A Region Profiler for a Standard ML compiler based on Region Inference

Niels Hallenberg (faxe@diku.dk)

June 14, 1996

Abstract

In this report we present the region profiler used in the ML Kit (a Standard ML compiler that uses region inference, [7]). The profiler is based on work done by Colin Runciman and David Wakeling ([17]) on a heap profiler for lazy functional programs. The profiler can be used to reduce memory usage of programs compiled with the ML Kit. Hints and examples on writing programs in a more region friendly way are given.

Supervisor: Mads Tofte

Contents

1 Introduction

Functional languages like Standard ML [12, 11] normally use a garbage collector to reclaim some of the unused memory. Another approach is to use a region-based memory management scheme where allocation and deallocation of regions should give a more eager reuse of memory [23, 21, 22].

Region Inference is a static analysis which only gives an approximation of the dynamic memory behaviour of the program. In some cases the analysis gives a bad approximation and memory allocations which could be local are treated as global allocations with the result that memory is not eagerly reused. In that case we say the program has a *space-leak*.

Experiments on a working Standard ML compiler using region inference (from now on The ML Kit [6, 7][1]) show that, for a large number of programs, memory is reused eagerly, but for some programs, the programmer has to change the programs to avoid space-leaks. Because the ML Kit does not combine garbage collection and region inference,[2] a space-leak can result in an unreasonably large memory use.

To make it easier to identify space-leaks and predict memory demands for the result program, we have implemented a region profiler in the ML Kit. When the compiled program is executed, the region profiler will collect data about the dynamic memory behaviour. Afterwards the data can be viewed graphically, making the transformation of the original space-leaking ML program into a more space-efficient ML program (with respect to region inference) easier. We expect the following problems to be exposed by the profiler.

- The size of regions. How are the sizes of regions distributed? Besides regions holding large data structures, we expect to have a large number of small regions with short live ranges. This will indicate that memory is eagerly reused.
- How many allocation points allocate in the same region? Many allocation points in one region indicates that the live range for that region is large. We expect the average live range for regions to be small.
- What `letregion` expressions give a large number of simultaneously active run-time regions?
- Are there regions at run-time (allocated on the region stack) that are never used? This will indicate that the analysis removing zero sized regions can be improved.
- How is the machine stack used? The ML Kit uses the stack to store exception handlers, live values at function calls, finite regions and descriptors for infinite regions. By inspecting the stack we can, for example, see the effect of making a recursive call tail recursive. A tail call does not push live values (etc.) on the stack.
- Is space allocated for infinite regions used effectively; how much is waste due to partitioning into region pages? Does the region page size influence the amount of waste?
- How well do additional analyses, like the storage mode analysis, work?

The region profiler uses a program that, with a profile data file as input, produces profile graphs. The program that produces the graphs is based on the heap profiler distributed with the `nhc` Haskell compiler [15]. In [16], Colin Runciman and Niklas Röjemo describe the profiler in detail.

The ML Kit compiles to both HP's PA-RISC architecture ([14, 13]) and to ANSI-C code. In this text we only focus on the ML Kit version with the PA-RISC back end. Profiling with the ANSI-C back end is similar.

Section 3 gives an introduction to region inference. It is shown how the dynamic memory behaviour can be determined (predicted) for the result program, by inspecting the region annotated lambda program. An example on how the region profiler is used is given in section 4. A detailed description of the profiler is in section 5, and section 7 contains valuable information about the implementation if one has to change the profiler. Section 6 contains hints about how to program *space–efficiently* with regions. In the next section we start with a brief overview of profiling in the ML Kit.

We assume that the reader is acquainted with Standard ML [12, 11] and the notation used in the typed call-by-value lambda calculus. It should not be necessary to be well of in the theory of region inference but more information on the theory can be found in [21, 23, 22, 2]. We will often refer to [7] because it contains a detailed description of how region inference is implemented in the ML Kit. Implementational details of the ML Kit can be found in [10, 5, 6]. Experience with the ML Kit is reported in [4, 23].

[1] This report is based on the ML Kit with Regions version 29g.

[2] Currently work is being done on combining region inference and garbage collection in the ML Kit.

2 Profiling in the ML Kit

Profiling in the ML Kit is done in several phases. The result of profiling a program is a set of *profile graphs* showing the dynamical memory use of the executed *target program.* We will use the graphs to locate space-leaks, and try to remove them by changing the *ML source program.* To do that we have to make a connection between the graphs and the ML source program; with a connection we can see where in the source program memory is used.

To be able to change the ML source program in a space friendly way, we also need some advice from the region inference analyses, and especially where and why the analyses must make conservative decisions.

The profiler outputs the ML source program as a *region annotated lambda program*, which has been optimised by the ML Kit. The lambda program represents the ML source program, and is annotated with information from region inference and the additional *region representation* analyses. Region representation inference (explained in [7]) consists of several analyses applied after region inference. Two of the analyses, *multiplicity inference* and *storage mode analysis* are relevant when profiling.

Each allocation point in the lambda program is annotated with a *program point* which gives the connection to the profile graphs.

Because *region variables* (explained in section 3) are passed around at runtime, we also need a *region flow graph.* We believe that with the following four tools, we are in a good position to see if a ML source program contains a space leak, and if so, change the source program in a space efficient way.

- a region annotated lambda program.
- a data file containing profiling data; called the *profile data file.*
- a region flow graph.
- a program to generate profile graphs with the profile data file as input. We call it the *graph generator.* The program is named `rp2ps` ("`r`egion `p`rofile `to` `p`ost`s`cript").

Profiling is sketched in figure 1.

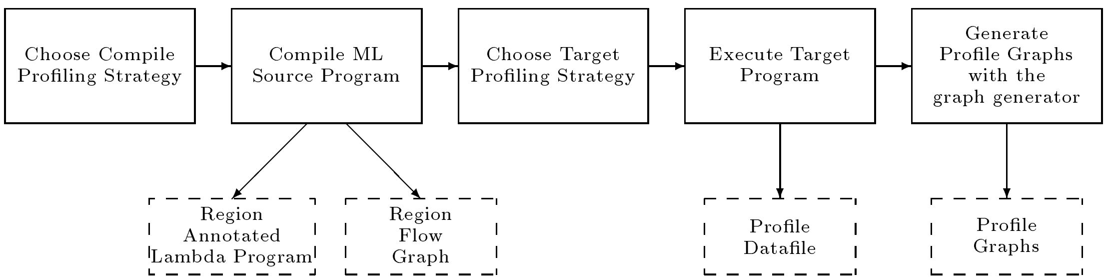


Figure 1: *Overview of the ML Kit profiler. Dotted boxes represent output from the profiler. By Compile Profiling Strategy we mean the profiler options chosen when compiling the ML source program. The profiler options given to the target program determines the Target Profiling Strategy.*

We have included some sections discussing region inference and the additional representation analyses because some knowledge is necessary to read the output from the profiler.

3 Region inference

Programs compiled using region inference allocate all values in abstract boxes called regions. Regions are allocated and deallocated while the program is running. Region inference ensures that the region manipulation follows a stack discipline.[3] In the ML Kit the region stack is split into a *finite* and an *infinite* region stack.

All regions in the finite region stack only hold one value[4] of fixed size (inferred at compile time) and are therefore implemented on a one-dimensional machine stack. There is no overhead in allocating objects in finite regions compared to normal machine stack allocations. We note that values of size one word are not stored on the stack, but kept in machine registers or machine words. In the ML Kit integers and booleans have size one word. Reals are not stored in floating point registers but put in finite and infinite regions.

The size of regions pushed onto the infinite region stack can not be determined at compile time. They are said to be of infinite size and are implemented by mapping the one-dimensional machine memory heap into a two-dimensional memory heap as shown in figure 2.

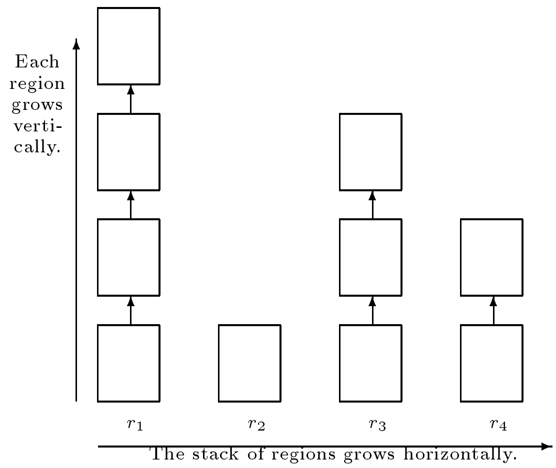


Figure 2: *A stack of infinite regions. Region r_4 is the topmost region. Each region consists of several region pages of fixed size connected by pointers. This framework gives some overhead when allocating objects. Mainly, an extra test is necessary to see if the object can be allocated in the topmost region page of the region or it is necessary to allocate a new region page.*

Infinite regions are necessary because of recursive data types such as lists. In a conventional imperative language allocation of values of unbounded size (e.g. lists) are done by the programmer. Having only objects of known size enable conventional compilers to store all objects on the machine stack.

3.1 Region annotated lambda program

The region inference algorithm [21] used in the ML Kit transforms a typed lambda program into a *region annotated* lambda program where allocation and deallocation of regions are explicit. In section 3.3 we show the region annotated terms from [7] and explain some of the main ideas when inferring the type for a region annotated term. The rules are used in section 6 to show how several ML constructs influence the memory consumption of the target program.

The region annotated lambda program makes it possible to predict how memory is managed at runtime. However, as we will see, *region polymorphic functions* [21, 22] are obstacles when we want to find regions in which values are allocated and deallocated. A region polymorphic function is a function which beside its argument is passed some regions in which values are to be stored. In the ML Kit a region is represented as a pointer and therefore passed in a machine word.

Moreover, in the ML Kit, region inference is supplemented with *region representation analyses* [7]. Among others, these analyses comprise a *storage mode analysis* which annotates each allocation with a storage mode `attop` or `atbot`. An allocation point with storage mode `atbot` will reset the region where the object is going to be allocated. Allocated memory is deallocated bringing the region size to zero. The object is then allocated bringing the region size to one object. Storage mode `attop` will just allocate the object in the region, increasing the region size with one object.

As an example on how to read a region annotated lambda program we use the function `longlist`:

[3] In [2] A. Aiken, M. Fähndrich and R. Levien present a region based analysis where the lifetime of regions is shortened giving a more eager memory reuse. However, in their framework, region allocation and deallocation does not follow the stack discipline.

[4] In this text we will use the words value and object interchangely.

```
fun longlist 0 = []
  | longlist n = n :: longlist (n-1)
```

with the following region annotated lambda program generated by the ML Kit.

```
0   let fun longlist(r5 : inf, r6 : inf, var3)(* at r4  pp2  *) =
1    (case  var3 of
2       0 => nil sat r5 pp3 with sat r6
3     | _ => let val v24 =
4              let val v26 =
5                letregion r8 : 2
6                in
7                  longlist(* atbot r8  *)(sat r5 ,sat r6 ,prim(-,[var3,1]))
8                                    (* region vector for longlist pp4  *)
9                end (*r8 *)
10              in
11                (var3,v26) attop r6  pp6
12              end
13            in
14              ::(v24)attop r5  pp7
15            end) (* shared region clos. atbot r4  pp2  *)
16  in
17    ...
18  end
```

The region annotated lambda program shows that `longlist` is region polymorphic with two infinite regions (`r5` and `r6`). All functions in the ML Kit takes one argument and for `longlist` the argument is passed in `var3` (in this case an integer).

After region inference the region representation analyses infer the size of each region by first inferring the number of times a value is put into each region [7, Multiplicity inference] and afterwards finding the precise size of objects allocated in regions [7, Physical size inference]. Regions inferred to hold more than one object get size *infinity* (written `inf`, line 0), and regions inferred to hold zero or one object the precise size of that object is written. For instance, region `r8` is inferred to hold at most one object of size two words, line 5. Regions are allocated by the letregion construct. It puts infinite regions on the infinite region stack and finite regions on the finite region stack.

A *shared closure* [5] for `longlist` is allocated `atbot` in region `r4` (line 15). The shared closure contains all free variables to `longlist` but no code pointer.[5] Region `r5` holds the `CONS` and `NIL` cells and `r6` holds the pair cells which are used to construct the result list [10, page 20][6].

If `var3` is 0 (line 2) we have `nil sat r5 with sat r6`, saying that `nil` is allocated *somewhere at* in region `r5`. Somewhere at is a third storage mode given by the storage mode analysis meaning that the actual storage mode (`atbot` or `attop`) must be checked at runtime. We cannot check it statically, because region `r5` is given as an argument at runtime and can be one of many possible regions where we can store `atbot` in some and `attop` in others. Region `r6` is also checked for resetting because we have a `NIL` constructor in connection with the construction of a list (`r6` is an auxiliary region, see section 3.7).

If `var3` > 0 (line 3–15) we call `longlist` recursively by calculating a new argument (`prim(-, [var3,1])`) and we allocate a new *region vector*[7] [5] in a local region `r8` containing the actual region arguments `r5` and `r6`.

After returning from `longlist`, variable `v26` contains the list constructed so far, and region `r8` (containing the region vector for the returned call) is deallocated. A pair `(var3,v26)` is allocated `attop` in region `r6` and a `CONS` cell for `::` that is applied to the pair is allocated `attop` in region `r5`.

Each allocation point in the region annotated lambda program has a program point written `pp#`.

3.2 Region annotated terms

The target language of the region inference algorithm in the ML Kit extends the language presented in [7]. We will not present the language in detail but comment on several constructs. The motivation for this presentation is that we will use the inference rules to show why some programs allocates a lot of memory and why others do not (see section 6).

The language of expressions we use resembles the language in the ML Kit (printed as the region annotated lambda program). At this stage however, all values are put in regions, and there is not differentiated between

[5]Shared closures are built for `letrec` bound functions because they are always "fully applied". By "fully applied" we mean that both argument and actual regions are passed to the function when applied. All letrec bound function names are known by the compiler, so no code pointer is necessary. The union of the free variables of the functions are put in the same shared closure for all functions bound in the same `letrec` expression.

[6]In the ML Kit lists are constructed by having `NIL` and `CONS` cells in one region and pairs with a pointer to the next `CONS` or `NIL` cell and a value in another region.

[7]When calling a region polymorphic function, the actual regions are passed in a region vector, which is a record of regions allocated in memory. Implementing multiple argument and result passing in the ML Kit would especially improve polymorphic function application by passing region actuals as arguments.

finite and infinite regions. It is first in the region representation analyses, that word sized regions are removed, and regions are split in finite and infinite regions. The syntactic classes we use are:

$a ::= \texttt{at}\ \rho$
$\vec{a} ::= a_1, \ldots, a_n \mid \epsilon$
$b ::= \rho$
$\vec{b} ::= b_1, \ldots, b_n \mid \epsilon$

$e ::= x_{il} \mid f_{il}\ [\vec{a}]\ a\ e \mid (\lambda^{\epsilon.\varphi} x : \mu.e)\ a \mid \texttt{false}\ a \mid \texttt{true}\ a$
$\mid \texttt{letrec}\ f : (\pi, \rho)\ [\vec{b}]\ x\ a = e_1\ \texttt{in}\ e_2\ \texttt{end}$
$\mid e_1\ e_2$
$\mid \texttt{let}\ x : (\sigma, \rho)\ = e_1\ \texttt{in}\ e_2\ \texttt{end}$
$\mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$
$\mid \texttt{exception}\ \mathit{excon}\ \langle\tau\rangle\ a\ \texttt{in}\ e\ \texttt{end}$
$\mid (\texttt{raise}\ e) : \mu$
$\mid e_1\ \texttt{handle}\ e_2$
$\mid \texttt{prim}\ \mathit{name}\ e_1, \ldots, e_n$
$\mid \texttt{letregion}\ b\ \texttt{in}\ e\ \texttt{end}$

In for instance, **true** a we have an *allocation directive* (a) which says that something is allocated in region ρ. Region polymorphic functions are applied to both actual regions ($[\vec{a}]$) and an argument (e) at runtime. The *formal region variables* of a letrec bound function are written as a vector with zero or more region binders ($[\vec{b}]$).

The set of exception constructors ExCon is ranged over by *excon*. The type τ is optional in the exception expression; τ is used to give the type of the argument to unary exceptions. The language contains several primitive operations taking one or more arguments. A special construct **prim** *name* $e_1, \ldots, e_n$ incorporates the primitives into the language where *name* is the name of the primitive.

We now show parts of a type system (also found in [7]) equivalent to the type system used by the region inference algorithm. Semantic objects used are *type variables* ($\alpha \in$ TyVar), *effect variables* ($\epsilon \in$ EffVar), *region variables* ($\rho \in$ RegVar), *effects* ($\varphi \in$ Eff) and *atomic effects* ($\eta \in$ AtomEff). Region variables represent regions. Effect variables represent effects. An atomic effect can either be a *put effect* (a value is written into region ρ, $\mathbf{put}(\rho)$), a *get effect* (a value is accessed in region ρ, $\mathbf{get}(\rho)$) or an effect variable. An effect (φ) is a finite set of effects. We will often abbreviate, for instance $\mathbf{get}(\rho_1), \ldots, \mathbf{get}(\rho_n)$ to $\mathbf{get}(\rho_1, \ldots, \rho_n)$.

We have *simple types* τ, *type and places* μ, *simple type schemes* σ and *compound type schemes* π.

$\tau ::= \mu \xrightarrow{\epsilon.\varphi} \mu' \mid \alpha \mid \texttt{bool} \mid \mu\ \texttt{ref} \mid \texttt{exn} \mid \texttt{unit}$
$\mu ::= (\tau, \rho)$
$\sigma ::= \tau \mid \forall\alpha.\sigma \mid \forall\epsilon.\sigma$
$\pi ::= \underline{\tau} \mid \forall\alpha.\pi \mid \forall\epsilon.\pi \mid \forall\rho.\pi$

The function type $(\tau_1, \rho_1) \xrightarrow{\epsilon.\varphi} (\tau_2, \rho_2)$ says that the argument is expected to have type τ_1 and the argument is in region ρ_1. The function returns a value of type τ_2 put into region ρ_2. The effect, φ, of applying the function is the effect of evaluating the body of the function. A value with type (τ, ρ) **ref**, is a reference to a value of type τ held in region ρ.

The inference system infers judgements of the form $TE \vdash e : \mu, \varphi$, which is read, "in type environment TE, the expression e has type and place μ and effect φ." The type environment maps program variables to pairs of the form (σ, ρ) or (π, ρ).

The type system does not allow region polymorphism in connection with let expressions so therefore we have both simple and compound type schemes. Type $\underline{\tau}$ is used to insure that variable x in rule 1 (shown below) is let bound and variable f in rule 9 is letrec bound.

3.3 Inference Rules

A variable x is looked up in TE and instantiated to a simple type τ via substitution S.

$$\frac{TE(x) = (\sigma, \rho) \qquad \sigma \geq \tau\ \mathit{via}\ S}{TE \vdash x_{il(S)} : (\tau, \rho), \emptyset} \tag{1}$$

A substitution S maps region variables to region variables, type variables to types and effect variables to arrow effects organized as a triple. S is written

$$S = (\{\rho_1 \to \rho'_1, \ldots, \rho_l \to \rho'_l\}, \{\alpha_1 \to \alpha'_1, \ldots, \alpha_m \to \alpha'_m\}, \{\epsilon_1 \to \epsilon'_1.\varphi'_1, \ldots, \epsilon_n \to \epsilon'_n.\varphi'_n\})$$

For any compound (or simple) type scheme

$$\pi = \forall\rho_1, \ldots, \rho_l, \alpha_1, \ldots, \alpha_m, \epsilon_1, \ldots, \epsilon_n.\underline{\tau}$$

we say that τ is an *instance of* π ($\pi \geq \tau$) via S if $S(\pi) = \tau$. For each use of a program variable (x_{il} or f_{il} in $f_{il}\ [\vec{a}]\ e$), the program variable is decorated with the *instance list* of substitution S;

$$il(S) = ([\alpha'_1, \ldots, \alpha'_l], [\rho'_1, \ldots, \rho'_m], [\epsilon'_1.\varphi'_1, \ldots, \epsilon'_n.\varphi'_n]).$$

The variable x in rule 1 is let bound and therefore not region polymorphic. There is no effect when looking up the variable in TE. It is first when accessing x that we have a get effect, hence the effect $\emptyset$ in 1.

Constants are elaborated by

$$\frac{}{TE \vdash \texttt{true at}\ \rho : (\texttt{bool}, \rho), \{\mathbf{put}(\rho)\}} \tag{2}$$

and

$$\frac{}{TE \vdash \texttt{false at}\ \rho : (\texttt{bool}, \rho), \{\mathbf{put}(\rho)\}.} \tag{3}$$

We create a value, hence we have a put effect into region ρ. Other constants have similar rules.

A conditional is elaborated by

$$\frac{TE \vdash e : (\texttt{bool}, \rho), \varphi \qquad TE \vdash e_1 : \mu, \varphi_1 \qquad TE \vdash e_2 : \mu, \varphi_2}{TE \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \mu, \{\mathbf{get}(\rho)\} \cup \varphi \cup \varphi_1 \cup \varphi_2.} \tag{4}$$

The two branches have same type and place. This can restrict the choice of places in the type of e_1 and e_2. This restriction also influence the way in which lambda abstractions are elaborated. The rule is easily extended to switch constructs.

A lambda abstraction is elaborated by

$$\frac{TE + \{x : \mu_1\} \vdash e : \mu_2, \varphi \qquad \varphi \subseteq \varphi'}{TE \vdash (\lambda^{\epsilon.\varphi'} x : \mu_1.e)\ \texttt{at}\ \rho : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \rho), \{\mathbf{put}(\rho)\}.} \tag{5}$$

The effect, φ (possibly extended to φ'), of evaluating the body is moved up as an *arrow effect* (also called the *latent* effect). The effect is saved on the arrow and first brought into effect when the function is applied. The effect variable ϵ is a representative for the effect φ'. We have a put effect because a closure is created and stored in region ρ.

> **Example:** Increasing the latent effect can be necessary when elaborating an if construct with two lambda abstractions in the branches. For instance, the expression
>
> $$\texttt{if true at}\ \rho_1\ \texttt{then}\ (\lambda x.\texttt{true at}\ \rho_2)\ \texttt{at}\ \rho_3\ \texttt{else}\ (\lambda x.\texttt{false at}\ \rho_4)\ \texttt{at}\ \rho_5$$
>
> is elaborated by rule 4 and 5. We have
>
> $$TE \vdash \texttt{true at}\ \rho_1 : (\texttt{bool}, \rho_1), \{\mathbf{put}(\rho_1)\}$$
> $$TE \vdash (\lambda x.\texttt{true at}\ \rho_2)\ \texttt{at}\ \rho_3 : ((\alpha_1, \rho_6) \xrightarrow{\epsilon_1.\{\mathbf{put}(\rho_2)\} \cup \varphi'_1} (\texttt{bool}, \rho_2), \rho_3), \{\mathbf{put}(\rho_3)\}$$
> $$TE \vdash (\lambda x.\texttt{false at}\ \rho_4)\ \texttt{at}\ \rho_5 : ((\alpha_2, \rho_7) \xrightarrow{\epsilon_2.\{\mathbf{put}(\rho_4)\} \cup \varphi'_2} (\texttt{bool}, \rho_4), \rho_5), \{\mathbf{put}(\rho_5)\}.$$
>
> When unifying in rule 4 we set $\varphi'_1 = \{\epsilon_2, \mathbf{put}(\rho_4)\}$ and $\varphi'_2 = \{\epsilon_1, \mathbf{put}(\rho_2)\}$.
> When moving an arrow effect into an effect (as in φ'_1) the arrow effect is split into an effect variable and an effect. They are inserted as two entities.

An application is elaborated by bringing the latent effect of the function into effect.

$$\frac{TE \vdash e_1 : (\mu_1 \xrightarrow{\epsilon.\varphi_0} \mu_2, \rho), \varphi_1 \qquad TE \vdash e_2 : \mu_1, \varphi_2}{TE \vdash e_1\ e_2 : \mu_2, \varphi_0 \cup \varphi_1 \cup \varphi_2 \cup \{\epsilon, \mathbf{get}(\rho)\}} \tag{6}$$

We access the closure in region ρ hence $\mathbf{get}(\rho)$.

Effect and type polymorphism, but not region polymorphism, is accepted in let constructs.

$$\frac{TE \vdash e_1 : (\tau_1, \rho), \varphi_1 \qquad \sigma = \forall \vec{\alpha}\vec{\epsilon}.\tau \qquad \mathrm{fv}(\vec{\alpha}, \vec{\epsilon}) \cap \mathrm{fv}(TE, \varphi_1) = \emptyset \qquad TE + \{x : (\sigma, \rho)\} \vdash e_2 : \mu, \varphi_2}{TE \vdash \texttt{let}\ x : (\sigma, \rho) = e_1\ \texttt{in}\ e_2\ \texttt{end} : \mu, \varphi_1 \cup \varphi_2} \tag{7}$$

Because x has scope e_2 then, as a minimum, the region holding the value bound to x is alive around the let construct. We use $\mathrm{frv}(A)$ to denote the set of free region variables in A, and $\mathrm{fv}(A)$ to denote the set of free region, type and effect variables in A.

Example: The sequence $(e_1; e_2; e_3)$ is by the ML Kit front end translated into nested let constructs equivalent to the program fragment at the left.

```
let                    let
  val _ = e1             val a = e1
in                     in
  let                    let
    val _ = e2             val b = e2
  in                     in
    e3                     e3
  end                    end
end                    end
```

Regions holding the result of e_1 and e_2 are first deallocated after evaluation of e_3, even though the result of e_1 and e_2 are not used by e_3. It seems possible to have the ML Kit insert local letregions on regions holding the result of e_1 around e_1, and as well for e_2. Note that it is not in general possible to deallocate the regions holding the result of e_1 and e_2 before e_3 in the example at the right because the regions holding a and b should be accessible in e_3.

In letrec constructs there is effect and region polymorphism in both e_1 and e_2. Type polymorphism however is only accepted in e_2.

$$\frac{\begin{array}{c}\pi = \forall\vec{\rho}\vec{\epsilon}.\underline{\tau} \qquad \tau = \mu_1 \xrightarrow{\epsilon.\varphi} \mu_2 \qquad \pi' = \forall\vec{\alpha}.\pi \qquad \mathrm{fv}(\vec{\alpha},\vec{\rho},\vec{\epsilon}) \cap \mathrm{fv}(TE,\varphi_1) = \emptyset \\ TE + \{f : (\pi,\rho)\} \vdash (\lambda^{\epsilon.\varphi} x : \mu_1.e_1)\ \texttt{at}\ \rho : (\tau,\rho),\varphi_1 \qquad TE + \{f : (\pi',\rho)\} \vdash e_2 : \mu,\varphi_2\end{array}}{TE \vdash \texttt{letrec}\ f : (\pi',\rho)\ [\vec{\rho}]\ x\ \texttt{at}\ \rho = e_1\ \texttt{in}\ e_2\ \texttt{end} : \mu, \varphi_1 \cup \varphi_2} \tag{8}$$

The closure for f resides in region ρ. In the ML Kit, more than one function can be bound in the same letrec:

$$\begin{array}{l}\texttt{letrec}\ f_1 : (\pi_1,\rho)\ [\vec{b_1}]\ \ x\ \texttt{at}\ \rho\ =\ e_1, \\ \qquad\qquad\vdots \\ \qquad f_n : (\pi_n,\rho)\ [\vec{b_n}]\ x\ \texttt{at}\ \rho\ =\ e_n, \\ \texttt{in} \\ \quad e \\ \texttt{end}\end{array}$$

A *shared closure* containing all free variables for $f_1, \ldots, f_n$ is stored in region ρ. Because letrec bound functions are known at compile time, no code pointer is needed in the closure. It happens, that the functions do not have any free variables, and we therefore allocate a region to a shared closure of size zero. The function `fun` *fac* n = `if` $n \leq 1$ `then` 1 `else` $n{*}fac(n-1)$ is an example of that.

Example: Consider `letrec` *square* $x = x * x$ `in` e `end`. The function *square* will be translated to a function that is region polymorphic in both argument and result. The premises in rule 8 are satisfied by

$$\begin{array}{l}\pi = \forall\epsilon,\rho_{arg},\rho_{res}.\tau \\ \tau = (\texttt{int},\rho_{arg}) \xrightarrow{\epsilon.\{\mathbf{get}(\rho_{arg}),\mathbf{put}(\rho_{res})\}} (\texttt{int},\rho_{res}) \\ \pi' = \pi \\ \varphi_1 = \{\mathbf{put}(\rho_\lambda)\} \\ \mathrm{fv}(\epsilon,\rho_{arg},\rho_{res}) \cap \mathrm{fv}(\varphi_1) = \emptyset \\ \emptyset + \{square : (\pi,\rho_\lambda)\} \vdash (\lambda^{\epsilon.\varphi} x : (\texttt{int},\rho_{arg}).(\texttt{prim MULprim}\ x\ x\ \texttt{at}\ \rho_{res})\ \texttt{at}\ \rho_\lambda : (\tau,\rho_\lambda),\varphi_1 \\ \emptyset + \{square : (\pi',\rho_\lambda)\} \vdash e : \mu,\varphi_2 \\ \emptyset \vdash \texttt{letrec}\ square : (\pi,\rho_\lambda)\ [\rho_{arg},\rho_{res}]\ x\ \texttt{at}\ \rho_\lambda\ = (\texttt{prim MULprim}\ x\ x\ \texttt{at}\ \rho_{res})\ \texttt{in}\ e\ \texttt{end} : \mu,\varphi_1 \cup \varphi_2\end{array}$$

If *square* is applied in e then φ_2 will contain the latent effect in τ. We will use the simple type `int` in our examples even though `int` is not part of the simple types on page 7.

Application of a letrec bound function is done by first creating a *region vector* holding region actuals and then performing the application. This is different from [7], where a closure holding the free variables and region actuals are created, and then applied to an argument.

$$\frac{\begin{array}{c} TE(f) = (\pi, \rho) \quad \pi \geq \tau \textit{ via } S \quad \tau = \mu_1 \xrightarrow{\epsilon.\varphi} \mu_2 \\ il(S) = (\_, [\rho_1, \ldots, \rho_m], \_) \quad TE \vdash e : \mu_1, \varphi_e \end{array}}{TE \vdash f_{il(S)}\ \texttt{[at}\ \rho_1, \ldots, \texttt{at}\ \rho_m\texttt{]}\ \texttt{at}\ \rho'\ e : \mu_2, \varphi \cup \varphi_e \cup \{\mathbf{get}(\rho, \rho'), \mathbf{put}(\rho'), \epsilon\}} \tag{9}$$

The shared closure for f is accessed in region ρ. A region vector holding region $\rho_1, \ldots, \rho_m$ is stored in region ρ'. The region vector is accessed in the call, so we also have a get effect on ρ'. The actual regions $\rho_1, \ldots, \rho_m$ is found in the instance list $il(S)$.

Example: We will now apply the *square* function from the previous example on the constant 2,

$$square_{il}\ \texttt{[at}\ \rho_{Aarg}, \texttt{at}\ \rho_{Ares}\texttt{]}\ \texttt{at}\ \rho_{reg.vec.}\ \texttt{2}$$

We use the type environment from the previous example.

$$TE = \{square : (\forall \epsilon, \rho_{Farg}, \rho_{Fres}.(\texttt{int}, \rho_{Farg}) \xrightarrow{\epsilon.\{\mathbf{get}(\rho_{Farg}), \mathbf{put}(\rho_{Fres})\}} (\texttt{int}, \rho_{Fres}), \rho_{shared\_clos})\}$$

The substitution S applied on the compound type scheme maps ρ_{Farg} to ρ_{Aarg} and ρ_{Fres} to ρ_{Ares}. The argument `2` elaborates to $(\texttt{int}, \rho_{const})$. Region ρ_{const} and ρ_{Aarg} are unified. The application then elaborates to

$$(\texttt{int}, \rho_{Ares}), \{\mathbf{get}(\rho_{Aarg}), \mathbf{put}(\rho_{Ares})\} \cup \{\mathbf{put}(\rho_{Aarg})\} \cup \{\mathbf{get}(\rho_{shared\_clos}, \rho_{reg.vec.}), \mathbf{put}(\rho_{reg.vec.}), \epsilon\}.$$

Letregions are inserted around expressions using regions which are not used before or after the expression.

$$\frac{TE \vdash e : \mu, \varphi \quad \mathrm{fv}(\varphi') \cap \mathrm{fv}(TE, \mu) = \emptyset}{TE \vdash \texttt{letregion}\ \mathrm{frv}(\varphi')\ \texttt{in}\ e\ \texttt{end} : \mu, \varphi \setminus \varphi'} \tag{10}$$

Example: If we look at the previous example it seems resonable to put

$$\texttt{letregion}\ \rho_{Aarg}, \rho_{reg.vec.}\ \texttt{in}\ \ldots\ \texttt{end}$$

around the application, because ρ_{Aarg} is only used to hold the argument and $\rho_{reg.vec.}$ the region vector. The argument and the region vector are local to the application. We have

$$\mathrm{fv}(TE, (\texttt{int}, \rho_{Ares})) = \{\rho_{shared\_clos}, \rho_{Ares}\}.$$

We set

$$\varphi' = \{\mathbf{get}(\rho_{Aarg}, \rho_{reg.vec.}), \mathbf{put}(\rho_{Aarg}, \rho_{reg.vec.}), \epsilon\}$$

Now we have $\mathrm{fv}(\varphi') \cap \mathrm{fv}(TE, (\texttt{int}, \rho_{Ares})) = \emptyset$, so the letregion rule can be used. The expression

$$\texttt{letregion}\ \rho_{Aarg}, \rho_{reg.vec.}\ \texttt{in}\ square_{il}\ \texttt{[at}\ \rho_{Aarg}, \texttt{at}\ \rho_{Ares}\texttt{]}\ \texttt{at}\ \rho_{reg.vec.}\ \texttt{2 end}$$

elaborates to $(\texttt{int}, \rho_{Ares}), \{\mathbf{put}(\rho_{Ares}), \mathbf{get}(\rho_{shared\_clos})\}$. We have used the type environment, TE, from the previous example.

Unary exception constructors

When declaring unary exceptions an *exception name* is stored in a region ρ. An exception name consists of

- a unique *exception number* generated at runtime because of the generative nature of exceptions,
- a string containing the *name of the exception*, which is printed when the exception is handled at the outer level.

$$\frac{TE + \{excon : ((\tau, \rho_1) \xrightarrow{\epsilon.\emptyset} (\texttt{exn}, \rho_2), \rho)\} \vdash e : (\tau_e, \rho_e), \varphi_e}{TE \vdash \texttt{exception}\ excon\ \tau\ \texttt{at}\ \rho\ \texttt{in}\ e : (\tau_e, \rho_e), \varphi_e \cup \{\mathbf{put}(\rho_2)\}} \tag{11}$$

Because exceptions can escape (be raised and never handled) ρ_1, ρ_2 and all regions free in τ will always be *global regions*. Global regions are allocated at the outer level of the program and never deallocated. The closure that *excon* is bound to is not created because it can only be applied by the `EXCONprim` primitive when generating an exception value. There is no effect on the arrow because the effect of applying the constructor and argument is known by the `EXCONprim` primitive. Declaring an exception constructor allocates two words in ρ_2:

$$\overbrace{\boxed{\textit{exn. number}}\boxed{\textit{name of exn.}}}^{\textit{exception name}}.$$

$$\frac{TE(excon) = ((\tau, \rho_1) \xrightarrow{\epsilon.\emptyset} (\texttt{exn}, \rho_2), \rho) \quad TE \vdash e : (\tau, \rho_1), \varphi_e}{TE \vdash \texttt{prim EXCONprim}\ excon\ e\ \texttt{at}\ \rho_2 : (\texttt{exn}, \rho_2), \varphi_e \cup \{\mathbf{put}(\rho_2)\}} \tag{12}$$

Creating a constructed exception value with the `EXCONprim` primitive stores another two words in region ρ_2:

$$\boxed{\textit{ptr. to exn name}}\boxed{\textit{exn. value}}.$$

The region ρ_1 in rule 11 and 12 is global, because when generating the exception value, by applying `EXCONprim` on the the exception constructor and value, it is not necessary to copy the argument into another global region (for example ρ_2). We save the copying of the argument, but restrict the choice of the argument region (ρ_1) to be global. This can result in the case that other regions (used to compute the argument) are forced to be global too.

Instead of storing the exception name in region ρ_2, in rule 11, we could create a closure, containing the exception name as a free variable, in region ρ. Applying the exception constructor on a value should then store the exception name and value into ρ_2. Then, we would only have put effects on the global region ρ_2 when the exception value is created. However, this has the drawback, that the exception name cannot be reused if the same exception constructor is applied several times to different arguments; at present only two words, $\boxed{\textit{ptr. to exn name}}\boxed{\textit{exn. value}}$, are allocated when creating the exception value.

Nullary exception constructors

Declaring a nullary exception constructor will generate an exception value, so

$$\frac{TE + \{excon : (\texttt{exn}, \rho)\} \vdash e : (\tau_e, \rho_e), \varphi_e}{TE \vdash \texttt{exception}\ excon\ \texttt{at}\ \rho\ \texttt{in}\ e : (\tau_e, \rho_e), \varphi_e \cup \{\mathbf{put}(\rho)\}} \tag{13}$$

when creating an exception value with the `EXCONprim` primitive we only have to find the address of the value.

$$\frac{TE(excon) = (\texttt{exn}, \rho)}{TE \vdash \texttt{prim EXCONprim}\ excon\ \texttt{at}\ \rho : (\texttt{exn}, \rho), \emptyset} \tag{14}$$

Declaring a nullary exception constructor (rule 13) allocates the following three words in the global region ρ:

$$\overbrace{\boxed{\textit{ptr. to exn. name}}}^{\text{extra indirection}} \text{ and } \overbrace{\boxed{\textit{exn. number}}\boxed{\textit{name of exn.}}}^{\textit{exception name}}.$$

The extra indirection introduced above is used to get uniform representation of both unary and nullary exception values.

Handlers

The handler expression e_1 **`handle`** e_2 puts an exception handler (e_2) on the stack (i.e. the pointer to the closure for e_2 held in a region, say $\rho_{handleClos}$), and stores a pointer to the previous exception handler (see figure 3). The expression e_2 will always be a lambda abstraction.

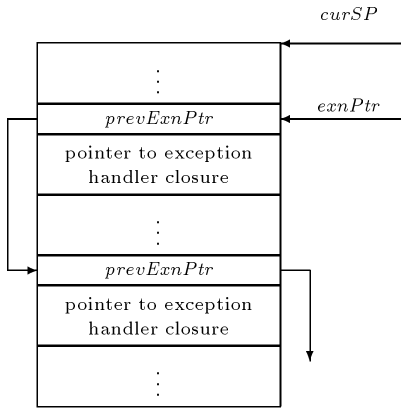


Figure 3: *The top exception handler is kept in a global variable exnPtr.*

If an exception is raised by `raise` e in expression e_1 of the handler expression, the stack is adjusted down to *exnPtr*. Regions allocated after the *exnPtr* was pushed onto the stack are also deallocated. The handler function is now applied to the exception packet raised by accessing the region, $\rho_{handleClos}$. Region inference has to ensure that the regions deallocated are not needed for the rest of the computation. This is ensured by the stack nature of regions. All regions allocated in e_1 (i.e. allocated after the handler closure) are only used by e_1. Exception handling in the ML Kit is also explained in [10].

For the e_1 `handle` e_2 construct we have to unite the effects for both e_1 and e_2 including the latent effect of the handler function (i.e. we assume that an exception value can be raised in e_1).

$$\frac{TE \vdash e_1 : \mu_2, \varphi_1 \qquad TE \vdash e_2 : ((\mathtt{exn}, \rho_1) \xrightarrow{\epsilon.\varphi} \mu_2, \rho_2), \varphi_2}{TE \vdash e_1 \ \mathtt{handle}\ e_2 : \mu_2, \varphi \cup \varphi_1 \cup \varphi_2 \cup \{\epsilon, \mathbf{get}(\rho_2)\}} \tag{15}$$

The effect of raising an exception is only the effect of evaluating the exception packet. The effect of applying the handler function has already been included in the handle construct. The expression `raise` e gets a fresh type and place so that, for example, it is possible to elaborate `if` *fail* `then raise` *exn_fail* `else` e, where e can elaborate to an arbitrary type and place.

$$\frac{TE \vdash e : (\mathtt{exn}, \rho), \varphi_e}{TE \vdash (\mathtt{raise}\ e) : \mu : \mu, \varphi_e} \tag{16}$$

As we will see (section 6), declaring exceptions in loops can be rather disastrous with respect to memory use.

References

A reference to value v is evaluated by applying the primitive `REFprim` to v. In memory we have

tag `REFprim`	*value*

,

where the first word is used by the polymorphic equality function to identify the reference at runtime.

$$\frac{TE \vdash e : (\tau_e, \rho_e), \varphi_e}{TE \vdash \mathtt{prim\ REFprim}\ e\ \mathtt{at}\ \rho : ((\tau_e, \rho_e)\ \mathtt{ref}, \rho), \{\mathbf{put}(\rho)\} \cup \varphi_e} \tag{17}$$

Dereferencing a value is done by accessing the region holding `REFprim`:

$$\frac{TE \vdash e : ((\tau, \rho)\ \mathtt{ref}, \rho_{ref}), \varphi_e}{TE \vdash \mathtt{prim\ DEREFprim}\ e\ \mathtt{at}\ \rho : (\tau, \rho), \{\mathbf{get}(\rho_{ref})\} \cup \varphi_e} \tag{18}$$

The type of a reference is not region polymorphic, so when assigning a new value to a reference the value must be in the same region as the previous value:

$$\frac{TE \vdash e_1 : ((\tau_1, \rho_1)\ \mathtt{ref}, \rho_{ref}), \varphi_1 \qquad TE \vdash e_2 : (\tau_1, \rho_1), \varphi_2}{TE \vdash \mathtt{prim\ ASSIGNprim}\ e_1\ e_2\ \mathtt{at}\ \rho : (\mathtt{unit}, \rho), \varphi_1 \cup \varphi_2 \cup \{\mathbf{get}(\rho_{ref}), \mathbf{put}(\rho)\}} \tag{19}$$

We only have a get effect on ρ_{ref} because no new memory cell is allocated. The value () (of type unit) is put in region ρ.

Equality

Polymorphic equality is implemented by tagging all values. A function, implemented in the runtime system, traverses the two values to compare. A get effect is therefore assumed om all region variables in the two types.

$$\frac{TE \vdash e_1 : (\tau_1, \rho_1), \varphi_1 \qquad TE \vdash e_2 : (\tau_2, \rho_2), \varphi_2 \qquad \vec{\rho} = \mathrm{frv}(\tau_1) \cup \mathrm{frv}(\tau_2)}{TE \vdash \mathtt{prim\ EQUALprim}\ e_1\ e_2\ \mathtt{at}\ \rho : (\mathtt{bool}, \rho), \varphi_1 \cup \varphi_2 \cup \{\mathbf{get}(\vec{\rho}), \mathbf{get}(\rho_1, \rho_2), \mathbf{put}(\rho)\}} \tag{20}$$

3.4 Multiplicity and physical size inference

Multiplicity Inference infers logical sizes of regions. That is the number of values put into each region. The Physical Size Inference phase [7] will translate multiplicities into the physical size of the objects allocated.

A type system and an algorithm for inferring multiplicities are given in [25]. Only multiplicities of 0, 1 and ∞ are used in the ML Kit, because regions with finite multiplicity greater than 1 occur seldomly ([25]).

The physical size of each letregion bound and formal region variable is written on the region annotated lambda program. The size of a formal region variable ρ_f is an upper bound of the size of objects allocated into ρ_f when the function is applied.

The function f in `letrec` $f\ [\rho : 1]\ x\ =\ e_1$ `in` e_2 `end` will store a value with size one word into region ρ each time it is applied. At runtime, ρ can be *aliased* with say, ρ', where ρ' has multiplicity n or ∞.

We say that two region variables are aliased if they can denote the same region at runtime. If for example actual region variable ρ_{actual} is given as argument to formal region variable ρ_{formal} then ρ_{formal} and ρ_{actual} are aliased. Two formal region variables can also be aliased; they can be aliased with the same actual region variable.

3.5 Storage mode analysis

The storage mode analysis takes all allocation directives (`at` ρ) and turns them into one of the following, where

- `attop` ρ, means that the value is always stored at top in region ρ. This reflects the original meaning of the `at` ρ annotation. Region inference is based on the invariant, that all values are stored at top. The storage mode analysis finds allocation points where it is safe to weaken that invariant.
- `atbot` ρ, means that region ρ will be reset before allocating the value. Resetting is done by freeing all memory allocated for the region. The region is still allocated on the region stack after resetting. It is only necessary to reset infinite regions; finite regions only holds one object and their size is fixed. If ρ' is letregion bound it will always appear as `attop` ρ' or `atbot` ρ'. No formal region variables can have storage mode `atbot`, so the multiplicity (finite of infinite) is known at compile time. No test is therefore needed at runtime to decide whether to reset ρ or not.
- `sat` ρ, means that the actual storage mode (`attop` or `atbot`) is determined at runtime. If ρ is a formal region parameter of a letrec bound function it will always appear as `attop` ρ or `sat` ρ. The region identifier holds the multiplicity (finite or infinite) and the actual storage mode (`attop` or `atbot`) at runtime. Computing the allocation v `sat` ρ is as follows: first the multiplicity of ρ is checked, and if finite then v is stored into ρ. If infinite then the storage mode for ρ is checked, and if `atbot` then ρ is reset before storing v. If `attop` then v is stored into ρ.

When applying a letrec bound function to actual regions, the storage mode (`attop` or `atbot`) and multiplicity will be passed together with the region. The internal representation of regions makes it possible to both pass the region, multiplicity, and storage mode in one word.

The storage mode analysis is essential for handling tail recursion. Tail recursion in conventional compilers is normally handled by using loops instead of function calls, and thereby only creating one activation record [1]. In the ML Kit, the storage mode analysis tries to reuse the region holding the argument in the tail call. Instead of having a storage consumption that is linear in the call depth in the region holding the argument, the region should only contain one argument at any time.

As explained in [7], the storage mode analysis is based on statically inferred liveness properties. If we look at the function `longlist` on page 6, we see that the pair on line 11 is allocated `attop` in region `r6`. This is because variable `v26` is live at the allocation point. Region `r5` and `r6` are free in the type of `v26`, hence region `r6` contains live data. The region vector in line 7 is allocated `atbot` because it is the only value that region `r8` will ever contain.

The storage mode analysis is not always good at inserting `atbot` annotations, and some programs need extra programmer attention. Experience has shown that the profiler is very good at finding allocation points where we assume an `atbot` allocation but actually get an `attop` allocation. In section 6 we present some hints on making functions tail recursive. More details can be found in [23, 20] and [4].

3.6 Region flow graph

The *region flow graph* has three purposes:

- show how regions are aliased.
- show the storage modes passed in region polymorphic function applications.
- show inferred multiplicities for the letregion or letrec bound region variables. The multiplicities are not essential for the graph, and are just annotated as extra information.

In region polymorphic function applications, formal region variables are aliased with letregion bound region variables. At runtime, every allocation will be in regions, that were originally bound to letregion bound region variables. However, it is not always easy to see which letregion bound region variables will be used in allocations inside region polymorphic functions. Given an allocation into a formal region variable, say ρ_3, and the knowledge that the allocation at runtime is into a region originally bound to the letregion bound region variable ρ_1, the graph will show the possible call sequences which will alias ρ_3 with ρ_1. For instance, the program

```
letregion ρ1 : ∞, ρ2 : 3
in
  letrec f [ρ3 : ∞, ρ4 : 3] x = (x + (1.0 sat ρ3) attop ρ3, 2.0 attop ρ3) sat ρ4
  in
    letrec g [ρ5 : ∞, ρ6 : 3] x = f [sat ρ5, sat ρ6] x
    in
      g [atbot ρ1, atbot ρ2] 1.0
    end
  end
end
```

will generate the region flow graph

$$f\ \texttt{[}\rho_3:\infty\texttt{]} \xrightarrow{\texttt{sat}} g\ \texttt{[}\rho_5:\infty\texttt{]} \xrightarrow{\texttt{atbot}} \texttt{letregion [}\rho_1:\infty\texttt{]}$$
$$f\ \texttt{[}\rho_4:3\texttt{]} \xrightarrow{\texttt{sat}} g\ \texttt{[}\rho_6:3\texttt{]} \xrightarrow{\texttt{atbot}} \texttt{letregion [}\rho_2:3\texttt{]}.$$

Note that the region flow graph is a "reverse calling" (or "called from") sequence where for instance the first arrow means that f is called from g. The function f is called in scope of g with ρ_5 and ρ_6 as actual region parameters being aliased with ρ_3 and ρ_4 respectively. Note that a pair occupies three words in memory.

The storage mode (passed with the region) is written on the arrow. Two region variables can be aliased in several call sites giving only one edge in the graph. They can be aliased with different storage modes, so we can choose to write all the possible storage modes on the arrow or just a calculated maximum (or minimum). Usually it is interesting to see if two region variables are aliased with storage mode `attop` because this can be the reason for a space leak. We therefore choose to write the maximum calculated in the following way.

first storage mode	**second storage mode**	**result storage mode**
`atbot`	`atbot`	`atbot`
`atbot`	`attop`	`attop`
`attop`	`atbot`	`attop`
`sat`	`sat`	`sat`
`sat`	`attop`	`attop`
`attop`	`sat`	`sat`
`attop`	`attop`	`attop`
-	-	Fail

The combinations (`sat`, `atbot`) and (`atbot`,`sat`) will never happen because given the edge $\rho_1 \longrightarrow \rho_2$ then ρ_2 will either be formal or letregion bound. If ρ_2 is formal then only `sat` or `attop` can be passed. If ρ_2 is letregion bound, then only `atbot` or `attop` can be passed.

3.7 Auxiliary regions

The memory used to store data structures (defined with constructors in datatype declarations) are distributed over a number of regions. In this section we show how some of these regions are checked for resetting at program points where datatype constructors are allocated.

A polymorphic list type can be defined thus:

$$\texttt{datatype}\ \alpha\ list\ \texttt{=}\ NIL$$
$$\texttt{|}\ CONS\ \texttt{of}\ \alpha * \alpha\ list.$$

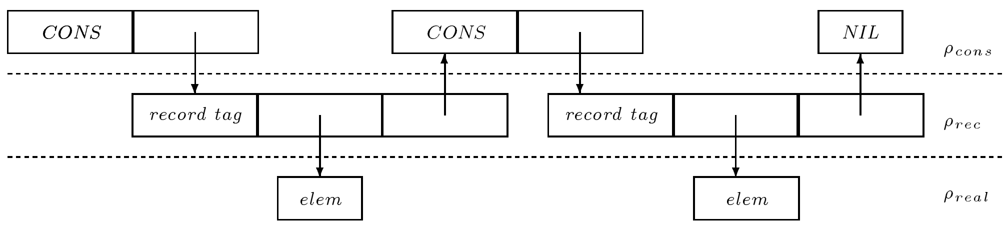
The two constructors get type schemes

$$NIL\ :\ \forall\alpha\rho_\alpha\rho_{rec}.(\alpha,\rho_\alpha)\ \ list_{[\rho_{rec}]}$$
$$CONS\ :\forall\alpha\rho_{cons}\rho_{rec}\rho_\alpha.((\alpha,\rho_\alpha)*((\alpha,\rho_\alpha)\ \ list_{[\rho_{rec}]},\rho_{cons}),\rho_{rec}) \xrightarrow{\texttt{put}(\rho_{cons})} ((\alpha,\rho_\alpha)\ \ list_{[\rho_{rec}]},\rho_{cons}).$$

We have a put effect in the type for *CONS* because a *CONS* cell is allocated in ρ_{cons} when the constructor is applied.

The two types contain three regions: one for the constructors (ρ_{cons}), one for the pairs (ρ_{rec}) and one for the elements (ρ_α).

A list of reals (α instantiated to `real`) will, at runtime, use three regions. The following figure shows how the list is represented in memory. The real type elements in the list are called *elem*.



If α is instantiated to `int` which is unboxed then the list will be in regions ρ_{cons} and ρ_{rec} only.

We have annotated a list of regions on the type name *list* in the above type for *NIL* and *CONS*. The regions in the list are called *auxiliary regions*. The auxiliary regions are the regions free in the type of the constructors minus the region containing the constructors and the regions paired with bound type variables in the data type declaration (i.e. minus ρ_{cons} and ρ_α in the above example).

If we write the above type shemes for *NIL* and *CONS* as $\forall\alpha\rho_\alpha\rho_{rec}.\tau_{nil}$ and $\forall\alpha\rho_\alpha\rho_{rec}\rho_{cons}.\tau_{cons}$, respectively, we have

$$aux\_regs = \mathrm{frv}(\tau_{nil},\tau_{cons}) \setminus (\{\rho_{cons}\} \cup \mathrm{frv}(\alpha,\rho_\alpha)) = \{\rho_{rec}\}.$$

Auxiliary regions are associated with an entire data type declaration. For instance, if we have a data type declaration

$$
\begin{aligned}
\texttt{datatype}\ (\alpha_1, \ldots, \alpha_n)\ \mathit{list}\ \texttt{=}\ & CONS_1 \\
& \vdots \\
\texttt{|}\ & CONS_l \\
\texttt{|}\ & CONS_{l+1}\ \texttt{of}\ \ldots \\
& \vdots \\
\texttt{|}\ & CONS_m\ \texttt{of}\ \ldots,
\end{aligned}
$$

then the auxiliary regions are

$$aux\_regs = \mathrm{frv}(\tau_{CONS_1}, \ldots, \tau_{CONS_m}) \setminus (\{\rho_{cons}\} \cup \mathrm{frv}(\alpha_1, \rho_{\alpha_1}) \cup \mathrm{frv}(\alpha_n, \rho_{\alpha_n})),$$

where $\tau_{CONS_1}, \ldots, \tau_{CONS_m}$ are the types in the type schemes for the constructors $CONS_1, \ldots, CONS_m$. The region ρ_{cons} is the region where the constructors $CONS_1, \ldots, CONS_m$ are allocated.

Usually ρ_{cons} and the auxiliary regions will be local to the list (i.e. only contain objects in connection with the list).

The code which creates a list is organized such that the first constructor allocated for the list is *NIL*; the argument to a constructor is fully evaluated before the constructor is applied and thereby allocated. The first allocated constructor can therefore not be recursive in the list type. The *NIL* constructor will often be the first allocated object in ρ_{cons} and hence the allocation will often get storage mode `atbot` or `sat`.

The auxiliary regions annotated on the *NIL* constructor will also, if possible, get storage mode `atbot` or `sat` at the program points where *NIL* is allocated. It is often the case that they get the same storage mode as ρ_{cons}; if ρ_{cons} is reset then no constructors, in ρ_{cons}, will be alive and the arguments given to the constructors are only alive if they have been de-constructed.

If we look at the region annotated lambda program for `longlist` on page 6 we see on line two that *NIL* is allocated `sat` in region `r5`. If ρ_{cons} has been reset then there will be no *CONS* constructors alive, and the auxiliary regions holding the arguments to *CONS* can often be reset too. On line two we see that the auxiliary region also gets storage mode `sat`. The function `longlist` generates a list of integers, so only two regions are used to store the list.

In section 6.11 we show an example on how to reuse memory allocated for data structures, and here auxiliary regions are essential.

Polymorphic datatypes

In the list generated by `longlist` there is only one auxiliary region, but in general there can be more. For instance, a special list can be defined as follows

$$
\begin{aligned}
\texttt{datatype}\ \alpha\ \mathit{list'}\ \texttt{=}\ & NIL \\
\texttt{|}\ & CONS\ \texttt{of}\ \alpha\ \mathit{list'} * ((\alpha * \alpha) * (\alpha * \alpha)).
\end{aligned}
$$

The two constructors gets type scheme

$$
\begin{aligned}
NIL\ &: \forall \alpha \rho_\alpha \rho_1 \rho_2 \rho_3 \rho_4 . (\alpha, \rho_\alpha)\ \mathit{list'}_{[\rho_1,\rho_2,\rho_3,\rho_4]} \\
CONS\ &: \forall \alpha \rho_{cons} \rho_\alpha \rho_1 \rho_2 \rho_3 \rho_4 . ((\alpha, \rho_\alpha)\ \mathit{list'}_{[\rho_1,\rho_2,\rho_3,\rho_4]}, \rho_{cons}) * \\
&\qquad ((((\alpha, \rho_\alpha) * (\alpha, \rho_\alpha), \rho_1) * ((\alpha, \rho_\alpha) * (\alpha, \rho_\alpha), \rho_2), \rho_3), \rho_4) \\
&\qquad \xrightarrow{\mathtt{put}(\rho_{cons})} ((\alpha, \rho_\alpha)\ \mathit{list'}_{[\rho_1,\rho_2,\rho_3,\rho_4]}, \rho_{cons}).
\end{aligned}
$$

The argument passed to *CONS* will be in four pairs which are put into four auxiliary regions $(\rho_1, \ldots, \rho_4)$. The auxiliary regions for *list'* are then

$$aux\_regs = \mathrm{frv}(\tau_{nil}, \tau_{cons}) \setminus \{\rho_{cons}, \rho_\alpha\} = \{\rho_1, \ldots, \rho_4\},$$

where τ_{nil} and τ_{cons} are the types for *NIL* and *CONS* in the above type schemes.

If we look at the region annotated lambda program for a function *copy_list'*

```
fun copy_list' NIL = NIL
  | copy_list' (CONS(t, e)) = CONS(copy_list' t, e)
```

we see that the four auxiliary regions $(\rho_1, \ldots, \rho_4)$ have storage mode `sat` at the program point where the *NIL* constructor is allocated (`r52, r55, r56` and `r57`).

```
fun copy_list'(r51 : inf, r52 : inf, r55 : 0, r56 : 0, r57 : 0, var5)(* at r2  pp50  *) =
    (case  var5 of
            NIL$23 => NIL$23 sat r51  pp51   with sat r57 sat r56 sat r55 sat r52
         |  CONS$23 =>
   ...
```

If we instead declare *list'* as

```
datatype list_rp = NIL
                 | CONS of list_rp * (((real * real) * (real * real)) *
                                      ((real * real) * (real * real))),
```

then the regions containing the reals will also get storage mode `sat` at the program point where *NIL* is allocated:

```
fun copy_list_rp(r347 : inf, r348 : inf, r351 : 0, r352 : 0, r353 : 0, r354 : 0, r355 : 0,
                r356 : 0, r357 : 0, r358 : 0, r359 : 0, r360 : 0, r361 : 0, r362 : 0,
                r363 : 0, r364 : 0, r365 : 0, var37)(* at r286  pp73  *) =
     (case  var37 of
             NIL$35 => NIL$35 sat r347  pp74   with sat r365 sat r364 sat r363 sat r362 sat r361
                                                    sat r360 sat r359 sat r358 sat r357 sat r356
                                                    sat r355 sat r354 sat r353 sat r352 sat r351
                                                    sat r348
           | CONS$35 =>
                   ...
```

We have sixteen auxiliary regions; eight regions for the reals, and eight regions for the pairs.

It is possible for the programmer to instantiate the type of *copy_list'* such that the function will not be polymorphic in the list type. Then it should be possible to also give the auxiliary regions containing the elements storage mode `sat` at the program point where *NIL* is allocated. For instance, with

```
fun copy_list'' NIL : (real * real) list' = NIL
  | copy_list'' (CONS(t, e)) = CONS(copy_list'' t, e)
```

it should be possible to give the regions containing the reals storage mode `sat` (as in function `copy_list_rp`). This is not, at present, possible in The ML Kit. Note that this would increase the region vector passed to *copy_list''* compared to the region vector passed to *copy_list'*.

Only nullary constructors resets auxiliary regions

In The ML Kit only nullary constructors can reset auxiliary regions. It should be possible, however, to also reset some of the auxiliary regions when allocating non recursive unary constructors. If we, for instance, change *list'* to

```
datatype α list'' = NIL
                  | VAL of real
                  | CONS of α list'' * ((α * α) * (α * α)).
```

and *copy_list'* to

```
fun copy_list'' NIL= NIL
  | copy_list'' (CONS(t, e)) = CONS(copy_list'' t, e)
  | copy_list'' (VAL (i)) = VAL i
```

we see that none of the five auxiliary regions are reset when the *VAL* constructor is allocated.

```
fun copy_list''(r87 : inf, r88 : inf, r91 : 0, r92 : 0, r93 : 0, r94 : 0, var37)(* at r50  pp54  *) =
     (case  var37 of
             NIL$19 => NIL$19 sat r87  pp55   with sat r94 sat r93 sat r92 sat r91 sat r88
           | CONS$19 =>
                ...
           | VAL$19 => VAL$19(prim(decon(VAL$19), [var37]) pp63 )sat r87  pp62
     )
```

It is obviously unsafe to reset the region containing the real value when allocating the *VAL* constructor. However the other four auxiliary regions used by the *CONS* constructor could get storage mode `sat` at the program point where the *VAL* constructor is allocated.

Introducing auxiliary regions on unary constructors will, in some cases, suffer under the overhead of checking and resetting many auxiliary regions. For instance, on balanced binary trees with unary constructors as leafs then when allocating approximately half of the leafs we first have to check and maybe reset auxiliary regions.

The list construct is in that sence very efficient because only one nullary constructor has to be checked and reset for an entire list. However, at present, balanced binary trees with nullary constructors as leafs introduce the same overhead as balanced trees with unary constructors as leafs would do.

3.8 Paths in the region flow graph

The profiler can find the possible paths between two region variables in the region flow graph (see section 5.2). In this section we outline the algorithm used to find the paths.

The paths from ρ_i to ρ_j in the region flow graph is found by first computing the *strongly connected components* ([3, page 191]) and then finding the paths between the two components holding ρ_i and ρ_j respectively.

A *directed subgraph* ($G = (V, E)$, where V is a set of vertices and E is a set of edges) is *strongly connected* iff, for each pair of vertices (v_i, v_j), there is a path from v_i to v_j and from v_j to v_i. The edges must be followed in the direction of the arrow only. A strongly connected component (scc) of a digraph is defined as a maximal *strongly connected subgraph.*

We define the equivalence relation S on the vertices thus: for any $v_i, v_j \in V$, $v_i S v_j$ iff there is a path from v_i to v_j and from v_j to v_i in G. The strongly connected components of G are then the equivalence classes of S. A scc is written $C = (V_C, E_C)$, where V_C contains the vertices from the equivalence class defining C and E_C contains all edges (v_i, v_j) from G, where v_i and v_j are in V_C.

We say that there is always a path from v_i to v_i, so the smallest connected components consists of one vertices. If v_i and v_j is in the same scc, then it is obvious that all vertices in the path from v_i to v_j and from v_j to v_i are also in the scc.

Given the strongly connected components $C_1, \ldots, C_n$ of a graph G we make the *condensation* of G which is a new digraph $G_{con} = (V_{con}, E_{con})$. The vertices set V_{con} consists of the n vertices C_i ($V_{con} = \{C_1, \ldots, C_n\}$). The edge set E_{con} contains an edge (C_i, C_j) iff there exists $v \in V_{C_i}$ and $w \in V_{C_j}$ and an edge from v to w in G (i.e. $(v, w) \in E$).

Given two vertices v and w in the region flow graph (i.e. in G), we can now use the condensation G_{con} to find the possible paths from v to w in G. Let C_v and C_w be the scc which contains v and w respectively.

The condensation G_{con} will always be acyclic, and a depth first traversal of G_{con} can find the possbile paths from C_v to C_w. We write the n possible paths between the two scc thus:

$$[(C_v, C_{1_0}, \ldots, C_{1_l}, C_w), \ldots, (C_v, C_{n_0}, \ldots, C_{n_m}, C_w)].$$

Each path $(C_v, C_{i_0}, \ldots, C_{i_l}, C_w)$ is now translated into a subgraph $G_{(v,w)_i} = (V_{(v,w)_i}, E_{(v,w)_i})$ of G where $V_{(v,w)_i} = V_{C_v} \cup V_{C_{i_0}} \cup \ldots \cup V_{C_{i_l}} \cup V_{C_w}$ and $E_{(v,w)_i} = \{(v,w) | v \in V_{(v,w)_i} \wedge w \in V_{(v,w)_i} \wedge (v,w) \in E\}$. Each element in the list

$$[G_{(v,w)_1}, \ldots, G_{(v,w)_n}],$$

is now a subgraph of G describing possible paths between v and w (there can be cycles is each subgraph).

4 Using The Region Profiler

Using the profiler is mainly split into three phases. First, the ML source program must be compiled with the profiling option turned on. Second, the assembler files must be assembled and linked using the profiling version of the runtime system. It is then possible to pass command line options controlling the profiling strategy when executing the target program. Finally, the exported profiling data (called the *profile data file*) must be processed by the *graph generator* giving the final *profile graphs*.

Both the target program and graph generator take several command line options, but mostly, only few options are necessary. In this section we show the steps from compiling to generating the profile graphs on an example program. The ML Kit is modified almost every day, so the snapshots shown can have changed in your version of the ML Kit.

4.1 Compiling the ML source program

In the ML Kit region profiling is controlled by the `interact` function [24] which you activate by typing `interact();` at the ML Kit command line. At the top level menu, choose menu item `Profiling`.

```
- interact();

        0       General.............  >>>
        1       File................  >>>
        2       Front End...........  >>>
        3       Lambda..............  >>>
        4       Region Inference.....  >>>
        5       Storage Mode Analysis  >>>
        6       Backends............  >>>
        7       Profiling...........  >>>

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

>7
```

Then choose region profiling by setting **extended region profiling**[8] on. The region annotated lambda program is written on a log file if **generate lambda code with program points** is on. You can set the log file in the `File` menu at the top level.

```
        Profiling

        0       region profiling............................ off
        1       extended region profiling................... on
        2       generate lambda code file with program points on
        3       generate VCG graph.......................... off
        4       Paths between two nodes..................... [] >>>
        5       Program points..............................  >>>

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

>q
```

You can now compile the example program (`let`) found in figure 4, by quitting the `interact` function, and typing

```
- evalFile "let";
```

Two files containing the target program `hpcode1.s` and `link.s` are generated. The ML Kit supports *incremental compilation* [8]. The assembler files called `hpcode#.s` are numbered from 1 to n, where n is the number of times the function `evalFile` has been executed without resetting the ML Kit with `reset()`. After a reset the assembler files will be numbered from one again. The link file (`link.s`) is changed each time a new assembler file (`hpcode#.s`) is generated. A target program consisting of several assembler files will execute each assembler file in order of creation.

4.2 Assembling and linking

Given the two files `hpcode1.s` and `link.s` you can generate the target program (`a.out`) by typing

```
cc -Aa hpcode1.s link.s runtimeHPPAProf.o -lm
```

where `runtimeHPPAProf.o` is the runtime system compiled with region profiling enabled[9]. Executing `a.out` now gives:

[8] Region profiling has been implemented several times in the ML Kit and the version described here is called extended region profiling.

[9] See the makefile in the runtime directory for more information on compiling the runtime system.

```
let
  fun map f nil = nil
    | map f (x :: L) = f x :: map f L
  fun longlist 0 = []
    | longlist n = n :: longlist (n-1)
  val list = longlist 100000
  val g = fn x : int => x + 1
  fun loop 0  = [42]
    | loop n = (let val foo = map g list in loop (n-1) end )
in
  loop 10
end;
```

Figure 4: *Example program* `let` *which maps function* `g` *on a large list 10 times.*

```
> a.out

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

The target program is controlled by switches explained in section 5.5. From above we see that (by default) we are profiling with a *virtual time* alarm ticking once every second. The profile data file is written on file `profile.rp`.

After the target program has executed, region statistics are printed on stdout. The region statistics are collected independently of the target profiling strategy above and are exact values for the program.

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

SBRK.
  Number of calls to sbrk                    :       1651
  Number of bytes allocated in each SBRK call :      24240
  Total number of bytes allocated by SBRK     :   40020240 (38.2Mb)

REGIONPAGES.
  Size of one page              :        800 bytes

  Max. no. of simultaneously allocated pages :      49514
  Number of allocated pages now              :          3

REGIONS.
  Size of infinite region descriptor :         28 bytes
  Size of finite region descriptor   :          8 bytes

  Number of calls to allocateRegionInf   :         25
  Number of calls to deallocateRegionInf :         22

  Number of calls to allocateRegionFin   :    1100023
  Number of calls to deallocateRegionFin :    1100023

  Number of calls to alloc                 :    2200014
  Number of calls to resetRegion           :         22
  Number of calls to deallocateRegionsUntil :          0

  Max. no. of co-existing regions (finite plus infinite) :     100037
  Number of regions now                                  :          3

  Live data in infinite regions :         24 bytes ( 0.0Mb)
  Live data in finite regions   :          0 bytes ( 0.0Mb)
  --------------------------------------------------------
  Total live data               :         24 bytes ( 0.0Mb)

  Maximum space used for region pages              :   39611200 bytes (37.8Mb)
  Maximum space used on data in region pages       :   22000068 bytes (21.0Mb)
      Space in regions at that time used on profiling :   17600112 bytes (16.8Mb)
  ---------------------------------------------------------------------------
  Maximum allocated space in region pages          :   39600180 bytes (37.8Mb)

  Memory utilisation for infinite regions (  39600180/  39611200) : 100%
```

```
  Maximum space used on the stack for finite regions             :     800092 bytes ( 0.8Mb)
      Additional space used on profiling information at that time :     800096 bytes ( 0.8Mb)
  -----------------------------------------------------------------------------------------------
  Maximum space used on finite regions on the stack              :    1600188 bytes ( 1.5Mb)

  Max. size of stack when program was executed        :    3601056 bytes ( 3.4Mb)
  Max. size of stack in a profile tick                :    3592632 bytes ( 3.4Mb)

*********End of region statistics*********
>
```

Memory for regions are managed by the runtime system [10, page 17] and in the `SBRK` part above we see how memory is allocated from the operating system.

Each region consists of several *region pages* whose size is found in the `REGIONPAGES` part (figure 2 on page 5 shows how infinite regions are build out of region pages). The value

```
Max. no. of simultaneously allocated pages :      49514
```

multiplied by

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

gives the maximal memory use in infinite regions (39611200 bytes).

In the `REGIONS` part, we see the number of calls to finite and infinite region operations, respectively. The target program has allocated 25 infinite regions and deallocated 22; hence three global regions were alive when the program finished. This shows that global regions are not necessarily deallocated, but if global regions are unified, then there will be only one live region when the target program finishes.[10]

No finite regions are alive. We have allocated 2200014 objects in infinite regions. It has been possible to reset an infinite region 22 times. The `deallocateRegionsUntil` operation is only used when raising exceptions.

Because objects allocated in infinite regions are not split across different region pages it is not always possible to fill out all region pages. The value

```
Memory utilisation for infinite regions (  39600180/  39611200) : 100%
```

shows memory utilisation at the moment where the program had allocated the largest amount of memory. The size of objects in finite regions allocated on the stack is shown together with the overhead produced by the profiler. The values

```
Max. size of stack when program was executed        :    3601056 bytes ( 3.4Mb)
```

and

```
Max. size of stack in a profile tick                :    3592632 bytes ( 3.4Mb)
```

can be used to see if it is necessary to profile more detailed. If the difference between the two figures is large you can profile with a smaller *time slot*. When the target program stops normal program execution and runs through memory, collecting profile data, we call it a *profile tick*. The time slot decides how often a profile tick occurs.

After execution of the target program we have a profile data file named `profile.rp`.

4.3 Processing the profile data file

From the profile data file `profile.rp` you can produce profile graphs with the graph generator `rp2ps`[11]. The graph generator is controlled by switches explained in section 5.6. You generate a *region graph* by executing

```
> rp2ps -region
```

The region graph is put in file `region.ps` shown in figure 5.

Regions shown are sorted by size (area) with the largest at top and the smallest at bottom. If there are more regions than can be shown in different shades, the smallest are collected in an *other band* at the bottom. The test program allocates about 22Mb, and it is mainly two regions (`r20inf` and `r21inf`) that are responsible for the large memory use. Each region is identified with a number that matches a `letregion` bound region variable in the region annotated lambda program (figure 10). Infinite regions end with "`inf`" and finite regions with "`fin`".

The *max. allocation line* "Maximum allocated bytes in regions: ..." at top of figure 5 shows the maximum number of bytes allocated in regions when the target program was executed. Because we also show the stack use on the graph (as the `stack` band), we offset the max. allocation line upwards by the maximum stack use shown. The space between the max. allocation line and the top band shows the inaccuracy of the profiling

[10] A flag `unify global regions` can be toggled in the `Region Inference` menu when using `interact`. If the program uses real values there will be two live regions, because *real*-type objects need special alignment at runtime.

[11] To compile `rp2ps` consult the makefile in the directory where `rp2ps` is located.

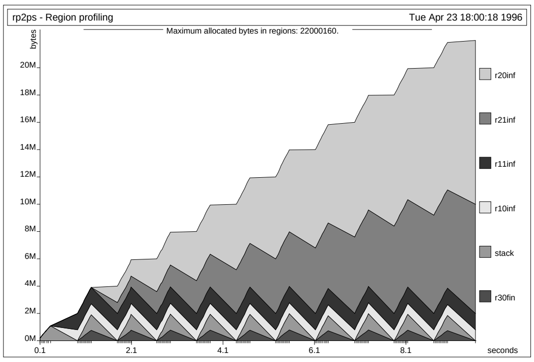


Figure 5: *The region graph shown is generated by* `rp2ps -region`*, and shows memory use for all regions and the machine stack. The graph shows six bands, where band* ***r20inf*** *is the band having the largest area.*

strategy used. Having a large gap indicates that a smaller time slot should be used (section 5.5) or maybe another compile profiling strategy (see for example section 5.7).

In the region annotated lambda program (figure 10) we find a region vector containing `r20` and `r21` in a call to `map` (line 63), and hence, the two regions contain the result list from `map`.

To see the allocation points responsible for allocations in region `r20`, you can produce an *object graph* for that region by typing:

```
> rp2ps -object 20
```

The object graph generated is shown in figure 6.

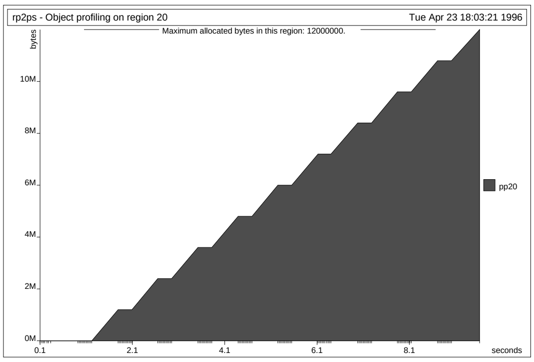


Figure 6: *The object graph shown is generated by* `rp2ps -object 20` *and shows the only allocation point (*`pp20`*) allocating memory in region* `r20`*.*

By consulting the region annotated lambda program (figure 10) we can locate program point `pp20` and see that all allocations in `r20` result from allocating pairs (line 55).

In figure 7 we see that region 10 (11 is similar) has constant size, and after consulting the region annotated lambda program (line 28), we conclude that the two regions hold the value `list` which is live to the end of execution.

If we look at region 30 (figure 8) we see the traditional *stack*-based implementation of recursive function calls. For each call to `map` we pass two regions (`pp18`, line 52). The region vector to hold them is first deallocated after the return from `map`, and we therefore get 10 peaks on the graph.

The recursive call of map is also shown on the stack, because here we have to store all live variables before each call (figure 9).

Each graph has some small *tabs* on the x-axis representing profile ticks. We see they are grouped which is not a normal behaviour and are due to details of the implementation. In the implementation we decided only to allow profile ticks at entrance to functions. Therefore there is no profile ticks on return from the `map` calls (this behaviour is especially clear on figure 8 and 9). By the region annotated lambda program (line 47–59) we conclude that it takes about $\frac{1}{2}$ second to make 100.000 returns each involving

- one deallocation of a finite region, line 53
- allocation of one pair, line 55

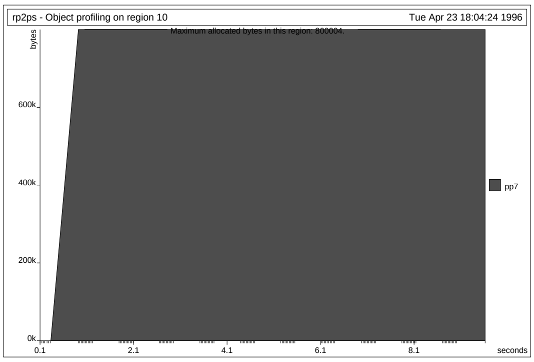


Figure 7: *The object graph generated by* `rp2ps -object 10`*, shows that all objects are allocated at start of execution and alive to end of execution.*

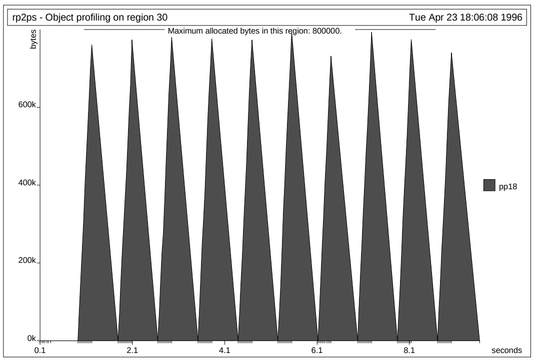


Figure 8: *Output from* `rp2ps -object 30`*. We have 10 peaks representing the 10 applications of* `loop`*.*

- allocation of one `CONS` cell, line 58

Allocation and deallocation of finite regions and objects are rather expensive when profiling, and thus, the time to return is not the same under normal execution.

As a little appetizer to section 6 we show a simple optimization on our example program from figure 4. We replace line 9 with

```
| loop n = (let val foo = map g liste in () end ; loop (n-1)).
```

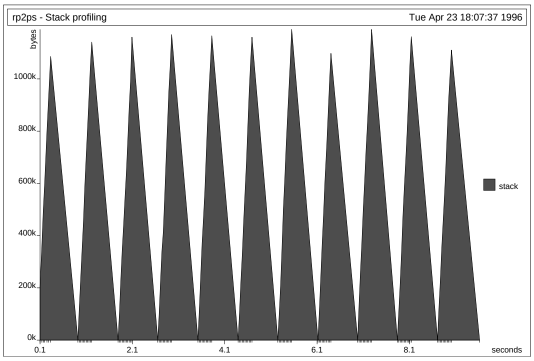


Figure 9: *Output from* `rp2ps -stack`*. We have 10 peaks representing the 10 applications of* `loop`*. Stack use is split in memory allocated for the infinite region stack (*`rDesc` *not shown on this graph) and an other part (*`stack`*) containing live variables across function calls and exception handlers. Data allocated in finite regions are not shown on a stack graph.*

```
23    letregion r10 : inf, r11 : inf
24    in
25      let val list =
26        letregion r13 : 2
27        in
28          longlist(* atbot r13  *)(atbot r10 ,atbot r11 ,100000)(* region vector for longlist pp8  *)
29        end (*r13 *)
30      in
31        letregion r14 : 1
32        in
33          let fun loop(r15 : inf, r16 : inf, var4)(* at r14  pp9  *) =
34            (case  var4 of
35                0 => ::((42,nil sat r15  pp12 with attop r16 )sat r16  pp11 )attop r15 pp10
36             |  _ => letregion r20 : inf, r21 : inf
37                     in
38                       let val foo =
39                             letregion r22 : 0
40                             in
41                               let fun map(r23 : inf, r24 : inf, var2)(* at r22  pp13  *) =
42                                 (case  var2 of
43                                    nil => nil sat r23  pp14 with sat r24
44                                   |  :: => let
45                                             val v11 = prim(decon(::), [var2]) pp15;
46                                             val v16 =
47                                               let
48                                                 val v17 = prim(+, [(#0 v11),1]);
49                                                 val v18 =
50                                                   letregion r30 : 2
51                                                   in
52                                                     map(* atbot r30  *)(sat r23 ,sat r24 ,(#1 v11))
                                                                     (* region vector for map pp18  *)
53                                                   end (*r30 *)
54                                               in
55                                                 (v17,v18)attop r24  pp20
56                                               end
57                                             in
58                                               ::(v16)attop r23  pp21
59                                             end) (* shared region clos. atbot r22  pp13  *)
60                               in
61                                 letregion r31 : 2
62                                 in
63                                   map(* atbot r31  *)(atbot r21,atbot r20 ,list)
                                                         (* region vector for map pp22  *)
64                                 end (*r31 *)
65                               end
66                             end (*r22 *)
67                           in
68                             letregion r33 : 2
69                             in
70                               loop(* atbot r33  *)(sat r15 ,sat r16,prim(-, [var4,1]))
                                                          (* region vector for loop pp23  *)
71                             end (*r33 *)
72                           end
73                         end (*r20  r21 *)) (* shared region clos. atbot r14  pp9  *)
74              in
75                letregion r36 : 2
76                in
77                  loop(* atbot r36  *)(attop r3 ,attop r2 ,10)(* region vector for loop pp25  *)
78                end (*r36 *)
79              end
80            end (*r14 *)
81          end
82        end (*r10  r11 *)
```

Figure 10: *Fragments of the region annotated lambda program generated when compiling example program* `let`*. In line 28 we call* `longlist` *shown on page 6.*

By putting `foo` inside a `let` expression the region inference becomes able to put a `letregion` $\rho_1, \ldots, \rho_n$ `in` ... `end` around the let expression. Because `foo` is dead after the body of the `let` expression, `foo` can be put in some of the regions ρ_i which are deallocated before the recursive call to `loop`. In figure 11 we see that this refinement saves roughly 18 Mb.

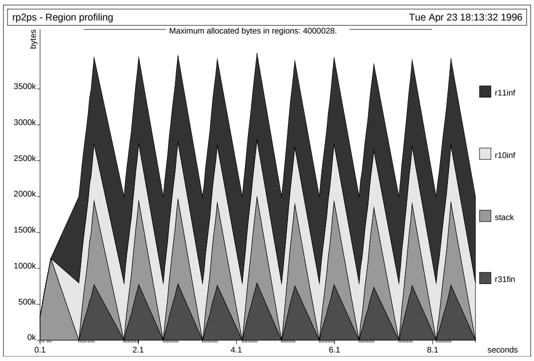


Figure 11: *The region graph shows the modified version of our* `let` *example, where* `foo` *is allocated in local regions. It is clear that the max. allocation line has been adjusted upwards by the size of the largest stack peak shown. The names of the region variables are not the same as on figure 5 and 10 because the ML source program has changed.*

5 Region Profiling in detail

In this section we go into details about the use of the region profiler. Besides detailing the three steps outlined in the previous section, we also look at region flow graphs, paths and the *VCG* tool. Section 5.1 – 5.4 and section 5.7 concerns the compile profiling strategy. Section 5.5 and 5.6 concerns the target profiling strategy.

5.1 Generating region flow graphs

When you compile the ML source program, two directed *region flow graphs* are generated. They are appended after the region annotated lambda program with program points on the log file. The first graph is almost equivalent to the graph used by the storage mode analysis (section 3.5) where region variables are nodes and an edge between two nodes ρ and ρ' is inserted if ρ is a formal region parameter of a function f which is applied to actual region parameter ρ'. This implies that `letregion` bound variables are always leaf nodes. The graph used by the storage mode analysis ([7]) deviates by also using effect variables as nodes.

The second graph is a Strongly Connected Component version of the first graph, where all nodes in each component are shown. The two graphs are described in section 3.6 and 3.8.

The region flow graph for our `let` example is as follows.

```
[Starting layout of graph...
global letregion[r2:inf]
global letregion[r3:inf]
letregion[r4:0]
longlist[r5:inf]   --r5 sat-->   [*r5*]    --r5 atbot-->   letregion[r10:inf]
longlist[r6:inf]   --r6 sat-->   [*r6*]    --r6 atbot-->   letregion[r11:inf]
letregion[r8:2]
letregion[r13:2]
letregion[r14:1]
loop[r15:inf]   --r15 sat-->   [*r15*]    --r15 attop-->   [*r3*]
loop[r16:inf]   --r16 sat-->   [*r16*]    --r16 attop-->   [*r2*]
letregion[r20:inf]
letregion[r21:inf]
letregion[r22:0]
map[r23:inf]   --r23 sat-->   [*r23*]    --r23 atbot-->   [*r21*]
map[r24:inf]   --r24 sat-->   [*r24*]    --r24 atbot-->   [*r20*]
letregion[r30:2]
letregion[r31:2]
letregion[r33:2]
letregion[r36:2]
...Finishing layout of graph]
```

Nodes in the graph are written in square brackets, where for example `longlist[r5:inf]` means that `r5` is a formal region parameter in function `longlist`. An asterisk inside a square bracket means that the node has been written earlier. Only the node identifier will then be printed. Region variables are node identifiers in the region flow graph. The region size is written after the region variable. The size printed for a finite region is in words. A few regions (named *global letregions*) are alive throughout the program. They always have size infinite.

Edges are written with the *from node* identifier inside the edge. The edge points at the *to node*. The text `--r24 atbot--> [*r20*]` is read: there is an edge from node `r24` to node `r20`. The maximal storage mode used when passing region `r20` as actual argument to formal region variable `r24` in a call to `map` is atbot (section 3.6).

When analysing region 20 (figure 6) we see that program point `pp20` is responsible for the allocations. In the lambda code (figure 10, line 55) we conclude that region variable `r24` is aliased with `r20`. This can also be seen in the region flow graph where we see an edge from node `[r24=map]` to `[*r24*]` (i.e. we have a cycle) and to `[*r20*]` which is `letregion` bound.

The *strongly connected component graph* contains the strongly connected components (scc) of the region flow graph. Each scc is identified with a *scc number* which is used as node identifier. The region variables contained in each scc are written as node info. All scc's contains only one region variable in our `let` example:

```
[Starting layout of graph...
[sccNo 21: r36,]
[sccNo 20: r33,]
[sccNo 19: r31,]
[sccNo 18: r30,]
[sccNo 17: r24,]   --sccNo 17-->   [sccNo 13: r20,]
[sccNo 16: r23,]   --sccNo 16-->   [sccNo 14: r21,]
[sccNo 15: r22,]
[sccNo 12: r16,]   --sccNo 12-->   [sccNo 1: r2,]
[sccNo 11: r15,]   --sccNo 11-->   [sccNo 2: r3,]
[sccNo 10: r14,]
```

```
[sccNo 9: r13,]
[sccNo 8: r8,]
[sccNo 7: r6,]   --sccNo 7-->   [sccNo 6: r11,]
[sccNo 5: r5,]   --sccNo 5-->   [sccNo 4: r10,]
[sccNo 3: r4,]
...Finishing layout of graph]
```

If you use incremental compilation ([8]), then only a region flow graph for the program being compiled now will be printed. Global region variables from earlier programs may be leaf nodes in the graph. We note, that if a global region variable from an earlier program is used as actual region parameter in a letrec application, then the storage mode passed will be attop.

5.2 Region flow paths

It can be difficult to find all *region flow paths* between two nodes in larger graphs. If you specify the *from node* and the *to node* before compiling the ML source program, then the ML Kit will find the possible paths. The algorithm used to find the paths are explained in section 3.8. You just have to type in a list of region variable pairs,

[(*formal reg. var. at pp., letregion bound reg. var.*), ...].

For instance, if you want to find all paths between region variable **r24** and **r20** in the **let** example, you activate **Paths between two nodes...** in the **Profiling** menu and type **[(24,20)]**. The following interaction

```
        Profiling

        0       region profiling............................ off
        1       extended region profiling................... on
        2       generate lambda code file with program points on
        3       generate VCG graph.......................... on
        4       Paths between two nodes..................... [] >>>
        5       Program points..............................  >>>

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

>4
<type an int pair list of region variables, ex. [(formal reg. var. at pp.,letregion bound reg. var.)]>
[(24,20)]
        Profiling

        0       region profiling............................ off
        1       extended region profiling................... on
        2       generate lambda code file with program points on
        3       generate VCG graph.......................... on
        4       Paths between two nodes..................... [(24,20)] >>>
        5       Program points..............................  >>>

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

>q
val it = () : unit
- evalFile "let";
```

prints

```
[...[Start path: [sccNo 12: r24,]--->[sccNo 9: r20,]]...]
```

after the region flow graphs in the log file.

5.3 The VCG tool

It is possible to export the region flow graphs onto a target file named **hpcode#.vcg** by setting **generate VCG graph...** on in the profiling menu. The graphs are exported in a format which can be read by the *VCG* (Visualization of Compiler Graphs) tool. The tool is started by typing **xvcg** *filename*, where *filename* is the file containing the exported graph. The *VCG* tool[12] can visualize the graphs graphically on screen or on paper. The tool, documented in [18][13], supports

[12] Information about the tool can be obtained from `http://www.cs.uni-sb.de/RW/users/sander/html/gsvcg1.html`. This report is based on version 1.30 found in file `vcg.1.30.r3.17.tar`.

[13] The documentation was included in file `vcg.1.30.r3.17.tar`, and has filename `visual.ps` found in the documentation directory `/VCG/vcg.1.30/doc/`.

- **subgraphs**. A graph consists of zero, one or more subgraphs.
- **folding**. A graph or subgraph can be folded to one node only.
- **hiding of edges**. Edges can be split in *edge classes*, and each class can be shown separately or together with other classes.

We export the two region flow graphs as two folded subgraphs, named "Region flow graph" and "SCC graph". A graph is then *unfolded* by choosing **Unfold Subgraph** from the pull down menu inside the `xvcg` window. The pull down menu is activated by pressing one of the mouse buttons. After activating **Unfold Subgraph** you have to pick the node(s) representing the graph(s) to unfold. This is done by clicking on the nodes with the left mouse button. Pressing the right mouse button will then unfold the chosen graphs.

Folding a graph is done by choosing **Fold Subgraph** in the pull down menu. You then click on a node, chosen at random, in the graph to fold with the left mouse button. Folding is activated by pressing the right mouse button.

Region Flow Paths

If the ML Kit finds some region flow paths, then they are exported with the region flow graph. Each path is numbered, and the path edges are put into the edge class represented by the *path number*. Edges in the region flow graph are also put in an edge class. It is now possible to see the region flow graph or the region flow paths separately.

The classes of edges shown in the `xvcg` window are specified by the **Expose/Hide edges** facility found in the pull down menu. For instance, the exported region flow graph for the `let` example with the path specification as in section 5.2 (`[(24,20)]`) gives the edge classes shown in figure 12. Each class number is identified by an informative text showing the contents of that edge class. The class representing the region flow graph is called "Graph". Naming of paths are done by the following pattern: `path#(`*`from node, to node`*`)`, where `#` is the path number. Path numbers are nice to have because there can be several paths between the same two nodes.

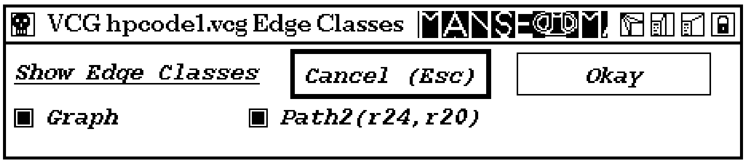


Figure 12: *The region flow graph has two edge classes. The graph edges are in class "Graph", and the path edges are in class "Path2(r24,r20)".*

Edges in the strongly connected components graph are not divided into edge classes.

If figure 13 we show an example region flow path as it will appear in the `xvcg` window. The path is from a region flow graph for a larger example program. It was a large region flow graph so it was not easy to find the path by inspection.

The *VCG* tool gives several facilities to control the view of a graph, which among others are *scaling*, *layout algorithm* and *printing*. Chapter five "Usage of the VCG tool" in the documentation explains every facility in detail.

5.4 Reducing the number of program points

You can reduce the number of program points shown in the region annotated lambda program by activating menu item `Program points`, toggle the shown button and activate `Print some program points`:

```
            Profiling

            0       region profiling............................ off
            1       extended region profiling................... on
            2       generate lambda code file with program points on
            3       generate VCG graph.......................... on
            4       Paths between two nodes..................... [(24,20)] >>>
            5       Program points.............................. >>>

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

    >5
            Profiling/Program points

            ============================  - - - - - - - - - - - - - - -
            | Print all program points  |    Print some program points
```

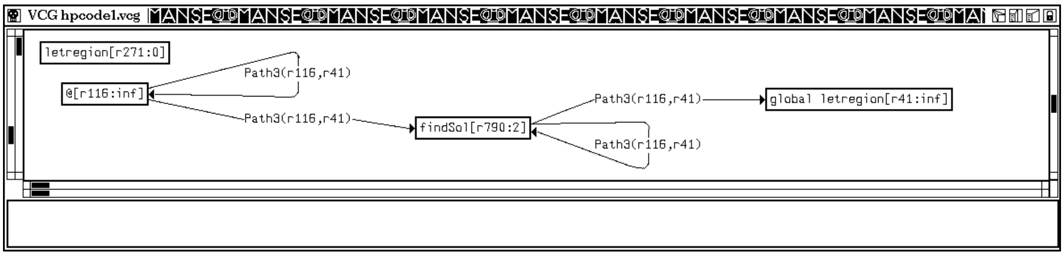


Figure 13: *The figure shows a region flow path. Storage modes are, at present, not annotated on the arrows (as on the region flow graph) but that would be an obvious improvement.*

```
        |            on              |           off (>>>)
        ============================  - - - - - - - - - - - - - - -


Toggle (t), Up (u), or Quit (q)

>t
        Profiling/Program points

        - - - - - - - - - - - - - - -  ============================
          Print all program points     | Print some program points |
                     off               |           on >>>          |
        - - - - - - - - - - - - - - -  ============================


Toggle (t), Activate chosen (a), Up (u), or Quit (q)
```

You then have to write the program points, that you want to see in the lambda program:

```
>a
        Profiling/Program points/Print some program points

        0       Program points to print [] >>>

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

>0
<type an int list, ex. [4,3]> or quit (q): >
[18,9]
        Profiling/Program points/Print some program points

        0       Program points to print [18, 9] >>>

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

>q
val it = () : unit
-
```

Only program point 9 and 18 will be shown on the region annotated lambda program.

5.5 Executing the target program with profiling

When the **hpcode** and **link** files are assembled and linked with the profiling version of the runtime system, then it is possible to give command-line options to the executable (say **a.out**) controlling the target profiling strategy. Typing **a.out -help** gives:

```
Help
You have compiled a ML source program under the ML Kit with profiling enabled.
This is the target program (a.out) which will generate a profile data file.
The help screen explains how you can set the profiling strategy.

usage: a.out [-notimer n | -realtime | -virtualtime | -profiletime]
      [-microsec n | -sec n]
      [-file outFileName]
      [-profTab] [-verbose] [-help]

where -notimer n        Profile every n'th function call.
```

```
        -realtime         Profile with the real timer.
        -virtualtime      Profile with the virtual timer.
        -profiletime      Profile with the profile timer.

        -microsec n       If a timer is chosen, then profile every n'th microseconds.
        -sec n            If a timer is chosen, then profile every n'th seconds.

        -file outFileName Use outFileName as profile datafile. Default is profile.rp

        -profTab          Print profiling table.
        -verbose          Verbose mode.
        -help             This help screen.
>
```

If you have to profile a very small program, then it is not always possible to specify a suitable time slot, and the `notimer` option can be used. It profiles at every `n`'th entrance to functions, and therefore gives more control when profiling small recursive functions. For instance `a.out -notimer 42` will profile at every 42th entrance to a function.

The UNIX implementation provides three different timers to specify time-slots. The following description is from the `getitimer` manual page for the HP–UX operating system, release 9.0.

```
The ITIMER_REAL timer decrements in real time.  A SIGALRM signal is
delivered when this timer expires.

The ITIMER_VIRTUAL timer decrements in process virtual time.  It runs
only when the process is executing.  A SIGVTALRM signal is delivered
when it expires.

The ITIMER_PROF timer decrements both in process virtual time and when
the system is running on behalf of the process.  It is designed to be
used by interpreters in statistically profiling the execution of
interpreted programs.  Each time the ITIMER_PROF timer expires, the
SIGPROF signal is delivered.  Since this signal can interrupt in-
progress system calls, programs using this timer must be prepared to
restart interrupted system calls.
```

By default the `virtualtime` timer is chosen. The time-slots can be specified in microseconds ($[1, \ldots, 999999]$) or in seconds ($[1, \ldots]$). Default is one second.

After execution summed statistics, as on page 19, is printed but by using option `-profTab`, then statistics on each region is printed too.

You can specify the profile data file with the `-file` option. Default is `profile.rp`.

`a.out -profiletime -microsec 42 -profTab` profiles with the profile timer ticking every 42th microsecond, and prints region statistics after execution.

`a.out -file MyFile -sec 20` profiles with the virtual timer ticking every 20th second, and saves the profile data in file `MyFile`.

Experience has shown that the virtual timer is adequate for most profiles. Using the virtual timer makes the profiles less dependent on the actual load on the machine.

5.6 Constructing graphs with rp2ps

The graph generator `rp2ps` is controlled with the following switches:

```
> rp2ps -help
Help
Help screen for graph generator rp2ps.
The graph generator creates one or more region graphs in postscript from a
profile data file.

usage: rp2ps [-region [filename]] [-stack [filename]] [-object regionNo [filename]]
       [-sampleMax n] [-sortByTime | -sortBySize]
       [-vert] [-eps width (in|mm|pt)]
       [-noOfBands n]
       [-comment time string] [-mark time]
       [-pregion regionNo] [-print] [-stat] [-findPrgPoint n]
       [-source filename] [-name name] [-interactive] [-help]

where -region        Profile all regions with respect to size (default output filename is region.ps).
      -stack         Profile stack with respect to size (default output filename is stack.ps).
      -object        Profile all objects in region regionNo.
                     (default output filename is object#.ps, where # is regionNo.
```

```
-sampleMax n    Use only n samples (default is 64).
-sortByTime     Choose sampleMax samples equally distributed over sample time.
-sortBySize     Choose the sampleMax largest samples.
                Default is sortByTime.

-vert           Put a vertical line in the region graph for each sample used.
-eps            Produce encapsulated postscript with specified width in
                inches (in), milli meters (mm) or points(pt).

-noOfBands n    Max. number of bands shown on the region graph. Default (and possible maximum) is 20.

-comment t str  Insert comment str (one word only) at time t in the region graph.
-mark    t      Insert mark at time t in the region graph.

-pregion n      Print region n on stdout.
-print          Print all profiling data on stdout.
-stat           Print some statistics on stdout.

-findPrgPoint n Print regions containing program point n.

-source name    Specify name of profile datafile (default is profile.rp).
-name           Name to print on top of region graph (default is rp2ps).

-interactive    Enter interactive mode.
-help           This help screen.
```

Using the `-region`, `-object` or `-stack` options makes it possible to explicitly give an output file name for the produced profile graphs. A heading (for example the name of the ML source program) in the profile graphs can be specified with `-name MyProgram` and comments are inserted with `-comment` *time str*, where *time* is a float in seconds and *str* is a string with one word only.

Profiling over larger time periods will often result in more profile-ticks to occur (also called *samples*) than one wants on the graph. The number of samples shown on the profile graphs are specified with option `-sampleMax` n. There are two methods to sort out samples:

- `-sortBySize` where the n (specified by `-sampleMax`) largest samples are kept. It is likely that the samples shown will be grouped on the x-axis because normally a program only has its maximum memory consumption in short periods compared to the total execution time.
- `-sortByTime` is used by default and makes a binary deletion of samples by time such that the n samples shown will be equal distributed on the x-axis.

Option `-sortBySize` is handy if you get some profile graphs with a large gab between the top band and the *max. allocation line*. If there is a large gab when using option `-sortBySize`, then you have to profile with smaller time-slots.

It is not always a good idea to show twenty bands on a region graph, because it can be difficult to discern between the shown bands. Option `-noOfBands` n sets the maximum number of bands shown to n. All other bands are gathered in an *other* band drawn at the bottom of the graph.

Allocation points inside region polymorphic functions can allocate into several regions. Option `-findPrgPoint` n prints all regions which (in a profile tick) have had a value allocated by the allocation point identified with program point n.

Examples on using `rp2ps`:

`rp2ps -region MyRegion -object 10 MyObject` produces a *region graph* in file `MyRegion` and an *object graph* of region 10 in file `MyObject`.

`rp2ps -region -sortBySize -sampleMax 100` produces a *region graph* in file `region.ps` where the 100th largest sampels are shown.

`rp2ps -stack -comment 1.34 MyComment` produces a *stack graph* in file `stack.ps` with the word `MyComment` inserted at time 1.34.

`rp2ps -region -comment 1.42 FirstWord SecondWord` will terminate (producing no profile graphs) because it is not allowed to use more than one word in a comment.

5.7 Profiling from inside the ML source program

It is possible to control the profiler from inside the ML source program. The profiler supports three primitive ML functions.

`profile_on:unit→unit;` turns the profiler alarm on.

`profile_off:unit→unit;` turns the profiler alarm off.

`profile_tick:unit→unit;` performs a profile tick.

It is not possible to control the profiling facility corresponding to the `-notimer` option found in section 5.5. A similar facility can be programmed manually with a counter and the `profile_tick` primitive.

It is allowed to use the primitives in the ML source program without having profiling activated. The primitives have no semantic effect then. If the ML source programs are used in other ML compilers, then you can declare `fun profile_on() = ()` etc. and thereby avoiding changing the program.

If you want the profiler to support more primitive ML functions (for example `profile_printStat` which prints profiling statistics) then you can implement the facility as a C function (say `printStat`) and call `printStat` with the external C call facility found in the ML Kit [9]. The three primitive functions above and other low level profiling functions are implemented as an extension to the runtime system written in C.

6 Hints on compiling with the ML Kit

There are several situations where the target program uses more memory than expected. In this section we present some hints on avoiding space leaks in the ML Kit. Many of the techniques presented here are also found in [23] and [4]. The aim of the ML Kit is to make fast and space efficient ML programs without having a garbage collector interrupting the execution. With the profiler we hope that the user can remove as many space leaks as necessary to get satisfactory target programs.

6.1 Letrec bound functions

Letrec-bound functions introduce region polymorphism. The principal guide line is that the more region polymorphism the better. If the function f with type scheme $\forall\vec{\alpha}\vec{\rho}\vec{\epsilon}.\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2$ has the same region variables in argument and result type then we have a *potential space leak*.[14] For instance, if the function map takes a list as argument in region ρ_1 and returns a new list also in region ρ_1, then the argument list will stay allocated even though it may be dead. This behaviour can be disastrous in programs manipulating large data structures. However, for tail recursive functions we will see that having the same regions in the argument and result type can be advantageous.

Having the same regions in the argument and result type can be dangerous in non tail recursive functions producing many small temporaries. In function *f_endo* (to heighten readability we give the example programs in SML syntax rather than in region annotated lambda program syntax)

```
fun calc_endo x = if false then x else x - 1.0
fun f_endo x = if x = 42.0 then x else (f_endo(calc_endo x)+x)
```

we use the rule for conditionals (i.e., rule 4, page 8) to elaborate `if` x `=`.... Type and place for x and $f\_endo(calc\_endo\ x) + x$ have to be the same. Variable x is elaborated by rule 1 and gets same type and place as the argument. The function gets type $\forall\rho_1\epsilon.(\texttt{int}, \rho_1) \xrightarrow{\epsilon.\varphi} (\texttt{int}, \rho_1)$. The problem is that by returning x in the then branch we do not create a new value which can be put in a fresh region. If we instead write

```
fun f_exo x = if x = 42.0 then 42.0 else (f_exo(calc_endo x)+x)
```

then *f_exo* gets type $\forall\rho_1\rho_2\epsilon.(\texttt{int}, \rho_1) \xrightarrow{\epsilon.\varphi} (\texttt{int}, \rho_2)$. The value of *calc_endo* x will pile up in ρ_1, but now it is more likely that ρ_1 will be deallocated after return even though the result is still alive. In figure 14 we see the space consumption of the following program.

```
val a = f_exo 100000.0
val b = f_endo 100000.0
val c = f_exo 100000.0
val b = f_endo 100000.0
```

The figure shows that temporaries are freed after *f_exo* and not after *f_endo*.

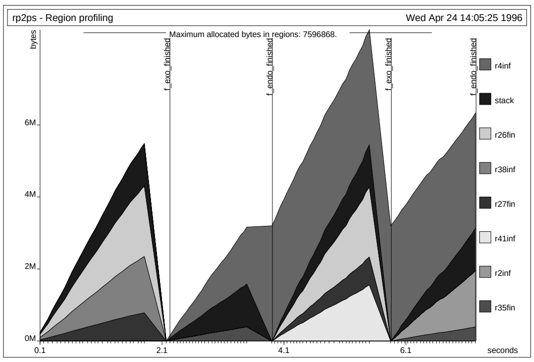


Figure 14: *It is obvious, that memory allocated by the f_exo calls is deallocated, but memory from the f_endo calls is not deallocated.*

Functions having all the regions from the argument type (μ_1) to also appear in the result type (μ_2) are in [23] called *endomorphisms* ($\text{frv}(\mu_1) \subseteq \text{frv}(\mu_2)$). An endomorph map function can be written

```
fun map_endo f xs = case xs of
                      [] => xs
                    | (x::xs) => (f x) :: (map_endo f xs)
```

[14] We use the words *potential space leak* broadly. A potential space leak may not turn out to be a space leak at all. Moreover, as we will see, it happens that what we thought to be a space leak can turn out to be the opposite.

For a function f with type $\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2$ where $\mathrm{frv}(\mu_1) \cap \mathrm{frv}(\mu_2) = \emptyset$ there is so much flexibility in choosing region variables that it is likely that f will not introduce space leaks. These functions are in [23] called *exomorphisms*. By not reusing xs in the case for the empty list we make the above map function exomorphic.

```
fun map_exo f xs = case xs of
                      [] => []
                    | (x::xs) => (f x) :: (map_exo f xs)
```

Usually the latent effect (φ) of a function type $(\forall\vec{\alpha}\vec{\rho}\vec{\epsilon}.\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2)$ contains get effects on regions in μ_1, put effects on regions in μ_2 and effects on region variables due to non local temporaries. If the latent effect contains put effects on region variables not in $\vec{\rho}$ and even worse not in μ_2 then we have a potential space leak. By activating **warn on escaping put effects** in the interact environment the ML Kit will warn on functions having a type scheme with put effects on regions not in $\vec{\rho}$ or μ_2.[15]

6.2 The single copy trick

The *single copy trick* can always be used to turn an endomorph function into an exomorph function. At the places where the result is returned a single exomorphic copy operation is inserted. The copy operation must create a new value, so the following polymorphic functions do not work:

```
fun copy x = x

fun copy x = let
               val a = x
             in
               a
             end

fun copy x = let
               val a = ref x
             in
               !a
             end
```

The following functions do work:

```
fun copy_int x = x+0

fun copy_real x = x+0.0

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

6.3 Losing region polymorphism with exceptions

In the ML Kit, exception constructors will always be in global regions (rule 13 and 11). It would be beneficial if one could make an analysis which could find more precise live ranges for exceptions, such that exceptions that are guaranteed to be handled could be given local regions.

Using exceptions in the ML Kit can be very space consuming because applying exception constructors to values forces the values to be in global regions too.

```
exception fail of real

fun fail_exn x = fail x

fun fac n acc = if n <= 1.0 then acc else fac (n-1.0) (acc*n)

fun loop 0 = fail_exn (fac 1000.0 1.0)
  | loop n = (fail_exn (fac 1000.0 1.0);
              loop (n − 1))

val b = loop 1000
```

[15]The interact function is explained in [24]. Warn on escaping put effects is found in path: `Region Inference / Warnings`.

In the above program function *fail_exn* will not be region polymorphic in its argument nor result type. The function $\lambda x.fail\ x$ has type $(\texttt{real}, \rho_1) \xrightarrow{\epsilon.\varphi} (\texttt{exn}, \rho_2)$, where ρ_1 and ρ_2 are global regions. It is not possible to quantify the two global regions when inferring the type for *fail_exn*. Therefore applying *fail_exn* to *fac* in the loop forces the result value (*acc*) from *fac* to be in a global region. For each call of the form *fac* `1000.0 1.0`, one thousand reals are stored in a global region (see figure 15 left).

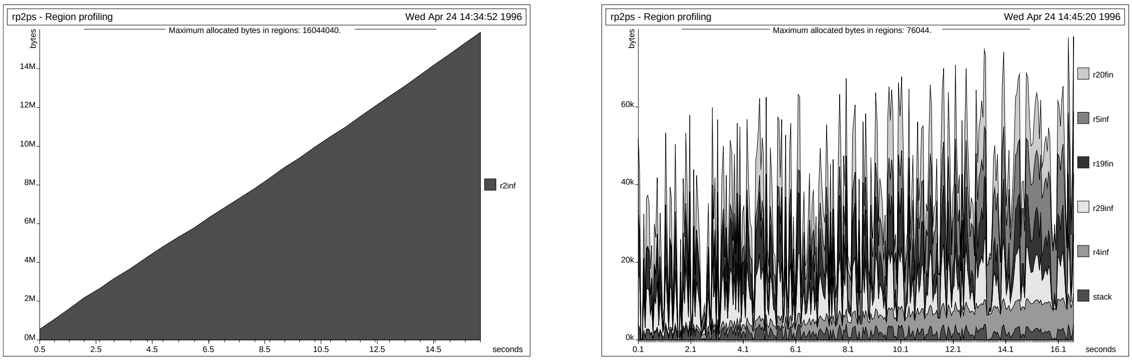


Figure 15: *To the left we have one global region containing all temporaries from fac. To the right all memory allocated for temporaries in fac are deallocated in each iteration of loop. Notice the values on the y-axis on the two figures.*

Inserting an exomorphic copy operation on the argument in *fail_exn* makes *fail_exn* region polymorphic in its argument:

```
fun copy x = x + 0.0

fun fail_exn' x = fail (copy x)
```

Now the result from *fac* is not forced into a global region, and the thousand reals can be stored in a region local to function *loop* (see figure 15).

6.4 References

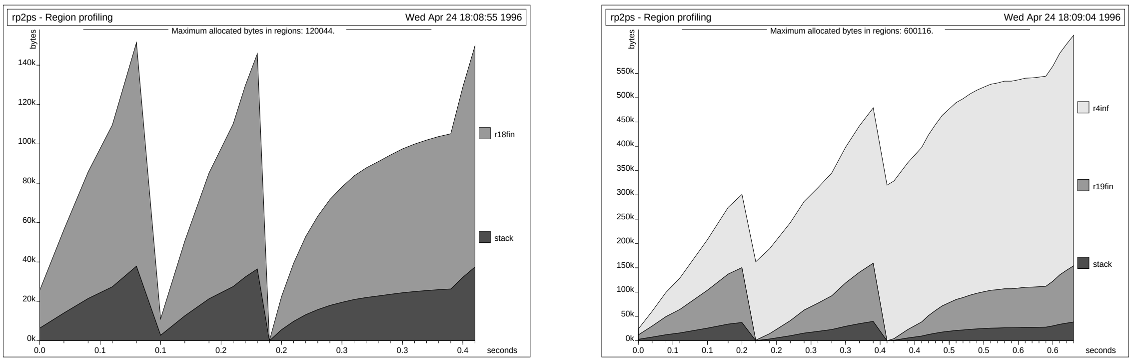
Updating references with the `ASSIGNprim` primitive (rule 19) forces the new and old value to have same type and place. If the values are unboxed (only takes up one word; an integer for example) then there will not be allocated more memory (the old value will be overwritten). The following program only uses a small amount of memory.

```
fun loop 0 f = f ()
  | loop n f = (f ();
                loop (n-1) f)

val counter = ref 0
fun inc () = (counter := (!counter + 1);
              !counter)

val res = (loop 10000 inc;
           loop 10000 inc;
           loop 10000 inc)
```

If the values are boxed (takes up more than one word) they will be allocated in a region and the pointer to the old value will be overwritten with a new pointer. The values are forced into the same region. Therefore changing the value from `int` to `real` in the previous program increases the memory use significantly (see figure 16).

```
val counter = ref 0.0
fun inc () = (counter := (!counter + 1.0);
              !counter)
```

It is therefore advantageous to declare references as local as possible.



(a) (b)

Figure 16: *The graph (a) shows the integer counter example. Region* **r18fin** *holds all closures of "loop n" in the loop iteration, which is applied to the function inc. The graph (b) shows the real counter example. Here region* **r19fin** *holds closure "loop n". Both region* **r18fin** *and* **r19fin** *are deallocated after return from the first call to loop. Region* **r4inf** *holds all reals used by the real counter and they are never deallocated.*

6.5 Let bound variables and eta conversion

By the let rule 7 (page 8) we see that let bound variables are not region polymorphic. Using non local let bound variables in letrec functions can make them less region polymorphic. For instance if we have a globally defined error message

```
val errorStr = ``Crash - global error occurred''
```

and a function *test*

```
fun test x = if ok(x) then x else errorStr
```

then the string x is forced into the same region as $errorStr$ no matter how local x is to the function where $test$ is applied. Function $test$ $(\forall\epsilon.(\texttt{string}, \rho_{errorStr}) \xrightarrow{\epsilon.\varphi} (\texttt{string}, \rho_{errorStr}))$ will neither be region polymorphic in its argument nor in its result type.

This space leak can be avoided by inlining the creation of $errorStr$ into the test function by

```
fun test x = if ok(x) then x else ``Crash - global error occurred''
```

or

```
fun test x = let
               val errorStr = ``Crash - global error occurred''
             in
               if ok(x) then x else errorStr
             end
```

Both of these suffer under a normal programming style where some global values are created once for all and then used throughout the program. However eta converting $errorStr$ will delay the creation of the string until needed and make the creation region polymorphic.

```
fun errorStr () = ``Crash - global error occurred''

fun test x = if ok(x) then x else errorStr ()
```

Function $test$ is still endomorph, but now it is region polymorphic in its argument $(\forall\rho_x\epsilon.(\texttt{string}, \rho_x) \xrightarrow{\epsilon.\varphi} (\texttt{string}, \rho_x))$.

6.6 Function composition and intermediate results

Because function composition is very useful in functional languages, we show how region inference "treats" function composition by inferring the type for an example composition. In $h = g \circ f$ the result of applying f to an argument is automatically passed to function g. In an environment with garbage collection, the memory used for the intermediate result from f to g will automatically be reused if there is no need for it.

In the ML Kit, however, the intermediate result will only be deallocated if it is possible to put it into a region which is only alive around $g \circ f$. Therefore, it is very important that both f and g are exomorphic.

Consider an exomorphic function g with type and place

$$\mu_g = (\forall \rho_{g1}\rho_{g2}\epsilon_g.(\mathtt{int}, \rho_{g1}) \xrightarrow{\epsilon_g.\varphi_g} (\mathtt{int}, \rho_{g2}), \rho_g), \quad \text{where} \quad \varphi_g = \{\mathbf{get}(\rho_{g1}), \mathbf{put}(\rho_{g2})\}$$

and an endomorph function f with type and place

$$\mu_f = (\forall \rho_{f1}\epsilon_f.(\mathtt{int}, \rho_{f1}) \xrightarrow{\epsilon_f.\varphi_f} (\mathtt{int}, \rho_{f1}), \rho_f), \quad \text{where} \quad \varphi_f = \{\mathbf{get}(\rho_{f1}), \mathbf{put}(\rho_{f1})\}.$$

We will now show how the type for $g \circ f$ is inferred. Consider the expression

$$\mathtt{letrec}\ h : (\pi, \rho_h)\ [\rho_{f1}, \rho_{g2}]\ x\ \mathtt{at}\ \rho_h \ = \ g\ [\rho_{g1}, \rho_{g2}]\ (f\ [\rho_{f1}]\ x)\ \mathtt{in}\ e\ \mathtt{end}$$

with

$$TE = \{g : \overbrace{(\forall \rho_{g1}\rho_{g2}\epsilon_g.(\mathtt{int}, \rho_{g1}) \xrightarrow{\epsilon_g.\varphi_g} (\mathtt{int}, \rho_{g2}), \rho_g)}^{\mu_g}; f : \overbrace{(\forall \rho_{f1}\epsilon_f.(\mathtt{int}, \rho_{f1}) \xrightarrow{\epsilon_f.\varphi_f} (\mathtt{int}, \rho_{f1}), \rho_f)}^{\mu_f}\}.$$

The expression is first elaborated by the letrec rule (8). The premises in the letrec rule imply (via the lambda rule 5) that the letrec application rule (9) is used twice on $\lambda\ x.g\ [\rho_{g1}, \rho_{g2}]\ (f\ [\rho_{f1}]\ x)$. The second time we use the letrec rule we get

$$TE + \{h : (\pi, \rho_h)\} + \{x : (\mathtt{int}, \rho_{f1})\} \vdash f\ [\rho_{f1}]\ x : (\mathtt{int}, \rho_{f1}), \varphi_f \cup \{\epsilon_f, \mathbf{get}(\rho_f), \mathbf{put}(\rho_f)\}.$$

The first time we use the letrec rule we get

$$TE{+}\{h : (\pi, \rho_h)\}{+}\{x : (\mathtt{int}, \rho_{f1})\} \vdash g\ [\rho_{g1}, \rho_{g2}]\ (f\ [\rho_{f1}]\ x) : (\mathtt{int}, \rho_{g2}), \varphi_g {\cup} \varphi_f {\cup} \{\epsilon_g, \epsilon_f, \mathbf{get}(\rho_g, \rho_f), \mathbf{put}(\rho_g, \rho_f)\},$$

where we unify ρ_{g1} and ρ_{f1} to, say ρ_{f1}.

The lambda rule (5) now gives

$$TE{+}\{h : (\pi, \rho_h)\} \vdash (\lambda^{\epsilon_h.\varphi_h} x : (\mathtt{int}, \rho_{f1}).g\ [\rho_{g1}, \rho_{g2}]\ (f\ [\rho_{f1}]\ x))\ \mathtt{at}\ \rho_h : ((\mathtt{int}, \rho_{f1}) \xrightarrow{\epsilon_h.\varphi_h} (\mathtt{int}, \rho_{g2}), \rho_h), \{\mathbf{put}(\rho_h)\}$$

with $\varphi_h = \varphi_g \cup \varphi_f \cup \{\epsilon_g, \epsilon_f, \mathbf{get}(\rho_g, \rho_f), \mathbf{put}(\rho_g, \rho_f)\} \cup \varphi'_h$

Quantifying $\tau = (\mathtt{int}, \rho_{f1}) \xrightarrow{\epsilon_h.\varphi_h} (\mathtt{int}, \rho_{g2})$ in the letrec rule gives type scheme $\pi = \forall \rho_{f1}\rho_{g2}\epsilon_f\epsilon_g\epsilon_h.\tau$.

We see that function h is exomorphic, but the intermediate result is allocated in ρ_{f1} and will be alive as long as the argument to h is alive. It is not possible for the letregion rule (10) to insert a letregion on ρ_{f1} around $g\ [\rho_{g1}, \rho_{g2}]\ (f\ [\rho_{f1}]\ x)$ after the application because $\rho_{f1} \in \mathrm{frv}(TE + \{h : (\pi, \rho_h)\} + \{x : (\mathtt{int}, \rho_{f1})\})$.

Having f exomorphic $(\forall \rho_{f1}\rho_{f2}\epsilon_g.(\mathtt{int}, \rho_{f1}) \xrightarrow{\epsilon_f.\varphi_f} (\mathtt{int}, \rho_{f2}), \rho_f)$ with $\varphi_f = \{\mathbf{get}(\rho_{f1}), \mathbf{put}(\rho_{f2})\}$ gives h the same type scheme (π) but now ρ_{f2} and ρ_{g1} have been unified. It is then possible to insert a letregion on ρ_{f2} around $g\ [\rho_{g1}, \rho_{g2}]\ (f\ [\rho_{f1}, \rho_{f2}]\ x)$ because $\rho_{f2} \notin \mathrm{frv}(TE + \{h : (\pi, \rho_h)\} + \{x : (\mathtt{int}, \rho_{f1})\})$ and $\rho_{f2} \notin \mathrm{frv}(\mathtt{int}, \rho_{g2})$.

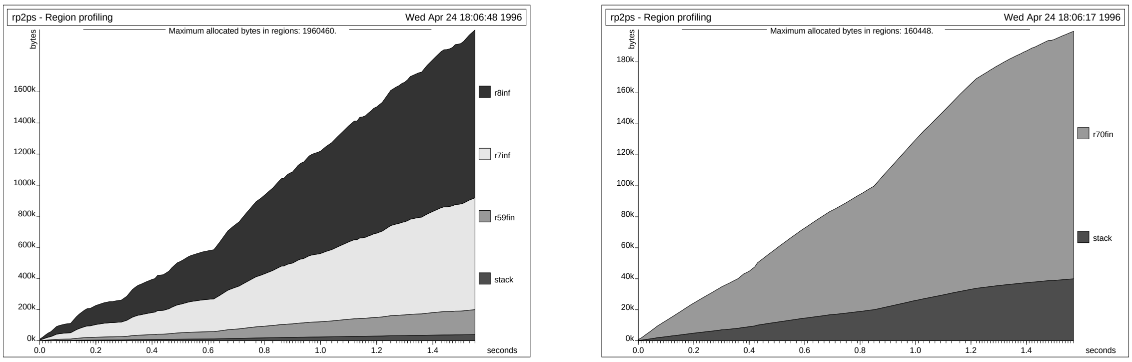
In figure 17 we see the difference of having the function *map* in the following program endomorph or exomorph.

```
fun inc x = x+1

fun map' x = map inc x

val global_list = [1,2,3,4,5,6,7,8,9]

fun length_exo [] = 0
  | length_exo (x::xs) = 1 + (length_exo xs)

fun length x = (length_exo o map') x

fun loop 0 f x = f x
  | loop n f x = (f x;
                  loop (n-1) f x)

val a = loop 10000 length global_list
```

If you change the role of f and g in $g \circ f$ above (let g be endomorph and f be exomorph), then the temporaries will be allocated as long as the result is alive.

6.7 Eta converting let bound functions

Let bound functions are not region polymorphic (rule 7). Given a region polymorphic function f then writing `let val` $g = f$ `in` e `end` loses the region polymorphism. However, the front end of the ML Kit translate all declarations of the form `let val` h `=` $\lambda\ x.e$ into `let fun` $h\ x$ `=` $(\lambda\ x.e)\ x$ so all let bound functions should be region polymorphic if possible.



(a) (b)

Figure 17: *The graph (a) shows memory use, when the map function is endomorph. Region* **r8inf** *holds the pairs and region* **r7inf** *the cons cells for the list generated by map. Region* **r59inf** *holds the closures for "loop n f". The graph (b) shows memory use when the map function is exomorph. There is no temporary list. Here region* **r70fin** *holds the closures for "loop n f".*

6.8 Lifetime of let bound variables and sequences

By the let rule (7) and letregion rule (10) we see that a region holding a let bound value is live to the end of the let construct. It is therefore advantageous to insert the value binding as close as possible to the expression where it is used. Instead of writing

```
let val x = calculate_x()
in let val y = calculate_y(x)
   in
     calculate_e(y)
   end
end
```

where x is used in y only, one should write

```
let val y =
  let val x = calculate_x()
  in
    calculate_y(x)
  end
in
  calculate_e(y)
end
```

As shown in the example under the let rule on page 9, results of all expressions in a sequence $(e_1;\ldots;e_n)$ are kept allocated until the end of the sequence. If expression e_i produces a space consuming result, then it can be advantageous to bind e_i in a local let construct and return a dummy value (like `()`) in the body of the let construct:

$$(e_1;\ldots;e_{i-1};\texttt{let \_} = e_i \texttt{ in } () \texttt{ end}; e_{i+1};\ldots;e_n)$$

This is what we did with the let example on page 22.

6.9 Tail recursive functions

The ML source program is translated to a language with letregion constructs. It is therefore not always the case that a tail recursive function in the source program is tail recursive in the region annotated lambda program. For instance the function *non_tail*

```
fun non_tail x =
  if x = 1.0
    then 1.0
    else let val y = x+1.0
         in
           non_tail y
         end
```

is not tail recursive in the region annotated lambda language because the region holding y (`r39`) has to be deallocated after the call:

```
fun non_tail(r33 : 4, var4)(* at r2  pp21  *) =
  let val v26 =
...
    letregion r39 : 4
    in let val y = letregion r40 in prim(+, [var4,1.0 atbot r40 ])atbot r39 end (*r40*)
       in letregion r41 in non_tail(*atbot r41*)(sat r33 ,y) end (*r41*)
       end
    end (*r39*)
...
```

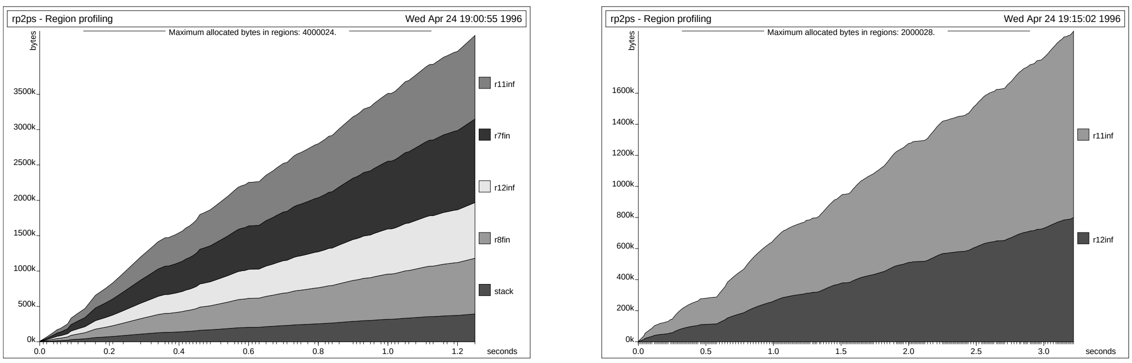
Making function *non_tail* endomorph will make it tail recursive because variable x and y are put in the same region. So this is an example where the opposite of the general guide line (see section 6.1) is beneficial.

A tail call in the ML Kit has the advantage that no live variables are pushed onto the stack and no region vector is build when the call is invoked. However, because the function is endomorph, it is vital that the storage mode analysis can annotate the allocation of the next argument atbot. If not, all arguments remain allocated as long as the "final" result of all the function calls is alive.

If the argument in each tail call is allocated atbot, then we have an "optimal" calling convention, because only one argument will ever be alive and no variables are pushed onto the stack due to the call.

In the following program we have two functions, where *intlist* is exomorph and *intlist'* is endomorph and tail recursive.

```
fun intlist (0,l) = l
  | intlist (n,l) = intlist (n-1,n::l)

fun intlist' (arg as (0,l)) = arg
  | intlist' (n,l) = intlist' (n-1,n::l)

val i =
  let
    val b = intlist (100000,[])
  in
    ()
  end

val i' =
  let
    val b = intlist' (100000,[])
  in
    ()
  end
```



(a) (b)

Figure 18: *The graph (a) shows the non tail recursive function intlist. The graph (b) shows the tail recursive function intlist', where all arguments are stored atbot. The tail recursive version saves roughly two megabytes of memory, compared to the non tail recursive version.*

The region graph at the left on figure 18 shows memory consumption by the non tail recursive *intlist*. All argument pairs are allocated in region `r7fin`. The region vector is put in region `r8fin` and we see that live variables are pushed onto the stack. Region `r11inf` and region `r12inf` hold the result list.

The other region graph only shows two regions (`r11inf` and `r12inf`) holding the result list. No arguments, region vectors or live variables are stored anywhere.

If we change value *i'* above to

```
val arg = (100000,[])
val i' =
  let
    val b = intlist' arg
  in
    ()
  end
```

then the storage mode passed for the argument region in the call to *intlist'* will be attop. All argument values produced in function *intlist'* will be allocated as long as the global variable *arg* is alive. If we use *intlist* instead, then the arguments produced by *intlist* will be deallocated when the whole list has been computed.

6.10 Curried functions

If there is a λ between the binding of a region variable ρ and an allocation into ρ, then the storage mode for that allocation will be attop ([7]). The storage mode has to be attop because a lambda abstraction can escape and it is not necessarily safe to reset the region ρ at all states in the program where the lambda abstraction can be applied.

All functions in the target program take one argument only. Curried functions are implemented by a sequence of lambda abstractions even if they are always fully applied. The function declaration **fun** $f\ x\ y\ =\ e$ is translated into **letrec** $f\ [\rho_1, \ldots, \rho_n]\ x\ =\ \lambda y.e$. All allocations in e involving ρ_i will have storage mode attop. Un-curry to **fun** $f\ (x, y)\ =\ e$ solves the problem:

```
fun add_c (a:real) b = a+b

fun add_uc ((a:real), b) = a+b
```

The function *add_c* will store the result attop in region `r5`. The function *add_uc* stores the result "somewhere at" in region `r8`.

```
fun add_c(r4 : 3, r5 : 4, var1) =
  (fn (* sat r4  pp3  *) var2 => prim(+, [var1,var2])attop r5  pp4 )

fun add_uc(r8 : 4, var3) = prim(+, [(#0 var3),(#1 var3)])sat r8  pp6
```

6.11 The double copy trick

The *double copy trick* is a method to perform something resembling user-controlled copying garbage collection on a part of the allocated memory. Suppose that we have an iteration (or computation) over a large recursive datatype where parts of the datatype are changed in each iteration. If the data is kept in the same region then it can grow quite large because the dead data is never de-allocated.

The storage mode analysis can be used to reset regions holding one datatype value. At a given point in the iteration two exomorphic copy operations are used to deallocate the memory for the dead data. The first copy moves the live data into a fresh region; then, if the original region only contains dead data, it is possible to perform a reset operation on the original region before the live data is copied back. If we let *compute* be the function which manipulates the data, and *timeToGC* return true if it is time for a garbage collection, then an iteration can then be programmed as

```
fun iter (arg as (0, data)) = arg
 | iter (n, data) =
   let
     val data' = compute data
   in
     iter (n-1, if (timeToGC 25)
                then doGC (data')
                else data')
   end
```

where the function *doGC* uses an exomorphic copy operation on the data type twice. Function *iter* is endomorph, and tail recursive. Only one argument pair (n, *data*) will be alive at any time. The function *iter* has the type scheme:

$$\forall \rho_1, \rho_2, \rho_3, \epsilon.(((\mathtt{int}, \rho_1) * (data, \rho_2), \rho_3) \xrightarrow{\epsilon.\varphi} ((int, \rho_1) * (data, \rho_2), \rho_3), \rho_4)$$

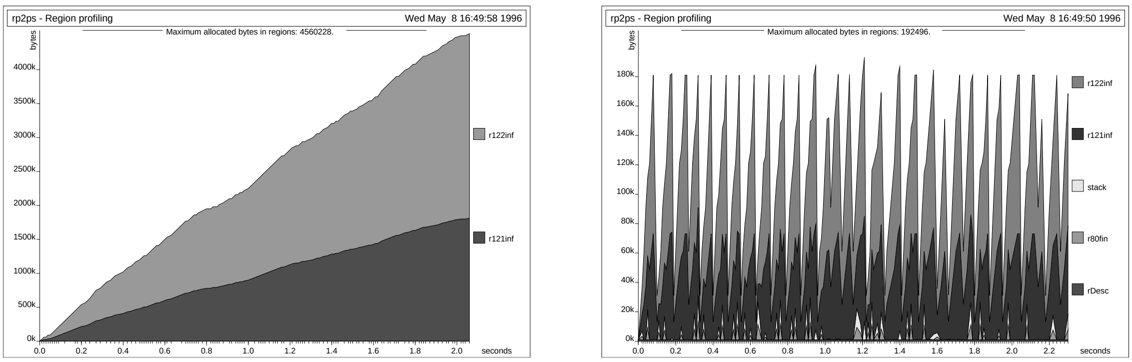
where we assume that the *data* elements reside in one region (ρ_2) only.

How often *timeToGC* shall return true is a trade off between execution time and memory consumption.

We now show two examples where the first example only needs one copy operation for a garbage collection and the second example neeeds two. In the first example we create a large list and make some manipulations on the elements.

```
datatype int_list = C of int * int_list
                  | N

fun makeList 0 = N
  | makeList n = C (n, makeList (n-1))

fun computeList (arg as N) = arg
  | computeList (C (n, l)) = C(n+1, compute l)

fun copy N = N
  | copy (C (n, l)) = C (n, copy l)

fun doGC data = copy data

local
  val counter = ref 0
  fun inc () = (counter := (!counter + 1);
                !counter)
in
  fun timeToGC x = ((inc ()) mod x) = 0
end

val result =
  let
    val l = makeList 1500
  in copy(#2(iter (150, l)))
end
```

Figure 19 shows the memory use of the above program with a garbage collection frequency of 1000 (no garbage collection with 150 iterations) and with a garbage collection frequency of 5.



(a) (b)

Figure 19: *The graph (a) shows how 150 instances of the same list is stored in region* **r121inf** *and* **122inf**. *Region* **r121inf** *holds the constructors for the list constructed in makeList and computed in computeList. Region* **r122inf** *holds the pairs for the list elements. The graph (b) shows how the two regions (***r121inf** *and* **r122inf***) are garbage collected.*

A nullary constructor occupies one word in memory, and a unary constructor occupies two words plus the value. A pair occupies three words. The list generated therefore occupies: one word for the `N` constructor and $1500 * 5 = 7500$ words for the `C` constructors with values. This gives a total of 30004 bytes for the list. The elements in the tree is replaced each time the list is recomputed, which adds up to 30000 bytes. The version without garbage collection therefore allocates 4530004 bytes for the list. The version with garbage collection for every 5'th computation allocate memory for the original list plus five computed list alive at the same time. That sums up to 180004 bytes.

We now look at a simple binary tree where we can show the double copy trick in detail. We use the datatype

```
datatype tree = NODE of tree * int * tree
              | LEAF
```

with the following functions

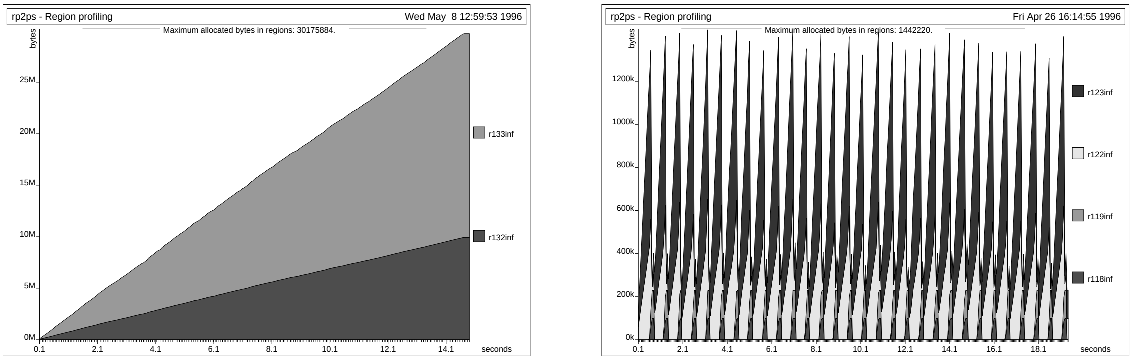
```
fun makeTree 0 = LEAF
  | makeTree h = NODE (makeTree (h-1), h, makeTree (h-1))

fun compute (arg as LEAF) = arg
  | compute (NODE (t1, n, t2)) = NODE(compute t1, n+1, compute t2)

fun copy LEAF = LEAF
  | copy (NODE (t1, n, t2)) = NODE (copy t1, n, copy t2)

fun doGC t =
  let
    val t' = copy t
  in
    (resetRegions t;
     copy t')
  end

val result =
  let
    val tree = makeTree 13
  in copy(#2(iter (150, tree)))
end
```

The function *doGC* contains the two copy functions and a function *resetRegions*. The function *iter* used above is found on page 39. If we look at the region annotated lambda program for the copy operation we see that it has not been possible for the storage mode analysis to pass storage mode "somewhere at" in the two recursive calls.

```
fun copy(r83 : inf, r84 : inf, var14)(* at r9  pp71  *) =
  (case  var14 of
       LEAF$49 => LEAF$49 sat r83  pp72   with sat r84
    |  NODE$49 =>
       let val v90 = prim(decon(NODE$49), [var14]) pp73 ;
           val n = (#1 v90);
           val t2 = (#2 v90);
           val v94 =
               let val v95 =
                       letregion r88 : 2
                       in copy(* atbot r88  *)(attop r83 ,attop r84 ,(#0 v90))
                       end (*r88 *);
                   val v97 =
                       letregion r89 : 2
                       in copy(* atbot r89  *)(attop r83 ,attop r84 ,t2)
                       end (*r89 *)
               in  (v95,n,v97)attop r84  pp79
               end
       in  NODE$49(v94)attop r83  pp80
       end)
```

Even though the allocation of a *LEAF* constructor is annotated somewhere at, the recursive call to *copy* with storage mode attop for the result regions (`r83` and `r84`) will always allocate *LEAF* attop. We therefore have to reset the regions containing value t after the first copy operation by hand. The ML Kit provides a *resetRegions* primitive (explained in [20]) which will reset all regions free in the type and place of value t if storing a value into the regions at that point in the program would reset the regions. The primitive is safe in the sence that only regions containing no live data will be reset. It is obvious, that value t is not used after the first copy operation so it is possible for the *resetRegions* primitive to reset the regions free in the type and place of t.

Figure 20 shows the memory use of the above program. The figure shows a version with a garbage collection frequency of 1000 (no garbage collection with 150 iterations) and with a frequency of 5.



(a) (b)

Figure 20: *The graph (a) shows how 150 instances of the same tree is stored in region* **`r133inf`** *and* **`132inf`**. *Region* **`r132inf`** *holds the constructors for the tree constructed in makeTree and computed in compute. Region* **`r133inf`** *holds the triples for the nodes. The graph (b) shows how the two regions (***`r133inf`*** and* **`r132inf`***) are garbage collected. Region* **`r119inf`** *and* **`r118inf`** *store the tree temporarily.*

The tree generated therefore occupies: $(2^h) = 8192$ words for the leafs and $(2^h - 1)6 = 49146$ words for the nodes, where $h = 13$ (a triple occupies four words). This gives a total of 229352 bytes for the tree. All nodes in the tree is replaced each time the tree is computed, which adds up to 196584 bytes. The first graph therefore occupies $229352 + (150 * 196584) = 29716952$ bytes in memory. The second graph will at maximum have the original graph plus five computed graphs alive at the same time. That sums up to $229352 + (5 * 196584) = 1212272$ bytes.

6.12 What to do with all these hints

We have presented several hints on programming with regions in the ML Kit, but it is not trivial to see how the hints can be combined, and what to do in practical situations where a space leak is located. The situation is even more complicated by the fact that several analyses is used on the ML source program during compilation.

It is possible that an optimization can have an positive effect on one analysis and an negative effect on another analysis. For example, turning an exomorphic function into an endomorph function restricts the posibility of region polymorphism in region inference but on the other hand gives the storage mode analysis a change to store arguments atbot for tail recursive functions.

However, because the region inference algorithm does have a formal specification in the form of region inference rules, it is possible to verify and understand what goes on. This also hold for the region representation analyses.

One also have to notice that the hints given in the previous sections are not exhaustive and other tricks exists (inspiration can be found in [4, 23, 20]).

Given an ML source program, we first profile the target program and locate possible space leaks. Then starting from the program points in the region annotated lambda program we see whether it is possible to use some of the hints outlined above, and of course experience with region inference is essential for getting an idea on how to remove the space leaks. To start out with it is an good idea to look out for:

exceptions. They are always put in global regions. Removing exception constructs from loops and using nullary exceptions instead of unary can be very effective.

references. The example with references in section 6.4 indicates that references in some situations can be very space consuming. Especially if the value which the reference is updated with (by the assign operation, `:=`) is stored attop.

endomorphic functions. Endomorphic functions can always be exomorphic by using the single copy trick.

curried functions. It can be advantageous to un-curry curried functions in connection with the storage mode analysis.

Every time one uses the double copy trick we recommend that you check that it actually works. If the region containing the data are not reset, then all copy operations will allocate yet more memory for the regions and the optimization turns out to be disastrous. We recommend that you always use the *resetRegions* primitive between the two copy operations. Remember, that it is always safe to use the *resetRegions* primitive.

We think it is best to use as much region polymorphism as possible because it is only for tail recursive functions that less region polymorphism can be advantageous. All memory allocated in exomorphic tail recursive functions are deallocated when the function returns from the calls. If arguments in endomorphic tail recursive functions are not stored atbot, then they are allocated as long as the "final" result is alive.

Our advise is to use as much region polymorphism as possible. Afterwards you can look out for tail recursive functions and make them endomorphic. Each time you make a tail recursive function endomorph, then check that you get the expected effect by profiling the program again. If not then make the function exomorphic again.

7 Implementation

In this section we discuss the main design decisions taken when implementing the profiler. It has been possible to extend the ML Kit with profiling without getting into conflicts with decisions already made in the ML Kit.

7.1 Two region stacks

We will discuss what happens on the machine stack when the target program is executed. The machine stack can be split in four parts.

1. Live variables pushed at function calls.
2. Previous exception pointers and exception handler closures shown on figure 3 (page 11).
3. Finite regions.
4. Infinite region descriptors.

If profiling is turned off, then allocating finite regions only sets space aside to a value (which will be stored later) by moving the stack pointer. It is not necessary to link the finite regions together. When deallocating a finite region the stack pointer will be above the region and we only have to adjust the stack pointer.

Infinite regions, however, need a *region descriptor* on the stack even though profiling is turned off ([10]). The region descriptor holds four pointers.

1. Pointer to the first *region page*.
2. Pointer to the next sequence of words to allocate.
3. Pointer to the end of the current region page. If an object cannot be allocated in the current page, then a new page will be allocated.
4. Pointer to the previous region on the stack. When raising an exception, then all regions above the address of the exception handler on the stack have to be deallocated. The previous link is then used to find the regions to deallocate.

On figure 21 we see a snapshot of a stack when profiling is turned off.

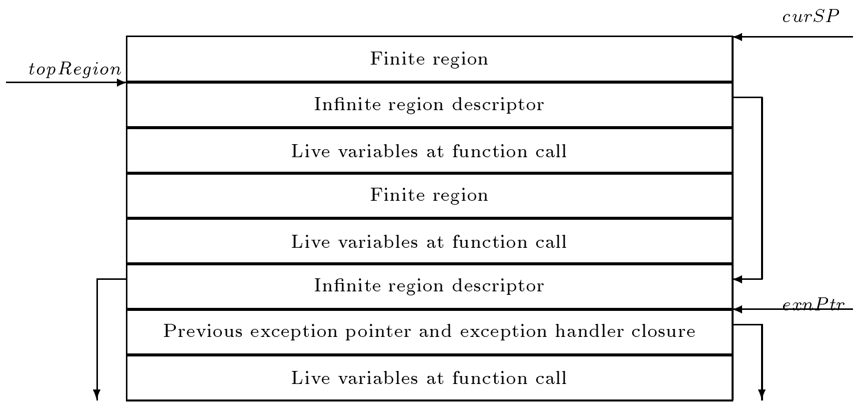


Figure 21: *This example stack contains one exception handler closure, two finite regions, two infinite region descriptors and three areas with live variables. The target program has been compiled with profiling turned off.*

We have to know the size of each object allocated when profiling, or we will not be able to traverse memory and count the number of allocated words. All allocation points in the region annotated lambda program are annotated with program points. Values allocated at runtime can now be identified by tagging all values with the program point. Each value is therefore tagged with an *object descriptor*.

Object Descriptor		
Program Point	Size Of Object	Object Data ···

We need extra information on both finite and infinite regions when profiling is turned on. At each profile tick the target program will traverse the two region stacks and inspect each region. Infinite and finite regions are traversed separately because they are implemented differently. Each region has a *region identifier* (a number) given at compile time which can be found as a letregion bound region variable in the region annotated lambda program. A finite region is extended with a *finite region descriptor* holding a previous region pointer and a region identifier.

		Object in Finite Region		
Previous Region Pointer	Region Identifier	Program Point	Size Of Object	Object Data ···
Finite Region descriptor		Object Descriptor		

Finite regions have an overhead of four words.

Infinite region descriptors are also extended with a region identifier. The infinite region statistics shown on page 19 is found by maintaining two counters in the region descriptor. The first counter holds the number of words used on objects in the region. The second counter holds the number of words used on object descriptors. The region descriptor now holds seven words (three extra words).

Previous Region Pointer	*allocPtr*	*borderPtr*	*firstPagePtr*	*allocNew*	*allocProfNow*	Region Identifier
Infinite Region descriptor						

The following drawing (figure 22) shows an example machine stack with the two region stacks, and a single region page in one of the infinite regions.

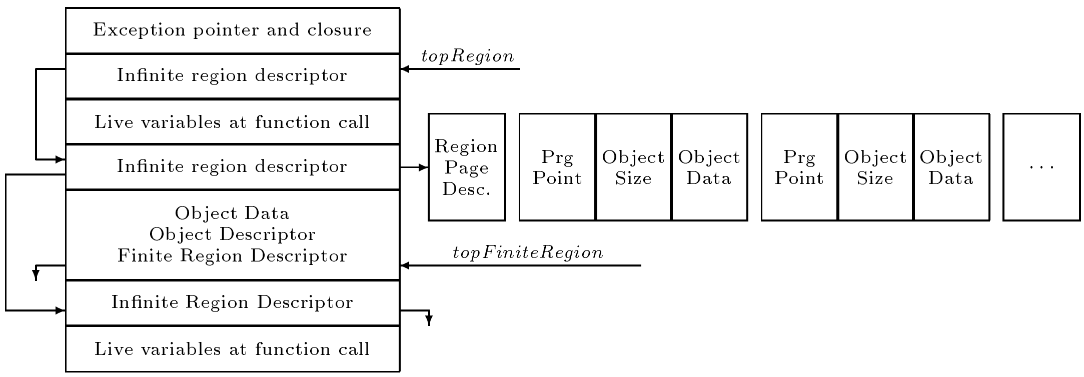


Figure 22: *The figure shows a snapshot of the machine stack when profiling is turned on. Variable topFiniteRegion points at the topmost finite region descriptor, and likely for topRegion. It is shown how objects are put in both finite and infinite regions.*

The function running through a region page, has to know when there is no more objects allocated. This is done by zeroing all words in a region page before it is used, and then if an object descriptor with zeroes is found we know that the rest of the region page is waste. This works by having all legal program points larger than zero.

7.2 Objects in Finite Regions

Space for objects in finite regions are allocated with the regions (by the letregion construct), but the program points (allocation points) are first known afterwards when the actual object is put into the allocated region. The size of the object allocated is known, so the size field of the object descriptor is updated. The program point is not known, so we use program point one indicating, that this is an object with no value stored yet. The first program point generated by the ML Kit is two.

When storing an object into a finite region, function `void storePrgPointProfiling(int pPoint, int *objPtr)` is used to update the program point.

7.3 Doubles

Doubles allocated in finite regions on the machine stack are aligned at runtime. This makes it necessary to use a dummy field `dummySizeReal = 42424242` as alignment field. The alignment field is recognised by the function `storePrgPointProfiling(pPoint, objPtr)` to locate the object descriptor when storing the program point for that object. If alignment is necessary, then the stack is as follows:

					Object data			
prevPtr	*regionId*	*prgPoint*	*size*	*dummySizeReal*	*doubleTag*	*doubleWord1*	*doubleWord2*	*oldSP*
Finite region desc.		Object desc.						

The object pointer points at the *doubleTag* value. If the stack is already double aligned the pointer to the double word is just above the object descriptor, and hence, when updating the program point, there has to be a unique difference between the size of any finite region object and `dummySizeReal`. The value for `dummySizeReal` is therefore chosen larger than the size of any other object ever allocated in a finite region.

				Object data.			
prevPtr	*regionId*	*prgPoint*	*size*	*doubleTag*	*doubleWord1*	*doubleWord2*	*oldSP*
Finite region desc.		Object desc.					

7.4 Interruption

When the target program is interrupted and a profile tick starts, then the program has to be in a safe state (i.e. all object and region descriptors have to be updated). If we allow a profile tick to occur at every state in the program then we have to update region and object descriptors in critical regions where no interruption is possible. This is usually controlled by semaphores. Object and region descriptors are updated so often, that the overhead introduced by semaphores will be significant, especially in execution time.

We therefore decided to only allow profile ticks at entrance to functions. It is easy to insert a *start profile tick* test at entrance to a function and any recursive (iterative) nature of the target program will be captured. We do not allow profile ticks at exit from functions. This only affects profiling of deep recursive functions (see figure 9 on page 22), but the memory usage is still clear from the profile graphs.

Each time a timer interrupt occurs, a global flag *timeToProfile* is set and normal program execution continues. This flag is then tested at entrance to a function and if set then a procedure *profileTick* is called which performs the profiling.

7.5 Module Profiling in The Runtime System

Code due to profiling in the runtime system is in module `Profiling`. The module exports functions controlling the timer and profile ticks.

`void profileTick(int *stackTop);` runs through the region stacks and collect data about the memory use. It is called with the address of the current stack pointer, so it can calculate the current stack size and thereby the number of words used on the stack for live variables and exception handlers.

`void profiling_on(void);` starts the timer. The timer is controlled by the command line options given at runtime.

`void profiling_off(void);` turns the timer off. Function `profiling_on` and `profiling_off` is all you need to control profiling from inside the ML source program.

`void resetProfiler();` resets the profiler which includes resetting the timer.

7.6 The Graph Generator

The graph generator is split in several modules. The module structure can be found in `Rp2Ps.h`. The layout of the profile data file is found in module `ProfileData`. Modules `Graph`, `Sample`, `Output`, `Curves` and `PostScript` contains code from the Haskell profiler ([17]), which has been modified to our needs. The internal data structures used are rather complicated due to efficiency (for example, a hash function is used to create the data structure). Module `Sample` contains a description of the data structure. Functions used to draw the graph is in `Graph`, `Output` and `Curves`. The postscript formatter is found in `PostScript`. The graph generator is compiled using the Makefile found in the Rp2Ps directory under the runtime system. This text is based on version 6.

8 Final remarks

We will now sum up on the problems outlined in the introduction and comment on our experience with the region profiler up til now.

The region graphs have been very good to show the size of regions and how the size evolved during execution of the target program. We will not conclude that in general only a small number of regions tend to be large; our example programs have not been general (or large) enough to justify such a statement. However, all our examples show only a small number of large regions.

The profiler does not directly give information about the average number of allocations into regions, but the region annotated lambda programs investigated until now have given the feeling, that the average live range is small.

A few programs (the factorial function for example, page 9) has shown that it is possible to allocate regions which are never used; it does not happen often, however.

The stack graph gives a good impression of what happens on the machine stack. This has been useful when investigating non tail and tail recursive functions.

The figure "Memory utilisation for infinite regions ..." (page 19) are very high for all our examples. We think that our page size of 800 bytes is a reasonable compromise. Too small page sizes gives an administrative overhead and too large page sizes makes the memory utilisation small.

We have shown that the storage mode analysis can save a lot of memory on tail recursive functions. If a function, supposed to be tail recursive, is not tail recursive, then the region flow graph and region annotated program are effective tools for locating and solving the problem.

It has been very important, that we could use some of the visualization facilities found in the Heap Profiler developed by Colin Runciman and David Wakeling. Especially the layout and postscript functions found in the package have been used (almost) without change.

It has been easy to use the *VCG* tool. The installation went without major problems on the HP-UX platform. The syntax used on the exported graph files is easy to use. We have only found a few flaws in the tool:

- Not all *layout algorithms* are implemented in version 1.30.
- Edges can only belong to one edge class. The ML Kit makes several copies of the same edge because it belongs to more than one class. All path edges belongs to at least two edge classes.
- It is not possible (from inside the `xvcg` window) to hide nodes in a graph which have neither in edges nor out edges. Usually there are many letregion bound region variables which are not used in calls to region polymorphic functions and they will neither have in nor out edges.

In general it seems that regions have small live ranges and memory is reused fairly eagerly. Some programs however allocates many values in global regions. Here the profiler has been effective in finding space leaks, and to give enough information to remove them.

Region Inference in the ML Kit does not handle some constructs (for example exceptions) very well. Space leaks produced by exceptions can be reduced by using nullary exception constructors or only applying one word values to unary exception constructors. It is also advantageous to not introduce exception constructors (or values) in loops. Maybe it is possible to shorten live ranges for exceptions, but it is not clear how well that can be done in combination with region inference.

It is very interesting to see how garbage collection can be combined with region inference. If only global regions are garbage collected, then it is likely that the garbage collection phase will be fast. If the garbage collector is optional, then it is possible to use a garbage collector on programs that are not tuned for the ML Kit and still get space efficient target programs. Programs tuned for the ML Kit can be executed without the garbage collector.

The user cannot in our profiler specify the "part" of a program ([19, *cost-centre*]) to which allocations should be attributed. It is not clear at this time to see if that would help the profiling task significantly because the region and program points (combined with the region annotated lambda program and region flow graphs) actually does that very well.

Experience with the ML Kit including the profiler is also reported in [4].

8.1 Acknowledgements

I will thank Martin Elsman, Tommy Højfeld Olesen, Martin Koch for valuable discussions and Mads Tofte for his patience during this project. Also, this project could not have been done without previous work on the ML Kit by Nick Rothwell, Lars Birkedal, David N. Turner, Martin Elsman and Mads Tofte. The experience reported by Peter Bertelsen and Peter Sestoft has also been very valuable.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison Wesley, 1986.

[2] Manuel Fähndrich Alexander Aiken and Raph Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Proc. of the ACM SIGPLAN '95 Conference on Programming Languages and Implementation (PLDI)*, pages 174–185, La Jolla, CA, June 1995. ACM Press.

[3] Sara Baase. *Computer Algorithms, Introduction to Design and Analysis.* Addison Wesley, second edition, 1988.

[4] Peter Bertelsen and Peter Sestoft. Experience with the ml kit and region inference. Department of Mathematics and Physics Royal Veterinary and Agricultural University, Copenhagen, Draft 1 of December 13 1995.

[5] Lars Birkedal. The ML Kit compiler – working note. Technical report, DIKU, Department of Computer Science, University of Copenhagen, July 1994.

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

[7] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, pages 171–183, St. Petersburg, Florida, January 1996. ACM Press.

[8] Martin Elsman. Note on Simple Incremental Compilation in the Region Based ML Kit Compiler. Department of Computer Science (DIKU), University of Copenhagen, September 1995.

[9] Martin Elsman and Niels Hallenberg. Note on Interfacing with C in the ML Kit with Regions. DIKU, Department of Computer Science, University of Copenhagen, December 1995.

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

[11] Robin Milner and Mads Tofte. *Commentary on Standard ML.* MIT Press, 1991.

[12] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[13] Hewlett Packard. *PA-RISC Assembly Language Reference Manual.* Hewlett Packard, fourth edition edition, January 1991.

[14] Hewlett Packard. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual.* Hewlett Packard, third edition edition, February 1994.

[15] Niklas Röjemo. Highlights from nhc - a space-efficient Haskell compiler. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 282–292, La Jolla, California, 1995. ACM SIGPLAN/SIGARCH and IFIP WG2.8, ACM Press.

[16] Colin Runciman and Niklaus Röjemo. New dimensions in heap profiling. Technical Report YCS-95-256, Deparment of Computer Sicence, Univeristy of York, England, Y01 5DD, 1995.

[17] Colin Runciman and David Wakelin. Heap profiling of lazy functional programs. *Journal of Functional Programming*, 3(2):217–245, April 1993.

[18] Georg Sander. *VCG, Visualization of Compiler Graphs, User Documentation V.1.30.* Universität des Saarlandes, 66041 Saarbrücken, Germany, February 1995.

[19] Patrick M. Sansom and Simon L. Peyton Jones. Time and space profiling for non-strict, higher-order functional languages. In *ACM Symposium on Principles of Programming Languages*, 1995.

[20] Mads Tofte. Resetting Regions, January 1996.

[21] Mads Tofte and Jean-Pierre Talpin. A theory of stack allocation in polymorphically typed languages. Technical Report DIKU-report 93/15, Department of Computer Science, University of Copenhagen, 1993.

[22] 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.

[23] Mads Tofte and Jean-Pierre Talpin. Region-based memory management for the typed call-by-value lambda calculus. Submitted for publication, April 1995.

[24] Mads Tofte (updated by Martin Elsman). *The* `interact` *Function*, December 15 1995.

[25] Magnus Vejlstrup. Multiplicity inference. Master's thesis, Department of Computer Science, University of Copenhagen, September 1994. report 94-9-1.