file, `run`. Typing `run` at the command prompt will execute the program with a predefined runtime profiling strategy, which is displayed when the program is run:

```
---------------------Profiling-Enabled---------------------
 The profile timer (unix virtual timer) is turned on.
 A profile tick occurs every 1th second.
 Profiling data is exported to 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. In general, what timers are available is very much system dependent. Under the HP-UX operating system, there are three possible timers, each of which can be enabled using one of the following options:[9]

`-realtime` Real time.

`-virtualtime` The execution time for the process.

`-profiletime` The execution time for the process together with the time used in the operating system on behalf of the process.

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

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 (and the slower the program will run). The *time slot* (i.e., 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`.[10]

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

There are several other possible command line options; use the `-h` option or the `-help` option for details.

16.7 Regions Statistics

If the executable file `run` is executed with the option `-showStat` then *region statistics* is printed just before the program terminates. Region statistics includes information about the use of regions and does not depend on the specifics of the runtime profiling strategy; in fact, region statistics includes only exact, non-sampled values for the program. Assuming that `run` is the executable file generated by compiling the program `life` with profiling enabled, executing `run -showStat` yields—just before the program terminates—the region statistics shown in Figure 16.10.

The `MALLOC` part of Figure 16.10 shows how memory is allocated from the operating system.

Each infinite region form a linked list of one or more *region pages* whose size is found in the `REGION PAGES` part. The value

```
Max number of allocated pages: 53
```

multiplied by

```
Size of one page: 800 bytes
```

gives

```
Max space for region pages: 42400 bytes (0.0Mb)
```

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

```
MALLOC
  Number of calls to malloc: 2
  Alloc. in each malloc call: 24240 bytes
  Total allocation by malloc: 48480 bytes (0.0Mb)

REGION PAGES
  Size of one page: 800 bytes
  Max number of allocated pages: 53
  Number of allocated pages now: 4
  Max space for region pages: 42400 bytes (0.0Mb)

INFINITE REGIONS
  Size of infinite region descriptor: 16 bytes
  Number of calls to allocateRegionInf: 95764
  Number of calls to deallocateRegionInf: 95761
  Number of calls to alloc: 858873
  Number of calls to resetRegion: 123378
  Number of calls to deallocateRegionsUntil: 0

ALLOCATION
  Max alloc. space in pages: 18056 bytes (0.0Mb)
    incl. prof. info: 36240 bytes (0.0Mb)
  Infinite regions utilisation (36240/42400): 85%

STACK
  Number of calls to allocateRegionFin: 3164508
  Number of calls to deallocateRegionFin: 3164508
  Max space for finite regions: 6608 bytes (0.0Mb)
  Max space for region descs: 256 bytes (0.0Mb)
  Max size of stack: 7412 bytes (0.0Mb)
    incl. prof. info: 11244 bytes (0.0Mb)
    in profile tick: 4524 bytes (0.0Mb)
```

Figure 16.10: Region statistics for the `life` program.

In the `INFINITE REGIONS` part, we see the number of calls to infinite region operations such as `allocateRegionInf` and `alloc`. The program allocates 95764 infinite regions and deallocates 95761; the three global regions are not deallocated before the region statistics is printed and the program terminates. The program allocates 858873 objects in infinite regions. It has been possible to reset an infinite region 123378 times. The `deallocateRegionsUntil` operation is called whenever an exception is raised, thus, we see that no exceptions were raised by the program.

Because objects allocated in infinite regions are not split across different region pages (except strings), it is not always possible to fill out a region page entirely. In the `ALLOCATION` part, the value

```
Infinite regions utilisation (36240/42400): 85%
```

shows memory utilisation for infinite regions at the moment where the program has allocated the largest amount of memory in infinite regions.

In the `STACK` part, we see that the program allocates and deallocates the same number of finite regions. We also see that the space used for finite regions is 6608 bytes and that the total use of stack space is 7412 bytes (excluding space used to hold profiling information). The stack size values

```
incl. prof. info: 11244 bytes (0.0Mb)
in profile tick: 4524 bytes (0.0Mb)
```

can be used to see if it is necessary to profile with a smaller time slot, which will often lower the difference between the two values.

16.8 Processing the Profile Data File

The profile datafile `profile.rp` can be processed by the graph generator `rp2ps` (read: RegionProfile2PostScript) found in the `kit/bin` directory.[11] The graph generator is controlled by command line options.

A region profile is produced by typing

```
rp2ps -region
```

[11] The `rp2ps` program is based on a profiler by Colin Runciman, David Wakeling and Niklas Röjemo.

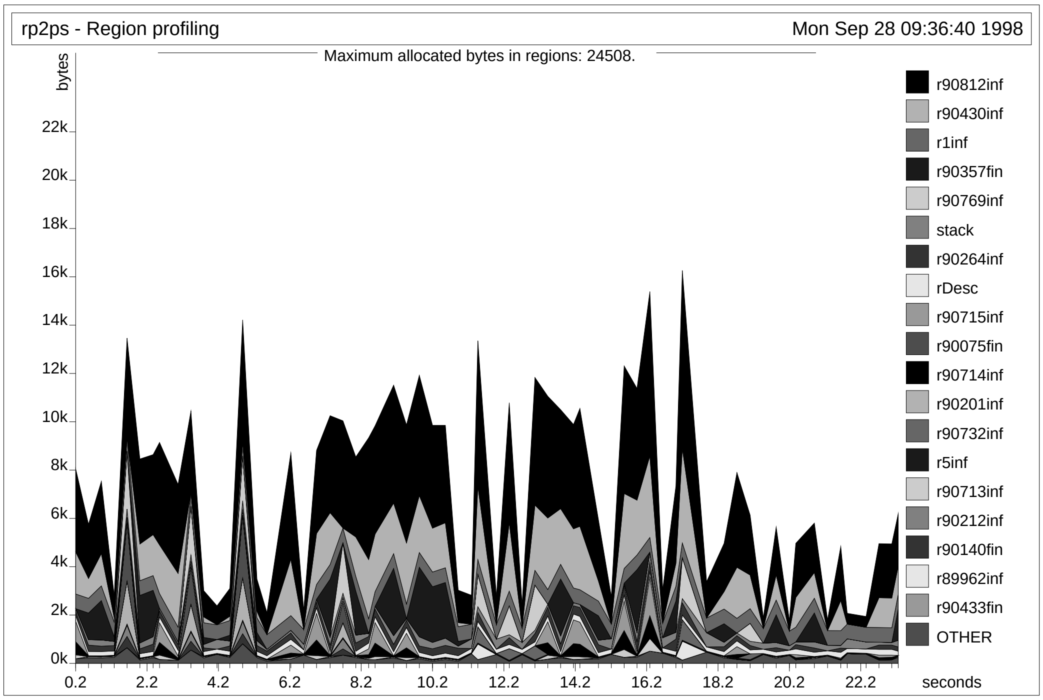


Figure 16.11: Region profile of the `life` program. The region that occupies the largest area is at the top. The graph was produced by first typing `run -microsec 20000` at the command prompt and then typing `rp2ps -region`.

at the command prompt. The program produces a PostScript file in file `region.ps` by reading profile information from the profile data file `profile.rp`, see Figure 16.1. A region profile for the `life` program is shown in Figure 16.11. The region that occupies the largest area is at the top. If there are more regions than can be shown in different shades, then the smallest regions 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 program. Infinite regions end with `inf` and finite regions end with `fin`. There are also a band named `rDesc` and a band named `stack`. The `rDesc` band shows the memory used on region descriptors of infinite regions on the stack. The stack band shows stack usage excluding finite regions and region descriptors for infinite regions.

The vertical line marked “Maximum allocated bytes in regions” in Fig-

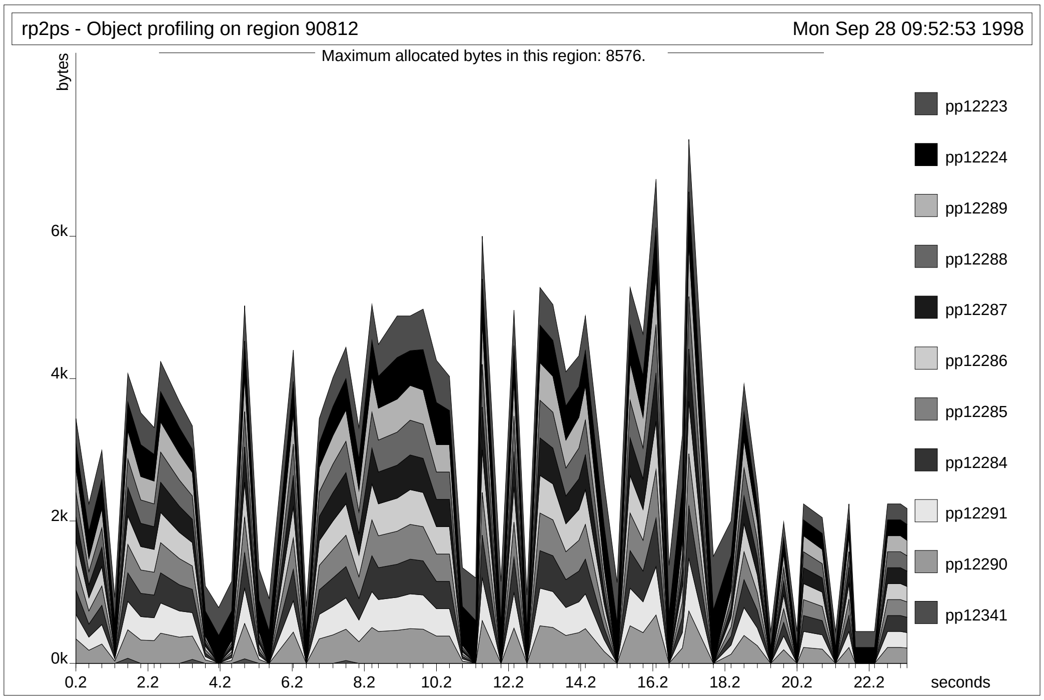


Figure 16.12: The object profile shows all allocation points allocating into region `r90812`.

ure 16.11 is called the *maximum allocation line*; it shows the maximum number of bytes allocated in regions when the program was executed. Because we also show the stack use on the graph (as the `rDesc` and `stack` band), the maximum allocation line is offset upwards by the stack use at the point where region allocation was at its highest. The space between the maximum allocation line and the top band shows the inaccuracy of the profiling strategy. To decrease the gap, it often helps to use a smaller time slot.

The largest region shown in Figure 16.11 is `r90812`. An object profile of region `r90812` is produced by typing

```
rp2ps -object 90812
```

at the command prompt. We obtain the object profile shown in Figure 16.12.

We see that allocation point `pp12223` is responsible for the largest amount of allocations in the program. The allocation point may be found in the

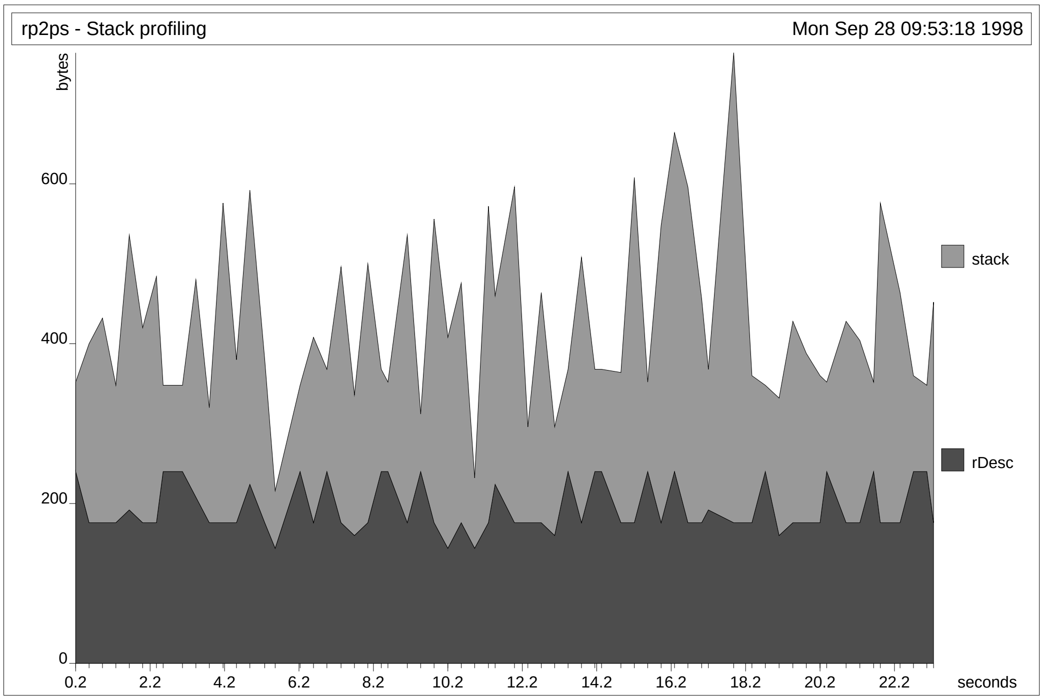


Figure 16.13: Memory usage on the stack excluding space for finite regions.

region-annotated program resulting from compiling the `life` program (remember to enable printing of program points). In general, program points may also stem from the Basis Library (search the .log files in the directory `kit/basislib`).

The stack profile shown in Figure 16.13 shows memory usage on the stack, excluding space used by finite regions. A stack profile is generated by typing

```
rp2ps -stack
```

at the command prompt.

16.9 More Advanced Graphs with `rp2ps`

This section gives a quick overview of the more advanced options that can be passed to `rp2ps`. First of all, it is possible to name the profiles with the `-name` option. Comments are inserted on the x-axis with the `-comment` option.

The profile data file may contain a large number of *samples* (the data collected by a profile tick is called a sample). By default, `rp2ps` uses only 64 samples. You can alter the setting with the `-sampleMax` option. The following two algorithms are used to sort out samples:

`-sortBySize` The n (specified by `-sampleMax`) largest samples are shown.

`-sortByTime` The n samples shown are equally distributed over time (default).

The `-sortBySize` option is useful if your profiles have a large gap between the top band and the maximum allocation line. If there is a large gap when using option `-sortBySize`, then it may help to profile with a smaller time slot. You can use the `-stat` option to see the number of samples in the profile data file. It is printed as `Number of ticks:`.

Figure 16.14 shows the profile for the following command line:

```
rp2ps -region -sampleMax 50 -name life
      -comment 9 "A comment at time 9" -sortByTime
```

The graph generator recognises several options that are not shown here. Help on these options is obtained by typing `rp2ps -h` or `rp2ps -help` at the command prompt.

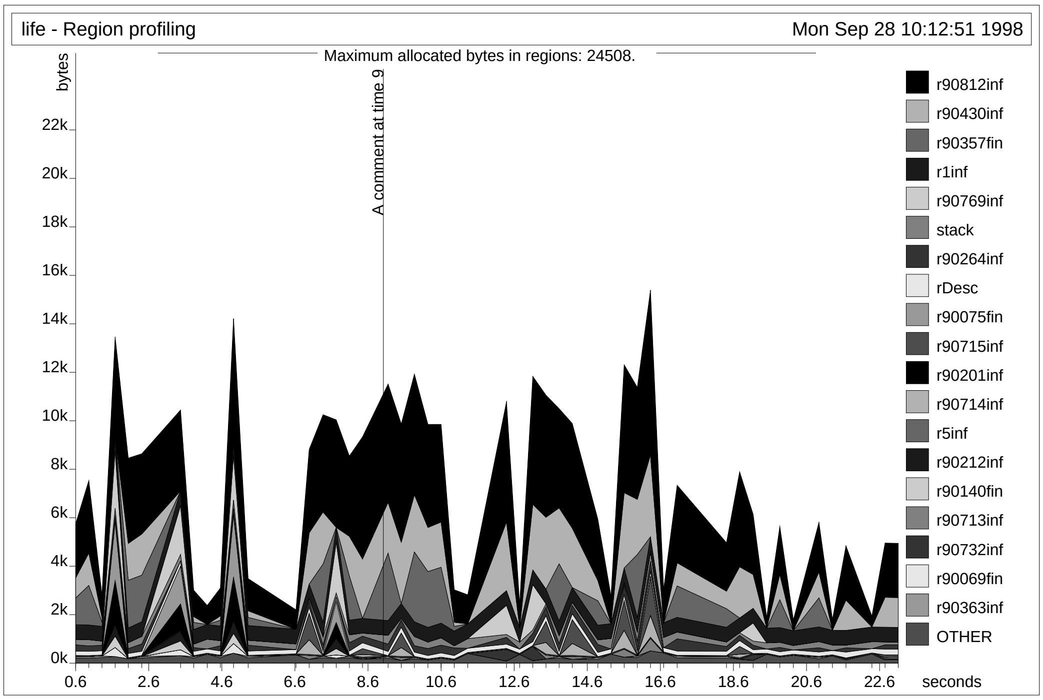


Figure 16.14: It is possible to insert comments in profile graphs.

Chapter 17

Interacting with the Kit

Starting the Kit was described in Section 2.8. To leave the Kit, type `quit` followed by a return character.

We have already described how to compile and run single source files (Section 2.8) and projects (Chapter 15). In the following sections, we give an overview of the Kit sub-menus that control printing and layout of intermediate forms. In Section 17.3, we explain how to use a so-called script file to set personal preferences for menu entries in the Kit.

17.1 Printing of Intermediate Forms

The menu `Printing of intermediate forms` controls what intermediate forms are printed when a program is compiled. A summary of the major phases that produce printable intermediate forms is shown in Figure 17.1. 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. This statistics can be printed by turning on the flag `statistics after optimisation` in the `Control/Optimiser` menu; the other flags found in this menu controls the optimiser.

Storage mode analysis (see Chapter 12) results in a MulExp expression, which can be printed by turning on the flag `print atbot expression`. After that, regions with only **get** effects are removed from the MulExp expression (see page 60). To see the resulting expression, turn on `print drop regions expression` or `print drop regions expression with storage modes`. (The latter flag also prints storage modes.) Physical size inference

Phase	**Result**	**Flag(s) that Print Result**
Elaboration	Lambda	(∗)
Elim. of Poly. Eq.	Lambda	(∗)
Lambda Optimiser	Lambda	`print optimised lambda expression` (∗)
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` (∗) `print drop regions expression with storage modes`
Physical Size Inference	MulExp	`print physical size inference expression` (∗)
Call Conversion	MulExp	`print call-explicit expression` (∗)
Code Generation	KAM	`print KAM code before register allocation` (∗)
Register Allocation	KAM	`print KAM code after register allocation` (∗)

Figure 17.1: The table shows how the menu items in the `Printing of Intermediate Forms` menu 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).

then 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 expression after physical size inference, turn on `print physical size inference expression`. After that, call conversion converts the MulExp expression to a call-explicit expression (see page 138). To see the result, enable the flag `print call-explicit expression`. After that, KAM code is generated. The KAM code before register allocation can be inspected by enabling the flag `print KAM code before register allocation`. The result of register allocation can be viewed by enabling the flag `print KAM code after register allocation`.

17.2 Layout of Intermediate Forms

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 region-annotated types, effects, and region allocation points (e.g., `at` ρ). All eight combinations of these three flags are possible, but if `print effects` is turned on it is best also to turn the two others on so that one can see where the effect variables and region variables that appear in arrow effects are bound.

Enabling the flag `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.

17.3 Using Script Files for Preferences

The Kit allows you to create a so-called *script file*, which can hold preferences for most of the entries found in the Kit menu. A script file can be provided either when the Kit is started or dynamically from within the Kit menu when the Kit is running. To provide a script file `script` at the time the Kit is started, type

```
kit -script script
```

from the shell. If no script file is provided on the command line, the Kit will try to read a script file `kit.script` from the working directory. During a session with the Kit, a script file can be read using the menu entry `Read a script file` from the `File` menu.

A script file is comprised by a sequence of *preferences*, each of which provides a setting for a given entry. Here is a script file that enables the boolean entry `print drop regions expression` (located in the `Printing of intermediate forms` sub-menu) and sets the integer entry `maximum inline size` (located in the `Control/Optimiser` sub-menu) to `0`:[1]

```
val print_drop_regions_expression : bool = true
val maximum_inline_size : int = 0  (* disable in-lining *)
```

Notice that spaces in the menu entries are replaced with scores in the script file and that script files may include ML style comments. By selecting `print all flags and variables` from the `Control` sub-menu, the Kit prints a table of all entries that may be set from a script file.

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.

[1]Script file `kitdemo/ex.script`.

Chapter 18

Calling C Functions

In this chapter, we describe how the Kit programmer can call C functions from within Standard ML programs. The Kit allows ML values to be passed to C functions, which again 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, whereas ML strings in the Kit are represented as a linked list of bounded sized character arrays. To allow the programmer to conveniently convert between C values and ML values, the Kit provides conversion functions and macros for commonly used data structures.

When the Kit calls a C function, data structures returned by the function are stored in regions that are allocated by the Kit. For dynamically sized objects of the resulting value, such as strings and lists, regions are allocated by the Kit and passed to the C function as additional arguments; the C function must then itself allocate space in these regions for the dynamically sized data structures. Moreover, for those parts of the resulting value for which the size can be determined statically, pointers to already allocated space are passed to the C function as additional arguments.

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. Instead, the Kit inspects the ML type provided by the programmer. The Kit assumes that functions with monomorphic types are region exomorphisms; region endomorphic functions may be described using ML polymorphism, see Section 18.6.

For every C function that is called from an ML program, the order of the additional region arguments is uniquely determined by the ML result type of the function. This type must be constructed from lists, records, booleans, reals, strings, integers, and type variables.

When profiling is enabled, yet another additional argument, a program point, is passed to the C function. This argument provides allocation primitives with information about what points in the program contributes with allocation, see Section 18.4.

Examples of existing libraries that can be accessed from within ML programs include the X Window System and standard UNIX libraries providing functions such as `time`, `cp`, and `fork`. There are limitations to the scheme, however. First, because C and the Kit do not share value representations, transmitting large data structures between C and ML will often involve significant copying. Second, some C libraries require the user to set up call-back functions to be executed when specific events occur. It is not currently possible with the Kit to have a C function call an ML function.

18.1 Declaring Primitives and C Functions

The Kit conforms in large parts to the Standard ML Basis Library. Part of the functionality found in this library is programmed in C and linked to the Kit runtime system. The declarations in system dependent parts of the library use a special built-in identifier called `prim`, which is declared to have type scheme $\forall\alpha\beta.(\texttt{string} * \texttt{string} * \alpha) \rightarrow \beta$ in the initial basis. A primitive function is then declared by passing its name to `prim`. For example, the declaration

```
fun (s : string) ^ (s' : string) : string =
  prim ("concatString", "concatStringProfiling", (s, s'))
```

declares string catenation. The argument and result types are explicitly stated so as to give the primitive the correct type scheme. The first string `"concatString"` denotes a C function identifier.[1] For the example declaration, the Kit generates a call to the C function `concatString` with arguments `s` and `s'`. The C function must then of course be present at link-time; if not, the Kit will complain. The second argument to `prim` is a C function identifier to use—instead of the first—when profiling is enabled, see Section 18.4. A convenient way to declare a C function is to use the following scheme:

$$\texttt{fun}\ \mathit{vid}\ (x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau\ \texttt{=}\ \texttt{prim(}\mathit{c\_func}, \mathit{c\_funcProf}, (x_1, \ldots, x_n)\texttt{)}$$

[1]Some strings (e.g., `"="` and `":="`) are recognised and implemented in assembler by the compiler.

The result type τ must be of the form

$$\begin{array}{rl} \tau ::= & \alpha \mid \texttt{int} \mid \texttt{bool} \mid \texttt{unit} \\ \mid & \tau_1 * \ldots * \tau_n \mid \tau\ \texttt{list} \mid \texttt{real} \mid \texttt{string} \end{array}$$

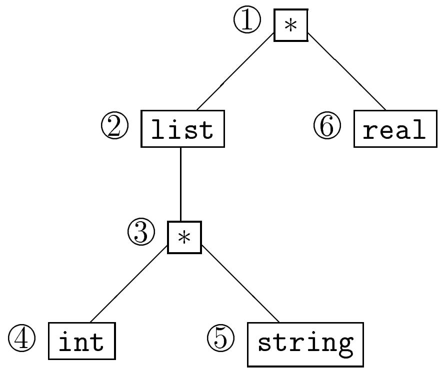
If the result type is one of α, `int`, `bool`, or `unit` then the result value can be returned in a single register. Contrary, if the result type represents an allocated value, the C function must be told where to store the value. For any type that is either `real` or a non-empty tuple type, and does 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 an allocated value that 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 auxiliary pairs
2. regions for elements (if necessary)

We now 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 a 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`, `rAddrLPairs`, `rAddrEPairs`, `rAddrEStrings` and `vAddrReal`, see Figure 18.1.

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 argument `rAddrLPairs` holds the region address for the auxiliary pairs of the list. Similarly, the arguments `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.



① `vAddrPair`

② `rAddrLPairs`

③ `rAddrEPairs`

④ Integers are unboxed

⑤ `rAddrEStrings`

⑥ `vAddrReal`

Figure 18.1: 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`.

Additional arguments holding pointers to pre-allocated space and infinite regions are passed to the C function prior to the ML arguments. Consider again the ML declaration

$$\texttt{fun}\ \mathit{vid}\ \texttt{(}x_1 : \tau_1, \ldots, x_n : \tau_n\texttt{)} : \tau\ \texttt{=}\ \texttt{prim(}\mathit{c\_func}, \mathit{c\_funcProf}, \texttt{(}x_1, \ldots, x_n\texttt{))}$$

The C function *c_func* must then be declared as

$$\texttt{int}\ \mathit{c\_func}\ \texttt{(int}\ \mathit{addr}_1, \ldots, \texttt{int}\ \mathit{addr}_m,\ \texttt{int}\ x_1, \ldots, \texttt{int}\ x_n\texttt{)}$$

where addr_1, …, addr_m are pointers to pre-allocated space and infinite regions as described above.

18.2 Conversion Macros and Functions

The runtime system provides a small set of conversion macros and functions for use by C functions that need to convert between ML values and C values. Using these conversion macros and functions for converting between representations protects you against any future change in the representation of ML values. The conversion macros and functions are declared in the header files:

```
src/Runtime/Tagging.h
src/Runtime/String.h
src/Runtime/List.h
```

18.2.1 Integers

There are two macros for converting between the ML representation of integers and the C representation of integers:[2]

```
#define convertIntToC(i)
#define convertIntToML(i)
```

To convert an ML integer `i_ml` to a C integer `i_c`, write

```
i_c = convertIntToC(i_ml);
```

To convert a C integer `i_c` to an ML integer `i_ml`, write

```
i_ml = convertIntToML(i_c);
```

The macros demonstrated here are used in the examples 2, 3, and 6 in Section 18.9.

18.2.2 Units

The following constant in the conversion library denotes the ML representation of `()`:

```
#define mlUNIT
```

18.2.3 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 C real. The following two macros are provided:

```
#define convertRealToC(mlReal)
#define convertRealToML(cReal, mlReal)
```

[2]In this release of the Kit, these macros are the identity maps, but that may change.

Converting an ML real `r_ml` to a C real `r_c` can be done with the first macro:

```
r_c = convertRealToC(r_ml);
```

Converting from a C real to an ML real (being part of the result value of the C function) is 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 18.1:

```
convertRealToML(r_c, r_ml);
```

If the ML real is part of a list element, then space must be allocated for the real before converting it. If `rAddr` identifies a region for the real, you write:

```
allocReal(rAddr, r_ml);
convertRealToML(r_c, r_ml);
```

These macros are used in the examples 3, 6 and 8 in Section 18.9.

18.2.4 Booleans

Four constants provide the values of true and false in ML and in C. These constants are defined by the following macros:[3]

```
#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 is similar to converting integers:

```
b_c = convertBoolToC(b_ml);
b_ml = convertBoolToML(b_c);
```

[3]Booleans in the Kit are tagged for historical reasons.

18.2.5 Records

Records are boxed. One macro is provided for storing and retrieving elements:

```
#define elemRecordML(recAddr, offset)
```

An element can be retrieved from a record `rec_ml` by writing

```
e_ml = elemRecordML(rec_ml, offset);
```

where the first element has `offset` 0. An element `e_ml` is stored in an ML record `rec_ml` by writing

```
elemRecordML(rec_ml, offset) = e_ml;
```

Two specialized versions of the `elemRecordML` macro are provided for pairs:

```
#define first(x)
#define second(x)
```

If the record is to be part of a list element then it is necessary to allocate the record before storing into it. This allocation is done with the macro

```
#define allocRecordML(rAddr, size, vAddr)
```

where `rAddr` denotes a region (i.e., a pointer to a region descriptor), `size` is the size of the record (i.e., the number of components), and `vAddr` 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 `vAddr` as argument.

The above macros are used in examples 8, 9 and 7 in Section 18.9.

18.2.6 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, int exn);
StringDesc *convertStringToML(int rAddr, char *cStr);
```

An ML string `str_ml` is converted to a C string `str_c` in already allocated storage of size `size` bytes by writing

```
convertStringToC(str_ml, str_c, size, exn);
```

where `exn` is some ML exception value (see Section 18.3) to be raised if the ML string has size greater than `size`.

A C string is converted to an ML string in the region denoted by `rAddr` by writing

```
str_ml = convertStringToML(rAddr, str_c);
```

The following function returns the size of an ML string:

```
int sizeString(StringDesc *str);
```

These macros are used in the examples 7 and 5 in Section 18.9.

18.2.7 Lists

Lists are always allocated in infinite regions. A list uses, as a minimum, one region for the auxiliary pairs of the list, see Figure 5.1 on page 53.

We shall now show three examples of manipulating lists. The first example traverses a list. Consider the following C function template:

```
void traverse_list(int ls) {
  int elemML;
  for ( ; 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 `ls`. The example uses a simple loop to traverse the list. The parameter `ls` points at the first constructor in the list. Each time we have a `CONS` constructor we also have an element, see Figure 5.1. The element can be retrieved with the `hd` macro. One retrieves the tail of the list by using the `tl` macro.

The following four macros are provided in the `src/Runtime/List.h` header file:

```
#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 mk_list_backwards(int pairRho) {
  int *resList, *pair;
  makeNIL(resList);
  while (/*more elements*/) {
    ml_elem = ...;
    allocRecordML(pairRho, 2, pair);
    first(pair) = (int) ml_elem;
    second(pair) = (int) resList;
    makeCONS(pair, resList);
  }
  return (int) resList;
}
```

First, we create the `NIL` constructor, which marks the end of the list. Then, 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. (In this release of the Kit, the `makeCONS` macro simply assigns its second argument the value of its first argument.) In the example, we have assumed that the elements are unboxed, thus, 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 mk_list_forwards(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(pair, cons);
  res = (int) cons;

  while (/*more elements*/) {
    ml_elem = ...
    allocRecordML(pairRho, 2, temp_pair);
    first(temp_pair) = (int) ml_elem;
    makeCONS(temp_pair, cons);
    second(pair) = (int) cons;
    pair = temp_pair;
  }
  makeNIL(cons);
  second(pair) = (int)cons;
  return res;
}
```

We create the `CONS` constructor and pair for the first element and return a pointer to the `CONS` constructor (the pair) 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 (`temp_pair`) because we have to update the pair for the previous element. The second component of the last pair contains the `NIL` constructor and thus denotes the end of the list.

The two macros `makeCONS` and `makeNIL` are provided in the `List.h` header file:

```
#define makeNIL(rAddr, ptr)
#define makeCONS(rAddr, pair, ptr)
```

18.3 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, however. As an example, consider the ML declaration:

```
exception Exn
fun raiseif0 (arg : int) : unit =
  prim("raiseif0", "raiseif0", (arg, Exn))
```

If we want the function `raiseif0` to raise the exception value `Exn` if the argument (`arg`) is 0 then we use the function `raise_exn` provided by the runtime system, by including the header file `src/Runtime/Exception.h`. The C function `raiseif0` may be declared thus:

```
void raiseif0(int i_ml, int exn) {
  int i_c;
  i_c = convertIntToC(i_ml);
  if (i_c = 0) raise_exn(exn);
  return;
}
```

There is no need to make the function return the value `mlUNIT`; in case the type of the return value is `unit` then the Kit automatically inserts code for returning the ML value `()` after the call to the C function.

The implementation of the `raise_exn` function for the native backend differs from that of the C backend; in the native backend, `raise_exn` never returns, whereas in the C backend, `raise_exn` sets a flag, which is checked (by code generated by the Kit) when the function returns. Thus, to obtain the same behaviour with the two backends, one should avoid side effecting expressions and statements between function returns and calls of `raise_exn`.

Exceptions are used in examples 6 and 7 in Section 18.9.

18.4 Program Points for Profiling

To support profiling, the programmer must provide special profiling versions of those C functions that allocate space in regions (i.e., that take regions as additional arguments). If profiling is enabled and at least one pointer to a region is passed to the C function then also a program point that represents the call to the C function is passed. The program point is used by the C function when allocating space in regions, as explained in Section 18.4. 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*)

No special version of the C function is needed if it does not allocate into infinite regions; in this case, the same C function identifier can be passed as the first and second argument to `prim`.

A program point passed to a C function is an integer; it identifies the allocation point that represents the C call in the program, see Chapter 16.

The runtime system provides 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);
```

Here is the profiling version of the C function `mk_list_backwards`:

```
int mk_list_backwardsProf(int pairRho, int pPoint) {
  int *resList, *pair;
  makeNIL(resList);
  while (/*more elements*/) {
    ml_elem = ...;
    allocRecordMLProf(pairRho, 2, pair, pPoint);
    first(pair) = (int) ml_elem;
    second(pair) = (int) resList;
    makeCONS(pair, resList);
  }
  return (int) resList;
}
```

The 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.

18.5 Storage Modes

As described in Chapter 12 on page 101, 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 header file `src/Runtime/Region.h` of the runtime system provides a macro `is_inf_and_atbot`, 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 also provided by the runtime system in the header file `src/Runtime/Region.h`, can be used to reset a region. Consider again the `mk_list_backwards` example. If the `atbot` bit of the region for the list is set, then this region can be reset prior to constructing the list:

```
int mk_list_backwards(int pairRho) {
  int *resList, *pair;
  if (is_inf_and_atbot(pairRho)) resetRegion(pairRho);
  makeNIL(resList);
  ...
}
```

The C programmer should be careful not to reset regions that potentially 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 that 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 try to access the argument(s).

18.6 Endomorphisms by Polymorphism

Until now, we have seen examples only of C functions that are region exomorphic, that is, functions that, in general, write their result into regions that are different from those in which the arguments reside.

A region endomorphic function has the property that the result of calling the function is stored in the same regions that hold the arguments to the function. Region endomorphic functions are useful when the result of the function shares with parts of the arguments. Consider the C function

```
int select_second(int pair) {
 return second(pair);
}
```

which selects the second component of `pair` (cast to an integer); the identifier `second` is defined in the header file `Tagging.h` by the macro definition

```
#define second(x)  (*((int *)(x)+1))
```

Now, for the Kit to make correct, that is safe, decisions about when to de-allocate regions, the endomorphic properties of a C function must be expressed in the region-annotated type scheme for value identifiers to which the C function is bound. The programmer can tell the Kit about region endomorphic behaviour of a C function by using type variables. For example, here is an ML declaration that binds a value identifier `second` to the C function `select_second`:[4]

```
fun second(pair : 'a * 'b) : 'b =
  prim("select_second", "select_second", pair)
```

The Kit associates the following region-annotated type scheme to the value identifier `second`:

$$\forall\alpha_1\alpha_2\rho_1\rho_2\rho_3\epsilon.((\alpha_1,\rho_1)*(\alpha_2,\rho_2),\rho_3)\xrightarrow{\epsilon.\{\mathbf{get}(\rho_3)\}}(\alpha_2,\rho_2)$$

Notice that the region-annotated type scheme expresses the region endomorphic behaviour of the C function.

18.7 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 appropriate header files.)

As an example, the file `kitdemo/my_lib.c` holds a set of example C functions. This file is compiled by typing (from the shell)

```
gcc -c my_lib.c
```

in the `kitdemo` directory. Now, to compile the file to work with profiling, type

```
gcc -DPROFILING -o my_lib_prof.o -c my_lib.c
```

[4] Project `kitdemo/select_second.pm`. The C file `select_second.c` must be compiled (using `gcc`) to form the object file `select_second.o` before the project can be compiled; if you forget, the Kit will complain.

```
import my_lib.o : my_lib_prof.o
in
   my_lib.sml
   test_my_lib.sml
end
```

Figure 18.2: Linking with external object files. The external object file `my_lib.o` is linked in when forming the executable file `run`. When profiling is enabled, the external object file `my_lib_prof.o` is used instead.

The project `my_lib.pm`, which is listed in Figure 18.2, expresses that, when profiling is disabled, the object file `my_lib.o` is linked in to form the executable file `run`. When profiling is enabled, the object file `my_lib_prof.o` is linked in instead. (The `: my_lib_prof.o` part of the project is required only when profiling is enabled.)

It may be necessary to modify the string entry `link with library` in the Kit menu `Control` so as to link in additional C libraries.

18.8 Auto Conversion

For C functions that are simple, in a sense that we shall soon define, the Kit can generate code that automatically converts representations of arguments from ML to C and representations of 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_auto(base : int, n : int) : int =
  prim ("@power_auto", "@power_auto", (base, n))
```

The power function may then be implemented in C as follows:

```
int power_auto(int base, int n) {
  int p;
  for (p = 1; n > 0; --n) p = p * base;
  return p;
}
```

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 works also when profiling is enabled.

The example shown here is example 4 of Section 18.9; it is part of the `my_lib.pm` project.

18.9 Examples

Several example C functions are located in the file `kitdemo/my_lib.c`. The project `kitdemo/my_lib.pm`, which is listed in Figure 18.2, makes use of these functions.

The source file `my_lib.sml`, which is part of the `my_lib.pm` project, contains the following ML declarations:

```
fun power(base : int, n : int) : int =
  prim ("power", "power", (base, n))

fun power_auto(base : int, n : int) : int =
  prim ("@power_auto", "@power_auto", (base, n))

fun power_real (base : real, n : int) : real =
  prim ("power_real", "power_real", (base, n))

fun print_string_list (string_list : string list) : unit =
  prim ("print_string_list", "print_string_list", string_list)

exception Power
fun power_exn (base : real, n : int) : real =
  prim ("power_exn", "power_exn", (base, n, Power))

exception DIR
fun dir (directory : string) : string list =
  prim ("dir", "dirProf", (directory, DIR))

fun real_list () : real list =
  prim ("real_list", "real_listProf", ())
```

```
fun change_elem (p : int*string) : string*int =
  prim ("change_elem", "change_elem", p)
```

The implementation of each of the C functions is summarized below (see the files `my_lib.c` and `my_lib.sml` in the `kitdemo` directory for detailed comments.)

Example 2 The `power` function shows how to convert integers with the macros `convertIntToC` and `convertIntToML`.

Example 3 The `power_real` function shows how to convert reals with the macros `convertRealToC` and `convertRealToML`.

Example 4 The `power_auto` function shows the use of auto conversion, which allows for easy linking to certain C functions.

Example 5 The `print_string_list` example shows how to traverse a list of strings. The technique can easily be adobted to other data structures (e.g., to lists of lists of strings).

Example 6 The `power_exn` function shows how an exception can be raised from a C function. Notice 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.

Notice also that we check the infinite regions for resetting at the start of the C function. The checks should 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 ML program.

Example 9 Function change_elem shows the use of the 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 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.

Chapter 19

Changes from Version 2

19.1 Modules and Separate Compilation

The most important development since Version 2 is the ability to compile Modules and the discipline of separate compilation. A distinguished feature of the way modules are compiled is that module constructs do not give rise to *any* code, so there is no runtime overhead in using modules. See Chapter 15.

19.2 Standard Basis Library

The ML Kit now contains (a large portion of) the Standard Basis Library, based on the Moscow ML version of the Library. To see exactly what parts of the Standard Basis Library are supported, consult the project file `basislib.pm` located in the directory `kit/basislib`.

19.3 Scalability

The Kit now compiles fairly large programs, including Hafnium's AnnoDomini (58.000 lines of SML) and the ML Kit itself (around 80.000 lines).

19.4 New Match Compiler

The pattern compiler has been rewritten, based on Sestoft's method [Ses96], which also is the basis of the Moscow ML match compiler.

19.5 New StatObject Module

The Kit contains a module, StatObject, which implements the semantic objects of the static semantics of the Core. Origianally, this was a very clean and very inefficient implementation of the Defininion. In Version 2 of the Kit, StatObject was replaced by an imperative and efficient, but complicated module. In Version 3, StatObject uses a clean, efficient and imperative implementation of StatObject. This is particularly useful for those that want to reuse the front-end of the Kit for other purposes.

19.6 More Efficient Representation of Lists

List constructors are now represented unboxed, that is, the least significant bits of a list value is used to distinguish between `nil` and a pointer to a pair (`::`) holding the head and the tail of the list. Thus, a list takes up only one region (for the auxiliary pairs) plus any regions for the elements of the list. Consult Chapter 5 for details.

Further Reading

Most of the following documents may be found from the ML Kit home page.

[BTV96] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183. ACM Press, January 1996.

[EH95] Martin Elsman and Niels Hallenberg. An optimizing backend for the ML Kit using a stack of regions. Student Project 95-7-8, Department of Computer Science, University of Copenhagen (DIKU), July 5 1995.

[KO96] Martin Koch and Tommy Højfeld Olesen. Compiling a higher-order call-by-value functional programming language to a RISC using a stack of regions. dissertation 96-10-5. Master's thesis, 1996.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[Ses96] Peter Sestoft. *Partial Evaluation*, volume 1110, chapter ML pattern match compilation and partial evaluation, pages 446–464. Springer-Verlag, February 1996.

[TB98] Mads Tofte and Lars Birkedal. A region inference algorithm. *Transactions on Programming Languages and Systems (TOPLAS)*, 20(4):734–767, July 1998.

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

[TT97] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

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.