A Brief Introduction to Regions

Mads Tofte*
UC Berkeley
Computer Science Division

Abstract

The paper gives an overview of region-based memory management. The emphasis of the paper is on the dynamic aspects of execution, specifically memory management. We illustrate how three static program analyses can be used for inferring memory management directives. They are *region inference*, *physical size inference*, and *storage mode analysis.* We describe these analyses for a skeletal language inspired by Standard ML. We also describe a region-based runtime system for the skeletal language, based on the runtime system of the ML Kit with Regions.

1 Introduction

Memory management is an essential aspect of computer programming. Simply put, the problem is that computers have finite memory and that some re-cycling of memory is necessary. A program that consistently allocates more memory than it releases will eventually use up all the memory of the computer. A program that de-allocates memory too early (i.e., when the memory contains values that are actually required by the remainder of the computation) may crash or give wrong results.

Most programming languages provide the programmer with a memory management discipline, which helps the user manage memory, in return for imposing restrictions on how memory may be used.

Indeed, memory management considerations have often played a central role in the design of programming languages. One famous example is the programming language Algol 60[Nau63], which introduced what we will call the *pure* stack discipline. In the pure stack discipline the runtime stack can hold all values produced by the computation, including temporary variables, non-local variables, return addresses, and even certain variable-sized arrays. (The term *pure* refers to the idea that *all* values can be stored on the stack.) The beauty of the pure stack discipline is that every point of allocation is paired with a point of de-allocation and these points are easily identified from the program text.

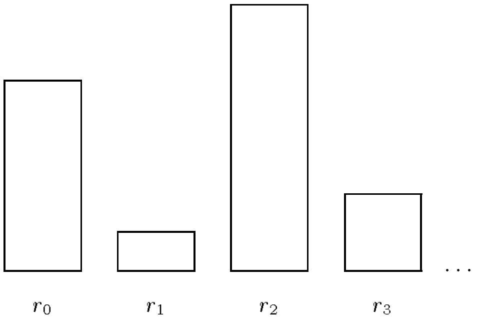

Figure 1: The store is a stack of regions; a region is a box in the picture. Four region names are shown: r_0, r_1, r_2, and r_3.

This gives predictability in programming and, when properly used, very space economical memory use.

The pure stack discipline has severe limitations, however. One limitation concerns physical sizes of values. The physical size of a value must be known at the latest when memory allocation for the value takes place at runtime. This rules out lists and other recursive datatypes, which are normally constructed incrementally. Also, the use of the stack for holding the environment rules out useful programming techniques, especially returning functions from function calls (under call-by-value). In particular, if we regard an object in object-oriented programming as containing functions (methods), then methods cannot in general return objects as results under the pure stack discipline.

Many programming languages get around these limitations by using a less pure stack discipline, where the values that cannot be put on the stack are stored in a *heap*. Recycling of memory is then done either explicitly by the programmer or automatically, by the *garbage collector*, which is a separate routine in the runtime system. Explicit management of memory is notoriously difficult and time consuming for the programmer. Garbage collection has been developed over a period of some forty years (see Wilson[Wil92] for an excellent overview). Thanks to this work, garbage collection has matured to be a practical form of memory management.

However, garbage collection does not offer the same level of predictability as the stack discipline. The separation of

[*] On sabbatical leave from University of Copenhagen

allocation and de-allocation makes it difficult to know how much memory a program uses. Typically, programmers resort to simply running the program on different input values to see what happens. It is difficult for a programmer who has no way of asserting or checking lifetimes of values to avoid that the garbage collector hangs on to values which are actually not needed.

Region-based memory management [TT94, BTV96, TT97] offers an alternative to the above memory management disciplines. At runtime, all values are stored in a so-called *stack of regions*, see Figure 1. All values that do not fit in one machine word are stored in regions. Such values include function closures and values of recursive types, such as lists and trees. Every region can grow dynamically and there is no fixed bound on the number of regions that can exist at runtime.

The size of a region can sometimes, but not always, be inferred at compile time.

The allocation and de-allocation of regions is determined at compile time, by a type-based analysis called *region inference*.

Region inference annotates every value-creating expression of the source program with an annotation of the form

$$\texttt{at}\ \rho$$

where ρ is a *region variable*. Note the distinction between *region* and *region variable*. A region is an area of memory at runtime. A region variable is a syntactic object in the program, so a program contains only finitely many region variables. At runtime, an environment maps region variables to so-called *region names*, where a regio name uniquely identifies a region.

Region inference also introduces bindings of region variables. One form of binding is the expression

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

which binds ρ in the expression e. At runtime, first a region is allocated at the top of the region stack, then e is evaluated (presumably storing and fetching values from the region) and finally, when `end` is reached, the region is popped off the stack. The expression e may itself contain further `letregion` expressions or call functions that contain `letregion` expressions. However, the `letregion` construct is the *only* primitive for allocating and de-allocating regions, from which it follows that regions obey a stack discipline.

The second form of binding of region variables takes the form

$$\texttt{letrec}\ f\texttt{[}\rho_1\texttt{,}\ldots\texttt{,}\rho_k\texttt{](}x\texttt{)}\ \texttt{at}\ \rho\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end} \qquad (1)$$

Here f is a (possibly recursive) function with formal parameter x, body e_1 and scope e_1 and e_2. The region variables $\rho_1, \ldots, \rho_k$ are called the *formal region parameters* of f. Intuitively, they indicate where f should place the values it produces — so e_1 may contain annotations of the form at ρ_i, $i \in \{1, \ldots, k\}$, among others. The scope of the binding $\rho_1, \ldots, \rho_k$ is e_1. The region variable ρ indicates where the closure for f is to be placed, in case f has free variables.

A function declared with the above `letrec` construct is said to be *region-polymorphic*. Every non-binding occurrence of f takes the form

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

where $\rho'_1, \ldots, \rho'_k$ are *actual region parameters*. The notation $\texttt{[}\rho'_1\texttt{,}\ldots\texttt{,}\rho'_k\texttt{]}$ at ρ' indicates that a record for holding the region parameters is created and stored in the region denoted by ρ'.

Different non-binding occurrences of f may use different actual region parameters. Indeed, region inference allows *polymorphic recursion* in regions, that is, referring to (1), inside e_1, f may be applied to actual region parameters which are different from the formal region parameters $\rho_1, \ldots, \rho_k$.

Region-based memory management has been implemented in *The ML Kit with Regions*[TBE⁺97], a compiler for Standard ML which uses regions as its sole form of memory management. The work described in this paper was mostly done at DIKU with the ML Kit group, and contains contributions due to Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, Magnus Vejlstrup, and Jean-Pierre Talpin in addition to those of the author.

2 Source Language

In this section we define the source language and present a sample program, which we use as a running example.

The syntactic categories of the source language are *integer constants*, ranged over by i, *constants*, ranged over by c, *variables*, ranged over by f and x, *binary operators*, ranged over by *bop*, *expressions*, ranged over by e, and *patterns*, ranged over by *pat*.

The grammar for the language is:

$$\begin{array}{rcl}
c & ::= & \texttt{nil} \mid \texttt{true} \mid \texttt{false} \mid i \\
bop & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{=} \mid \texttt{>} \mid \ldots \\
e & ::= & c \mid x \mid e\ bop\ e \\
 & \mid & \texttt{let}\ pat\texttt{=}e\ \texttt{in}\ e\ \texttt{end} \\
 & \mid & \texttt{(}e\texttt{,}e\texttt{)} \mid e\texttt{::}e \mid \lambda x.e \\
 & \mid & \texttt{case}\ e\ \texttt{of}\ pat\texttt{=>}e\ \mid\ pat\texttt{=>}e \\
 & \mid & \texttt{letrec}\ f\texttt{(}x\texttt{)=}e\ \texttt{in}\ e\ \texttt{end} \\
pat & ::= & c \mid x \mid \texttt{(}x\texttt{,}x\texttt{)} \mid x\texttt{::}x
\end{array}$$

Although this language may appear to be a high-level language, it is in fact designed with memory management in mind. Every constant is assumed to fit into one memory word or one register. (We do not treat constants of different sizes in this paper.) Every binary operators is assumed to correspond to a machine operation which can be carried out using registers alone, without allocation of memory.

The construct let pat=e_1 in e_2 end binds the variables that occur in *pat* with scope e_2. It does not cause allocation of memory (beyond allocation done by e_1 and e_2, of course).

The three constructs in the next line all allocate memory. The expression (e_1,e_2) evaluates e_1 and e_2 to values v_1 and v_2, respectively, and then stores the pair (v_1, v_2) in memory.

For brevity, we write `fst` e for

$$\texttt{let (x,y) =}\ e\ \texttt{in x end}$$

and `snd` e for

$$\texttt{let (x,y) =}\ e\ \texttt{in y end}$$

The expression e_1::e_2 evaluates e_1 and e_2 to values v_1 and v_2, respectively, and then stores a list cell with head v_1 and tail v_2 in memory. In fact, a list cell is also just a pair (v_1, v_2); the reason we treat lists and pairs differently in the source language is that region inference for pairs is different for region inference for lists.

The expression $\lambda x.e$ creates a *function closure* in memory. The closure is a triple $\langle x, e, E\rangle$, where E is an environment mapping the free variables of $\lambda x.e$ to values. Like Standard ML, our language uses static scoping.

```
letrec mk(n) = if n=0 then nil else n::mk(n-1)
in letrec app(p)=
     let (xs,ys) = p
     in case xs of
         nil => ys
       |  x::xs' => x::app(xs',ys)
     end
   in letrec len(q)=
        let (l,acc)=q
        in case l of
            nil=>q
          | y::zs => len(zs,acc+1)
        end
      in
        snd(len(app(mk 5000, mk 5000),0))
      end
   end
end
```

Figure 2: A source program.

The expression `case` e `of` pat`=>`e `|` pat`=>`e evaluated e to a value and then branches to the first or the second branch. For brevity, we write

$$\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3$$

for

$$\texttt{case } e_1 \texttt{ of true => } e_2 \texttt{ | false => } e_3$$

In `letrec` $f(x)$`=`e_1 `in` e_2 `end`, f is the name of a (possibly recursive) function with formal parameter x and body e_1. The scope of the declaration is e_1 and e_2.

Figure 2 shows a sample source program, which we shall use as a running example. `mk`(n) creates a list of length n; `app`$(l_1,\ l_2)$ appends lists l_1 and l_2; `len`(l,n) computes the length of the list l using n as an accumulating parameter. The "main" program creates two lists, appends them, and computes the length of the resulting list.

3 Region-Annotated Terms

The result of region inference is a region-annotated term. We add two new syntactic categories, *allocation points*, ranged over by a, and *binding points*, ranged over by b. We assume a denumerably infinite set RegVar of *region variables*; we use ρ to range over region variables. The grammar for region-annotated terms is:

$$\begin{array}{rcl}
a & ::= & \texttt{at } \rho \\
b & ::= & \rho \\
c & ::= & \texttt{nil} \mid \texttt{true} \mid \texttt{false} \mid i \\
bop & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{=} \mid \texttt{>} \mid \ldots \\
e & ::= & c \mid x \mid e\ bop\, e \\
 & \mid & \texttt{let } pat\texttt{=}e \texttt{ in } e \texttt{ end} \\
 & \mid & (e\texttt{,}e)a \mid (e\texttt{::}e)a \mid (\lambda x.e)a \\
 & \mid & \texttt{case } e \texttt{ of } pat\texttt{=>}e \mid pat\texttt{=>}e \\
 & \mid & \texttt{letrec } f\texttt{[}b_1\texttt{,}\ldots\texttt{,}b_k\texttt{]}(x)a\texttt{=}e \texttt{ in } e \texttt{ end} \\
 & \mid & f\texttt{[}a_1\texttt{,}\ldots\texttt{,}a_k\texttt{]}a_0 \\
 & \mid & \texttt{letregion } b_1\texttt{,}\ldots\texttt{,}b_k \texttt{ in } e \texttt{ end} \\
pat & ::= & c \mid x \mid (x\texttt{,}x) \mid x\texttt{::}x
\end{array}$$

Creating a pair, a list cell or a closure requires allocation of memory.[1] Hence the allocation points on $(e\texttt{,}e)a$, $(e\texttt{::}e)a$, and $(\lambda x.e)a$.

Next, consider an expression of the form

$$\texttt{letrec } f\texttt{[}\rho_1\texttt{,}\ldots\texttt{,}\rho_k\texttt{]}(x)a\texttt{=}e \texttt{ in } e \texttt{ end}$$

If f has free variables, a closure for f has to be stored; the allocation point a indicates where to put this closure. (We omit the allocation point when f has no free variables.) Next, ρ_1, ..., ρ_k are the *formal region parameters* of f; they indicate where f should put the values it creates.

An expression of the form

$$f\texttt{[at } \rho'_1\texttt{,}\ldots\texttt{, at } \rho'_k\texttt{]}a_0$$

applies f to *actual region parameters* ρ'_1, ..., ρ'_k. Different applications of the same f may use different actual region parameters. We say that f is *region-polymorphic.* The actual region parameters are passed to f in a record of size k words which is stored in the region indicated by a_0. For brevity, we write $f\texttt{[}\rho'_1\texttt{,}\ldots\texttt{,}\rho'_k\texttt{]}a_0$ for

$$f\texttt{[at } \rho'_1\texttt{,}\ldots\texttt{, at } \rho'_k\texttt{]}a_0$$

Furthermore, we write $\langle f[a_1,\ldots a_k]e\rangle$ as a shorthand for

$$\texttt{letregion } \rho \texttt{ in } f\texttt{[}a_1,\ldots,a_k\texttt{] at } \rho\ e \texttt{ end}$$

where ρ can be any region variable which occurs free neither in $a_1,\ldots,a_k$ nor in e.

Region inference transforms source programs to region-annotated terms. The region-annotated term is identical to the source program, except that the former contains allocation points and binding points. In other words, all the evaluation steps are precisely as in the source program, but memory allocation and de-allocation have been made explicit.

Figure 3 shows a region-annotated version of the source program from Figure 2. We now explain the main features of region-annotation by way of the example. Starting from the bottom, notice that the two applications of `mk` use different region parameters: the first uses ρ_9 and the second uses ρ_{10}. This is an example of region polymorphism. In this case the two calls of `mk` create two lists in different regions.

Region inference requires that all elements of a list be in the same region. Thus all the elements of the first list are in ρ_9 and all the elements of the second list are in ρ_{10}. However, different lists may reside in different regions.

The `app` function repeatedly conses the elements of the first list onto the second list. Thus the actual region argument to `app` is the region of the second list, ρ_{10}.

After the application of `app`, the first list and the pair of the two lists are garbage! Thus region inference has introduced a `letregion` on ρ_9 and ρ_{11} around the application.

How can region inference deduce that the first list and the pair are garbage? Well, it infers two pieces of static information for the application, namely

1. a *region-annotated type*, in this case `(int list,` ρ_{10}`)`; and

2. an *effect*, in this case set $\{\rho_9, \rho_{10}, \rho_{11}\}$.

[1] For simplicity, we allocate all pairs and closures although in practice, this is not necessary.

```
letrec mk[ρ2](n) =
  if n=0 then nil else (n::⟨mk[ρ2](n-1)⟩)atρ2
in letrec app[ρ4](p) =
     let (xs,ys) = p
     in case xs of
         nil => ys
       |  x::xs' =>
          (x::letregion ρ6
              in ⟨app[ρ4](xs',ys)atρ6⟩
              end)atρ4
     end
   in letrec len[ρ8](q)=
        let (l,acc)=q
        in case l of
            nil=>q
          | y::zs => ⟨len[ρ8](zs,acc+1)atρ8⟩
        end
      in
        letregion ρ10, ρ12
        in
          snd(⟨len[ρ12]
            letregion ρ9, ρ11
            in (⟨app[ρ10](⟨mk[ρ9] 5000⟩,
                         ⟨mk[ρ10] 5000⟩)atρ11
            end, 0)atρ12⟩)
        end
      end
   end
end
```

Figure 3: When applied to the source program in Figure 2, region inference produces the above region-annotated term.

In general, the region-annotated type of an expression is very much like the ML type of an expression, except that every type has been annotated with a region variable (indicating where the value lies).

The effect of an expression is a finite set of region variables.[2] It is an upper bound on the set of regions required by the evaluation of the expression. There are two ways in which an expression can "require" a region: either for accessing a value stored in the region or for writing a value into the region. The region inference system actually distinguishes between the two kinds of uses and makes use of this distinction, for example, to reduce the number of region parameters to region-polymorphic functions: only regions that are used for storing values need be passed to a region-polymorphic function. Such region parameters are called *put parameters*; we show only put parameters in this paper.

In the example at hand, the application of app puts values into ρ_9, ρ_{10}, and ρ_{11}.

To introduce the letregion, region inference employs the following rule:

> *If a region variable occurs in the effect of an expression but does not occur in the type of the expression and does not occur free in the type environment, then that region variable denotes a region whose use is purely local to the computation and which may therefore be discharged using* letregion*.*

Here a *type environment* is a map of program variables to their region-annotated, region-polymorphic types. In the concrete example, mk and app both have closed type schemes (i.e., region-polymorphic types with no free type or region variables) so only the region-annotated type and effect of the expression itself matters. Since ρ_9 and ρ_{11} are in the effect of the expression but not in the type, they may be discharged by a letregion. Note that ρ_{10} cannot be discharged, since it occurs in the type of the expression.

Similarly, the region-annotated type of the argument to snd is $((\texttt{int list}, \rho_{10}) * \texttt{int}, \rho_{12})$. The type of the result of the projection is simply int, so ρ_{10} and ρ_{12} may be discharged.

Now let us turn our attention to the auxiliary functions mk, app, and len. In the body of mk, note that the recursive call of mk uses the formal region parameter of mk as actual region parameter. Hence all list elements are pumped into the same region, ρ_2. A similar phenomenon is observed where app calls itself, but notice that the region where the argument pair (xs',ys) is stored is popped after the recursive call. The reason is a feature called *polymorpic recursion.* If we did not only show put parameters, app would have one more region parameter, ρ_3, say, which indicates where the argument p lies. When app calls itself recursively, it may instantiate ρ_3 to a different region variable, in this case ρ_6. The result of the recursive call has type $(\texttt{int list}, \rho_4)$ and the recursive call has effect $\{\rho_4, \rho_6\}$, so ρ_6 may be discharged.

Polymorphic recursion works for put regions too: a function can call itself using regions which are local to its body. This resembles the well-known concept of activation record for recursive functions, where each recursive call has its own activation record. With regions, however, the data that is local to an invocation may include, for example, regions that contain lists or binary trees.

Finally consider len. The interesting thing about len is that it is tail recursive. The function is written with

[2] This is a simplification; for the full story, see [TT97].

care to force the argument pair of the recursive call (that is, (zs, acc+1) into the same region as the argument pair, q. At first, this may seem silly, since it would appear that these argument pairs will pile up in ρ_8. However, the storage mode analysis described in the next section will ensure that the same two words are used for storing all the different argument pairs: the region will be "reset" before each iteration.

The way we force the argument pairs into the same region is by returning q in the first branch of the case statement. The two branches must have the same result type. Therefore q and the result of the tail call of len must have the same type. In other words, len must return its result in the same region as it receives its argument.

Were we to return acc instead of q in the first branch, this constraint would disappear, and polymorphic recursion would place the pair (zs, acc+1) in a local region:

```
letregion ρ9 in ⟨len[](zs,acc+1)atρ9⟩ end
```

This would be unfortunate, since the function would no longer be tail recursive. Indeed, the argument pairs would pile up on the stack and upon each return, the function would have to de-allocate two words on the stack.

4 Adding Storage Modes and Physical Sizes

As the example in the previous section illustrated, one has to do something more than just region inference for tail recursive functions. On closer inspection, tail recursion is just a special case of a more fundamental problem with region inference, namely that there are many computations where lifetimes simply are not nested! In the case of tail recursion, one has a sequence of pairs whose lifetimes do not overlap. In such cases, the obvious solution is to reset the region before each new store, that is, prefix a store operation with operations which reset the allocation pointer of the region to the beginning of the region. The net effect is that the region will be updated destructively, using the region as a traditional updatable store.

To express this in our language, we change the definition of allocation points to

$$a ::= \texttt{attop}\ \rho \mid \texttt{atbot}\ \rho \mid \texttt{sat}\ \rho$$

Here attop ρ means "store the value at the top of the region, extending the size of the region", atbot ρ means "reset the region, then store the value", and sat ρ means "store the value somewhere in ρ". The last storage mode is used when ρ is a formal region parameter to a region-polymorphic function. In this case, the actual storage mode is passed to the function at runtime, and the code for sat ρ tests whether the actual runtime is attop or atbot and performs the store accordingly.

A second important optimization is to distinguish between finite and infinite regions. Finite regions are allocated on the stack, while infinite regions are allocated using a more expensive scheme, described in Section 5.

This optimization turns out to be surprisingly effective. In many programs, over 95% of the allocations are done on the stack, resulting in a three-fold speed-up [BTV96]. Indeed, with finite regions, the analogy with activation records becomes even more appealing: finite regions are stored as "temporary" data which is local to each function invocation. Even infinite regions fit the traditional stack-discipline, see Section 5.

```
letrec mk[ρ2 : ∞](n) =
  if n=0 then nil else (n::⟨mk[sat ρ2](n-1)⟩)attopρ2
in letrec app[ρ4 : ∞](p) =
     let (xs,ys) = p
     in case xs of
         nil => ys
       |  x::xs' =>
          (x::letregion ρ6 : 2
               in ⟨app[sat ρ4](xs',ys)atbotρ6⟩
               end)attopρ4
     end
   in letrec len[ρ8 : ∞](q)=
        let (l,acc)=q
        in case l of
            nil=>q
          | y::zs => ⟨len[sat ρ8](zs,acc+1)satρ8⟩
        end
      in
        letregion ρ10 : ∞,  ρ12 : ∞
        in
          snd(⟨len[atbot ρ12]
            letregion ρ9 : ∞,  ρ11 : 2
            in (⟨app[atbot ρ10](⟨mk[atbot ρ9] 5000⟩,
                                ⟨mk[atbot ρ10] 5000⟩
                                )atbot ρ11
            end, 0)atbot ρ12⟩)
        end
      end
   end
end
```

Figure 4: When applied to the program in Figure 3, storage mode analysis and physical size inference produce the above term.

To express the results of region size inference, we introduce the notion of a *region size*, s, which is either a non-negative integer, or infinity:

$$s ::= i \mid \infty$$

Here i is physical size (in 32 bit words) and ∞ denotes an "infinite" region (i.e., a region for which no finite bound is found at compile-time). We then replace our previous definition of binding points by:

$$b ::= \rho : s$$

that is, every binding occurrence of a region variable is annotated by a physical size. If the binding point is part of a formal parameter list of a region-polymorphic function

$$\texttt{letrec f}[\rho : s, \ldots](x)a\texttt{=}e_1 \texttt{ in } e_2 \texttt{ end}$$

then s indicates the size of memory which f allocates into ρ, including calls to other functions or to f itself. If the binding point is part of a letregion expression

$$\texttt{letregion } \rho : s, \ldots \texttt{ in } e \texttt{ end}$$

then s indicates the size of region ρ.

The grammar for expressions is the same as before, with the revised reading of a and b, of course.

Figure 4 shows the result of performing storage mode analysis and physical size inference on the program from Figure 3.

We explain the main features of these two analyses by way of the example. First consider the body of `app`. The region ρ_6 is finite: it is represented by 2 words on the stack. The pair is stored `atbot`, which for a stack-allocated region means the same as `attop`, namely store the value into the stack a the point pointed to by the region variable (ρ_6).

The binding point of ρ_4 says that `app` puts an unbounded number of values into ρ_4, which is correct, since the number depends on the length of the first argument to `app`. The recursive call `app[sat` ρ_4`](xs',ys)atbot`ρ_6 passes ρ_4 with storage mode `sat`, since `app` has not created a value in ρ_4 which needs to be kept alive across the application. The list `(x::` $\cdots$`)` is stored into ρ_4 `attop`, because the result of the recursive call must not be overwritten.

Proceeding to the declaration of `len`, we see that both storage modes for ρ_8 are `sat`. That is because the call is a tail call and therefore there is no value locally live within `len` to protect. Note that at the point where `len` is applied to the actual region ρ_{12}, the storage mode is `atbot`, (since the pair is not live in the main expression after the call), so `len` will use `atbot` in all its iterations.

All the storage modes in the main part of the program are `atbot`. One might wonder why ρ_{12} has physical size ∞, when in fact it will never contain more than two words. The reason is that the physical size inference is based on counting how many stores there are in the region; for simplicity, the size inference analysis never decreases the size it associates with a region (if it did, the analysis might become more precise, but termination of the analysis would become a concern). Since there are potentially infinitely many stores into ρ_{12}, the region is classified as infinite.

5 The Abstract Machine

We now present an abstract machine which is a suitable intermediate form for code generation. Indeed the machine we present here is an extract of the Kit Abstract Machine[BTV96], which the ML Kit uses as an intermediate language on its way to C or HP PA-RISC code.

The abstract machine has 32-bit registers, a *runtime stack*, and a *region heap*. The *region heap* consists of a set of fixed-size *region pages*. A finite region is represented simply as a finite number of 32-bit words on the stack. An infinite region is represented by a *region descriptor* on the stack, together with a linked list of region pages in the region heap. A region descriptor is a triple which consists of a pointer to the first region page, a pointer to the last region, and a number indicating the remaining space in the last region page of the region.

Finally, the machine has a *free list* of region pages. Whenever an infinite region needs to grow, the runtime system extends it with a page from the free list; de-allocation of an infinite region is fast: simply append the region to the free list using a few pointer operations. Allocating an infinite region is done by pushing a region descriptor onto the stack and letting it point to a single region page taken off the free list.

The machine reserves certain registers for dedicated use. They are:

- `sp`, the *stack pointer* register, which points to the topmost word of the stack;
- `fn`, the *function pointer* register, which points to the closure of the function whose body is currently being evaluated;
- `varg`, the *value argument* register, through which all function value arguments are passed.
- `rarg`, the *region argument* register, through which all actual region arguments are passed;
- `ret`, the *return* register, through which all return values from function calls are passed;

We use r to range over the registers.

In addition, we assume an infinite set of *temporary variables*. We use t to range over temporary variables. A temporary variable can hold any word-size object. By a *variable* we understand a register or a temporary variable. We use x to range over variables. Variables can be updated destructively.

A variable con contain either a constant, a region name, or a pointer to a stored value. A *region name* is a pointer to a place in the stack. If the region is finite, the region name points to the beginning of the region; if the region is infinite, the region name points to the region descriptor. In both cases, the two least significant bits of the region name are used for special purposes. One bit, the *attbot bit*, is set when one wants to represent the storage mode `atbot`. The other bit, the *infinity bit*, is set when the region is an infinite region. The concept of passing a region to a region-polymorphic functions is realised by setting or clearing the two bits in the region name of the region and then passing the region name. We use *rn* to range over region names.

Further, the machine has a set of code *labels*, ranged over by l and the same set of constants as the previous languages in this paper.

In examples we write concrete temporary variables and labels in `this font`.

The syntactic categories of the abstract machine language are: *variables*, ranged over by x, *effective addresses*, ranged over by *ea*, *boolean expressions*, ranged over by *boolexp*, *arithmetic operators*, ranged over by *aop*, *statements*, ranged over by *stmt*, *functions*, ranged over by *fun*, and *programs*, ranged over by p.

The grammar for the abstract machine language appears in Figure 5. Useful abbreviations are shown in Figure 6.

We now describe the operations of the machine. The boolean expressions include the usual comparisons, which are always on word-sized objects. In `infinite(`x`)`, x is supposed to hold a region name; the predicate is true iff the infinity bit of the region name is set. Similarly, `atbotbit(`x`)` tests the atbot bit of x.

The statement x`:=`*ea* stores the value of *ea* in x. In x_1`:=`x_2`[`i`]`, x_2 is supposed to hold a memory address, say a; the statement stores the contents of the word with address $a + i$ in x_1. In x_1`[`i`]:=`*ea*, x_1 is supposed to hold a memory address, say a; the statement stores the value of *ea* in memory at address $a + i$.

The statement x`:= pushregion()` pushes a region descriptor for an infinite region onto the stack, storing the address of the region descriptor in x. Allocation of a finite region is done using the x`:= allocs(`i`)` abbreviation in Figure 6.

In x_1`:= allocm(`x_2,i`)`, x_2 is supposed to hold the name of an infinite region, say *rn*. The statement allocates a record of i words of memory from the region whose name is *rn* and stores a pointer to this record in x_1.

The statement `popregion` pops the topmost region, which must be an infinite region, off the runtime stack. Popping a finite region off the runtime stack is done by the `pop(`i`)` abbreviation in Figure 6.

```
      x  ::=  r | t

     ea  ::=  c | l | x

boolexp  ::=  ea=ea | ea>ea | ...
          |   infinite(x) | atbotbit(x)

    aop  ::=  + | - | ...

   stmt  ::=  x:=ea | x:=x[i] | x[i]:=ea
          |   x:=ea aop ea
          |   if boolexp then stmt else stmt
          |   x:= pushregion()
          |   x:= allocm(x,i)
          |   popregion
          |   setbotbit(x) | clearbotbit(x)
          |   setinfbit(x) | clearinfbit(x)
          |   resetregion(x)
          |   skip | jmp(ea)
          |   l:stmt
          |   {⟨temp t1,...,tn;⟩ stmt}
          |   stmt;stmt

    fun  ::=  fun l is stmt

      p  ::=  fun ⟨p⟩
```

Figure 5: The grammar for the abstract machine

```
     push(ea)  ≡  {sp:=sp+1; sp[0]:=ea}
x:= allocs(i)  ≡  {x:= sp; sp:= sp+i}
  x:= popto()  ≡  {x:=sp[0]; sp:= sp-1}
       pop(i)  ≡  sp:= sp-i
       return  ≡  {temp t; t:=popto(); jmp(t)}
```

Figure 6: Abbreviations

```
fun mk is
 if varg=0 then nil
 else {
  {temp t1, t2, t3;
   t1:= varg-1;          n-1
   t2:= allocs(1);       region for [sat ρ0]
   t3:= rarg[0];         access ρ2;
   t2[0]:= t3;           store [sat ρ2]
   push(varg)            save before call
   push(rarg)            save before call
   push(lab1)            save return address
   varg:= t1;
   rarg:= t2;
   jmp mk};              non-tail call
  lab1:{temp t1, t2;
   pop(rarg);            restore
   pop(varg);            restore
   pop(1);               end of letregion
   t1:=rarg[0];          access ρ2
   t2:= allocm(t1,2);    allocate pair
   t2[0]:= varg;         store (n,
   t2[1]:= ret;             ⟨···⟩)
   ret:= t2;             store return value
   return}
 }
```

Figure 7: Machine code for the `mk` function of Figure 4.

In `setbotbit`(x), x is supposed to hold a region name; the statement destructively sets the atbot bit of x. Similarly for `clearbotbit`(x), `setinfbit`(x), and `clearinfbit`(x).

In `resetregion`(x), x is supposed to hold the name of an infinite region, say *rn*. The statement frees all region pages of the region, except the first, and records in the region descriptor that the page is empty.

The statement `skip` does nothing while `jmp`(*ea*) jumps to the value of *ea* which is supposed to be a label.

The statement *l*`:`*stmt* is a labelled statement, typically a point of return from a function call.

The statement $\{\langle$`temp` $t_1,\ldots,t_n;\rangle$ *stmt*$\}$ is called a *block*. Here t_1, ..., t_n are temporary variables with scope *stmt*. Angle brackets ($\langle\ \rangle$) enclose optional phrase parts. Thus a block can also simply be used for parenthesizing statements.

Finally, a program is a sequence of function declarations.

Figure 7 shows abstract machine code that implements the `mk` function of Figure 4. The code is as elegant as one could hope for, except for the extra stack operations involved in creating the (singleton) region vector `[sat` ρ_2`]`. First of all, boxing a record with one element is a bit excessive. But one can do better: when a region-polymorphic function f calls itself using precisely the formal region parameters of f as actual region parameters and if all the actual region parameters have storage mode `sat`, then the call can simply re-use `rarg` without any change! The ML Kit contains this optimization. This is seen in the code `len`, see Figure 8.

6 Region Profiling

The ML Kit contains a *region profiler*, which makes it possible profile runtime space usage.

Figure 9 shows a region profile for our example program. The profile has time along the x-axis and memory use along the y-axis. The different shades represent different regions;

```
fun len is
 {temp l, acc;
  l:= varg[0];          access l
  acc:=varg[1];         access acc
  if l=nil then {ret:= varg; return}
  else {temp y, zs, t1, rho8;
    y:= l[0];
    zs:= l[1];
    t1:= acc+1;
    rho8:= rarg[0];
    if atbotbit(rho8) then resetregion(rho8)
    else skip;
    t2:= allocm(rho8,2);
    t2[0]:= zs;     store (zs,
    t2[1]:= t1;           acc+1)
    varg:= t2;
    jmp len}        tail call
  }
```

Figure 8: Machine code for the len function of Figure 4.

there is a shade for each letregion-bound region variable in the program. These region variables are listed, each with their shade, on the right. (The numbers differ from the ones in the previous sections.)

Each mark on the x-axis represent a point in time where the profiler interrupted the program and traversed the memory. We see three hills, followed by a plateau. The first hill is from the call mk[atbot ρ_9] 5000. During the first climb (till approximately 0.08 seconds) the stack grows, together with r101561, which is finite region which surrounds the recursive call of mk. The period from 0.08 seconds to 0.18 seconds consists of stack pops combined with list cell constructions in regions r101656 and r101657. The version of the Kit used for these measurements represents lists using *two* regions: one for the spine of the list and one for the pairs to which :: is applied; thus the lists take up twice as much space: 5000 cells × 4 words/cell × 4 bytes/word ≃ 80Kb. The two regions r101656 and r101657 correspond to ρ_9. Note that they stay alive till the plateau is reached.

The second and third tops are similar. The plateau represents the time when len is working. As predicted, len runs in constant space.

7 Conclusion

Region inference and the other static analyses presented in this paper make it possible to use classical stack-based implementation techniques for languages which are beyond the scope of the pure stack discipline.

As in the pure stack discipline, allocation points and de-allocation points are paired, and they are determined at compile time. Note, however, that allocation and de-allocation are not determined by syntax, as is normal in block-structured languages, but by static analyses. The programming style one has to adopt to make best use of the region stack resembles programming styles known from block-structured languages, however.

A particularly pleasing aspect of the techniques presented in this paper is that the elementary memory management operations are spread out over the computation and moreover, they each take constant time! The absence of interruptions of unbounded duration is clearly attractive for real-time programming.

There is no claim that region-based memory management removes the need for traditional runtime garbage collection techniques. The claim is that, at least as ML-like languages are concerned, the need for reference tracing garbage collection can be greatly decreased and in many cases eliminated altogether.

8 Further Reading

Region inference builds on previous work on effect systems, see [TJ92, DKGS87, JG91].

The home page for the ML Kit project is:

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

From the web page, one can access the most recent version of the ML Kit, technical reports, and some of the published papers.

The region inference rules are described in [TT94, TT97], where it is also proved that the region inference rules are sound, that is, that they prevent de-allocation of data which is actually needed by the remainder of the computation. These papers also contain measurements of object programs produced by the ML Kit and comparisons with object programs produced by Standard ML of New Jersey.

For more information on storage mode analysis, physical size inference, and other analyses performed by the ML Kit, see [BTV96].

For an algorithm that performs region inference, see [TB98]. The article contains a proof that the algorithm is sound with respect to the region inference rules.

Elsman and Hallenberg [EH95] describe the backend of the KIT, including the Kit Abstract Machine, register allocation, and runtime system.

Elsman [Els98] describes an analysis which eliminates polymorphic equality in ML programs. This analysis, combined with region inference, makes tagging of allocated objects unnecessary.

For advice on programming with regions in Standard ML, consult [TBE^{+}97].

References

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

[DKGS87] J. M. Lucassen D. K. Gifford, P. Jouvelot and M.A. Sheldon. Fx-87 reference manual. Technical Report MIT/LCS/TR-407, MIT Laboratory for Computer Science, Sept 1987.

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

[Els98] Martin Elsman. Polymorphic equality - no tags required. In *Second International Workshop on Types in Compilation*, March 1998.

[JG91] P. Jouvelot and D.K. Gifford. Algebraic reconstruction of types and effects. In *Proceedings of the 18th ACM Symposium on Principles of Programming Languages (POPL)*, 1991.

[Nau63] Peter Naur. Revised report on the algorithmic language Algol 60. *Comm. ACM*, 1:1–17, 1963.

[TB98] Mads Tofte and Lars Birkedal. A region inference algorithm. Transactions on Programming Languages and Systems (TOPLAS), July 1998.

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

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

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

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

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In Y. Bekkers and J. Cohen, editors, *Memory Management, Proceedings, International Workshop IWMM92*, pages 1–42. Springer-Verlag, September 1992.

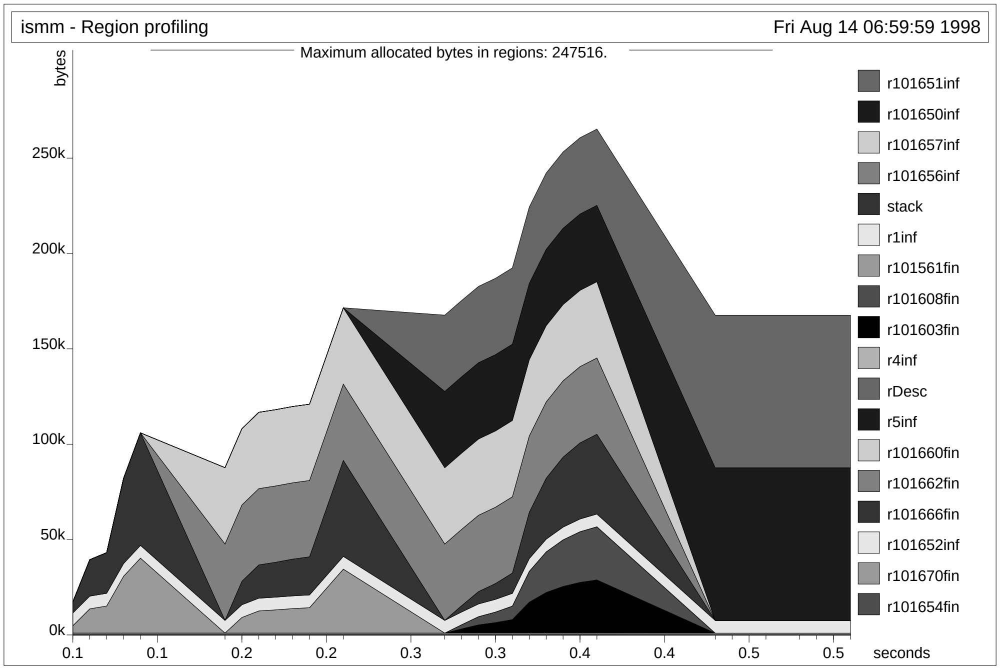

Figure 9: A region profile for the example program.