From Region Inference to von Neumann Machines via Region Representation Inference

Lars Birkedal, Carnegie Mellon University* Mads Tofte, University of Copenhagen†

Magnus Vejlstrup, NKT Elektronik‡

Abstract

Region Inference is a technique for implementing programming languages that are based on typed call-by-value lambda calculus, such as Standard ML. The mathematical runtime model of region inference uses a stack of regions, each of which can contain an unbounded number of values. This paper is concerned with mapping the mathematical model onto real machines. This is done by composing region inference with *Region Representation Inference*, which gradually refines region information till it is directly implementable on conventional von Neumann machines. The performance of a new region-based ML compiler is compared to the performance of Standard ML of New Jersey, a state-of-the-art ML compiler.

1 Introduction

It has been suggested that programming languages which are based on typed call-by-value lambda calculus can be implemented using regions for memory management[17]. At runtime, the store consists of a stack of regions. All values, including function closures, are put into regions. Region inference, a refinement of Milner's polymorphic type discipline, is used for inferring where regions can be allocated and where they can be deallocated. For each expression which directly produces a value (such as a constant, a tuple expression or a lambda abstraction), region inference also infers a region in which the value should be put. Experiments with a proto-type implementation of region inference and an instrumented interpreter have suggested that often it is possible to achieve very economical use of memory resources, even without garbage collection[17].

The potential benefits of region inference are:

* Work done while at University of Copenhagen. Current address: School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA; email: birkedal@cs.cmu.edu.

† Address: Department of Computer Science (DIKU), University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark; email: tofte@diku.dk.

‡ Work done while at University of Copenhagen. email: MGV_at_NKT_ELEKTRONIK@dscc.dk.

1. Region inference reclaims memory very eagerly and could hence lead to a (much desired) reduction in space requirements;
2. The region information inferred by the region inference algorithm might be useful to programmers who are interested in obtaining guarantees about maximal storage use and maximal lifetimes of data, as is the case with embedded systems;
3. If region inference is used without garbage collection (as we have done so far) it eliminates hidden time costs: all memory management operations are inserted by the compiler and are constant-time operations. This could be important for real-time programming.

The purpose of this paper is to report the results of ongoing efforts to study whether and how this potential can be realised. Based on experience with developing a new Standard ML compiler which uses regions for memory management, we propose a way to map the conceptual regions of region inference onto real machines. With the techniques we present below, we have found that

1. Region inference can result in significant space savings on non-trivial programs, in comparison with a state-of-the-art system which uses garbage collection;
2. Region-based evaluation of ML programs can compete on speed with the garbage-collection-based execution of a state-of-the-art ML system;
3. In practice, a high percentage of all memory allocations can take place on a traditional runtime stack

On the downside, it has to be said that region inference occasionally does not predict lifetimes with sufficient accuracy and that tail recursive calls tend to require special programmer attention. Thus we had to make minor changes to programs to make them run well with regions.

We are currently building an ML compiler to explore region inference; it is called the ML Kit with Regions, since it is built on top of Version 1 of the ML Kit[4].[1] The purpose of this paper is not to describe the Kit, but to describe solutions to key problems which presented themselves, when we tried to compile with regions. These solutions are in the form of additional type-based analyses which refine the information gained with region inference in ways which are

[1] For brevity, we shall refer to it as simply "the Kit", from now on.

essential when the target machine has a conventional linear address space of fixed size words and a number of registers.

The operational region-based semantics presented in [17] treats all values and regions uniformly: all values are put into regions and all regions have a potentially unbounded size. However, we have found that a key factor in achieving good results with region inference is making more careful distinctions between different kinds of regions according to how they should be represented and accessed. The following three kinds of regions fit naturally with common machine architectures:

1. Regions that are used for holding values of a type which fits naturally in a register or a machine word; such regions are not needed at runtime and can hence be eliminated. This situation arises for regions that hold integers and booleans, for example.

2. Regions for which one can infer a finite maximum size at compile time; such regions are conveniently placed on the runtime stack. This situation often applies to regions that hold a tuple or a closure.

3. Regions for which it is not possible to infer a size statically. Such a region can be represented by a linked list of fixed size pages. This situation typically arises when a region contains a list, a tree or some other value of a recursive datatype.

The first analysis we propose is *multiplicity inference*, which infers for each region an upper bound on how many times a value is put into that region. A *boxing analysis* then eliminates regions as described above. Next, *storage mode analysis* infers for each value allocation whether the value should be put at the top of the region (the normal case) or whether it is possible to store the value at the bottom, thereby overwriting any value which the region may already contain. The storage mode analysis involves a *region aliasing analysis*. The storage mode analysis is essential for handling tail recursion.

Multiplicity information and representation information can then be used in *physical size inference* which calculates an upper bound on the physical size of every region.

A key difference between different kinds of regions (besides their sizes) is the way in which they are allocated and accessed. This plays a central rôle in all the analyses. We use the term *Region Representation Inference* for the analyses starting with multiplicity inference and ending with physical size inference.

The Kit has an Abstract Machine (called the KAM) which models a RISC architecture except that it has infinitely many registers. After Region Representation Inference, compilation into the KAM is straightforward. Elsman and Hallenberg[6] have recently completed a backend from KAM to HP PA-RISC assembly language using proven techniques such as intraprocedural register allocation based on graph colouring. A backend generating ANSI C is also available. The ML Kit currently compiles all of Core ML (including recursive datatypes, references, exceptions and higher-order functions); an implementation of Modules is under consideration.

In the rest of the paper we describe the new region-specific program analyses, from multiplicity inference to KAM code generation. Sections 2 and 3 consist mainly of a review of previous work. We start out by presenting the language of region-annotated terms.

2 Source Language

Let Var be a denumerably infinite set of *program variables*, ranged over by x and f. The language of source expressions, e, is defined by:

$$\begin{array}{rl} e ::= & \texttt{true} \mid \texttt{false} \mid x \mid \lambda x.e \mid e_1 e_2 \\ \mid & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\ \mid & \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e\ \texttt{end} \\ \mid & \texttt{letrec}\ f(x)\ \texttt{=}\ e\ \texttt{in}\ e\ \texttt{end} \end{array}$$

Although source expressions appear untyped, region inference is only possible for expressions that are well-typed according to Milner's type discipline[13, 5].

We shall use the following program as a running example:

```
letrec f(x) =
  letrec facacc(p) =
    let n = fst p in let acc = snd p
    in if n=0 then p
         else facacc(n-1,n*acc)
    end end
  in (λy.facacc y, facacc(x+3,1))
  end
in (fst(f 7))(8,1)
end
```

Here we have taken the liberty to extend the skeletal language with pairs, projections (*fst* and *snd*), integer constants, and infix binary operations on integers (+, =, -, *). Also, we use parentheses for grouping. The above expression evaluates to the pair $(0, 8!) = (0, 40320)$.

3 Region-Annotated Terms

Tofte and Talpin[17] describe a type-based translation from source expressions to region-annotated terms (called "target terms" in [17]). These region-annotated terms contain only that type information which is needed for the evaluation of such expressions, namely region annotations. However, in this paper we use the region-annotated expressions as source expressions for further type-based transformations, so it is useful also to have an explicitly typed version of the language. We therefore present both, together with an erase function from explicitly typed to untyped expressions. When convenient, we shall present both an untyped and an explicitly typed version of our intermediate languages; the untyped version contains only the information which is used in the dynamic semantics of the language, while the explicitly typed expression contains information which is used for further translation.

3.1 Untyped Region-Annotated Terms

Let RegVar be a denumerably infinite set of *region variables*, ranged over by ρ. For any syntactic class, c, let $\vec{c}$ denote the syntactic class defined by

$$\begin{array}{rll} \vec{c} ::= & & \text{empty} \\ \mid & c & \\ \mid & c_1, \cdots, c_n & (n \geq 2) \end{array}$$

We now introduce syntactic classes of *allocation directives*, a, *region binders*, b, and *expressions*, e, by

$a ::= \texttt{at}\ \rho \qquad b ::= \rho$

```
e ::= true a | false a | x | (λx.e) a | e1 e2
    |  if e then e else e
    |  let x = e in e end
    |  letregion b in e end
    |  letrec f[b⃗](x) a = e in e end
    |  f [a⃗] a
```

This language of expressions will be used as our untyped language throughout, but we shall gradually refine the definitions of allocation directives and region binders to provide more information.

Let us briefly review the evaluation of region-annotated terms. (Details and an operational semantics are found in [17].) An expression `letregion` ρ `in` e `end` is evaluated thus: first a region is allocated and bound to ρ; then e is evaluated (probably using the region for storing and retrieving values) and then, when `end` is reached, the region is deallocated. An annotation of the form `at` ρ indicates that the value of the expression preceding the annotation should be put into the region bound to ρ. Writing a value into a region adds the value at the one end (referred to as the *top*) of the region, increasing the number of values held in that region by one.

A function f bound by `letrec` is region-polymorphic: it has a (perhaps empty) list of formal region parameters and may be applied to different actual regions at different call sites. An expression f `[`$\vec{a}$`]at`ρ creates a function closure in region ρ, in which the formals of f have been bound to the actual regions $\vec{a}$.

We write `letregion` $\vec{\rho}$ `in` e `end` for

```
letregion ρ1
in ··· letregion ρk in e end ···
end
```

when $\vec{\rho} = \rho_1, \ldots, \rho_k$. Further, f`[`$\rho_1, \ldots, \rho_k$`]` a abbreviates f`[at`$\rho_1, \ldots,$ `at`ρ_k`]` a. Expressions of the form

`letregion` ρ `in` f`[`$\rho_1, \ldots, \rho_k$`]at` ρ (e) `end`

(where $\rho \notin \{\rho_1, \ldots, \rho_k\}$ and ρ does not occur free in e) are so common that we abbreviate them to just f`[`$\rho_1, \ldots, \rho_k$`]` e.

A region-annotated expression corresponding to the source expression in Section 2 is shown in Figure 1.

Aiken, Fähndrich and Levien[1] have developed an analysis which seeks to do the actual region allocation in `letregion` ρ `in` e `end` as late as possible after the `letregion` and which seeks to do the region de-allocation as early as possible before the `end`. In some cases they achieve asymptotic memory savings over plain region inference (where allocation is done at `letregion` and de-allocation is done at `end`), and the result can never be worse than without their analysis.

3.2 Typed Region-Annotated Terms

The type system presented in this section is essentially the one of [17]. The system has its roots in work on effect inference [9, 11, 12, 16], which is also used in connection with concurrency[14].

For the region type system, we assume a denumerably infinite set TyVar of *type variables*, ranged over by α, and a denumerably infinite set EffectVar of *effect variables*, ranged over by ϵ.

An *effect*, φ, is a finite set of atomic effects. An *atomic effect*, η, is either a token of the form $\mathbf{get}(\rho)$ or $\mathbf{put}(\rho)$, or it is an effect variable. *Types*, τ, *types and places*, μ, *simple type schemes*, σ, and *compound type schemes*, π, take the form:

$$
\begin{array}{rcl}
\tau & ::= & \texttt{bool} \mid \alpha \mid \mu \xrightarrow{\epsilon.\varphi} \mu \\
\mu & ::= & (\tau, \rho) \\
\sigma & ::= & \tau \mid \forall\alpha.\sigma \mid \forall\epsilon.\sigma \\
\pi & ::= & \underline{\tau} \mid \forall\alpha.\pi \mid \forall\epsilon.\pi \mid \forall\rho.\pi
\end{array}
$$

An object of the form $\epsilon.\varphi$ (formally a pair (ϵ, φ)) on a function arrow $\mu \xrightarrow{\epsilon.\varphi} \mu'$ is called an *arrow effect*. Here φ is the effect of evaluating the body of the function.

A *finite* map is a map with finite domain. The domain and range of a finite map f are denoted $\mathrm{Dom}(f)$ and $\mathrm{Rng}(f)$, respectively. When f and g are finite maps, $f+g$ is the finite map whose domain is $\mathrm{Dom}(f) \cup \mathrm{Dom}(g)$ and whose value is $g(x)$, if $x \in \mathrm{Dom}(g)$, and $f(x)$ otherwise. $f \downarrow A$ means the restriction of f to A, and $f \setminus\!\setminus A$ means f restricted to the complement of A.

A *type environment*, *TE*, is a finite map from program variables to pairs of the form (σ, ρ) or (π, ρ).

A *substitution* S is a triple (S^r, S^t, S^e), where S^r is a finite map from region variables to region variables, S^t is a finite map from type variables to types and S^e is a finite map from effect variables to arrow effects. Its effect is to carry out the three substitutions simultaneously on the three kinds of variables.

For any compound type scheme

$$\pi = \forall\rho_1 \cdots \rho_k \alpha_1 \cdots \alpha_n \epsilon_1 \cdots \epsilon_m.\underline{\tau}$$

```
letregion ρ6 in
 letrec f[ρ0,ρ7,ρ8,ρ9,ρ10,ρ11,
          ρ12,ρ13,ρ14,ρ15](x) at ρ6=
  letrec facacc[ρ16,ρ17,ρ18](p) at ρ7=
    let n= fst p in let acc= snd p in
     letregion ρ19 in
      if letregion ρ21 in
         (n = 0 at ρ21) at ρ19
         end then p
      else facacc[ρ16,ρ17,ρ18]
             (letregion ρ24 in
               (n - 1 at ρ24) at ρ18
              end,(n*acc)at ρ17)at ρ16
     end end end
  in
   ((λy.facacc[ρ13,ρ14,ρ15]y)at ρ12,
    facacc[ρ9,ρ10,ρ11]
       (letregion ρ27 in
         (x + 3 at ρ27) at ρ11
        end, 1 at ρ10) at ρ9) at ρ8
  end
 in letregion ρ28,ρ29,ρ30,ρ31,ρ32 in
      (letregion ρ33,ρ34,ρ35,ρ36 in
        fst(letregion ρ37 in
              f[ρ37,ρ31,ρ33,ρ34,ρ35,ρ36,
                ρ32,ρ28,ρ29,ρ30] 7 at ρ37
            end)
       end) (8 at ρ30, 1 at ρ29) at ρ28
    end
 end
end
```

Figure 1: A Region-annotated Expression

and type τ', we say that τ' is *an instance of* π *(via* S*)*, written $\pi \geq \tau'$, if there exists a substitution

$$S = (\{\rho_1 \mapsto \rho_1', \ldots, \rho_k \mapsto \rho_k'\}, \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}, \{\epsilon_1 \mapsto \epsilon_1'.\varphi_1, \ldots, \epsilon_m \mapsto \epsilon_m'.\varphi_m\})$$

such that $S(\tau) = \tau'$. Similarly for simple type schemes. The *instance list of* S, written $il(S)$, is the triple

$$([\rho_1', \ldots, \rho_k'], [\tau_1, \ldots, \tau_n], [\epsilon_1'.\varphi_1, \ldots \epsilon_m'.\varphi_m])$$

More generally, we refer to triples of above form as instance lists and use il to range over them. Instance lists decorate applied (i.e., non-binding) occurrences of program variables.

We now present a type system for explicitly typed region-annotated terms. It allows one to infer sentences of the form $TE \vdash e : \mu, \varphi$. Formally, an *explicitly typed region-annotated term* is a term e, for which there exist μ and φ such that $TE \vdash e : \mu, \varphi$. For given TE and e there is at most one such μ and φ (and at most one derivation proving $TE \vdash e : \mu, \varphi$). The type system is essentially the same as the one in [17], except that we have dropped the source expressions and added type, region and effect annotations on terms.

Region-Annotated Terms $\quad \boxed{TE \vdash e : \mu, \varphi}$

$$\frac{}{TE \vdash \texttt{true at } \rho : (\texttt{bool}, \rho), \{\mathbf{put}(\rho)\}} \tag{1}$$

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

$$\frac{TE(x) = (\sigma, \rho) \quad \sigma \geq \tau \ via \ S}{TE \vdash x_{il(S)} : (\tau, \rho), \emptyset} \tag{3}$$

$$\frac{TE + \{x \mapsto \mu_1\} \vdash e : \mu_2, \varphi \qquad \varphi \subseteq \varphi'}{TE \vdash (\lambda^{\epsilon.\varphi'} x : \mu_1.e) \texttt{ at } \rho : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \rho), \{\mathbf{put}(\rho)\}} \tag{4}$$

$$\frac{\begin{array}{c} TE \vdash e_1 : (\mu' \xrightarrow{\epsilon.\varphi_0} \mu, \rho), \varphi_1 \\ TE \vdash e_2 : \mu', \varphi_2 \\ \varphi = \varphi_0 \cup \varphi_1 \cup \varphi_2 \cup \{\epsilon\} \cup \{\mathbf{get}(\rho)\} \end{array}}{TE \vdash e_1 \ e_2 : \mu, \varphi} \tag{5}$$

$$\frac{TE \vdash e_1 : (\texttt{bool}, \rho), \varphi_1 \quad TE \vdash e_2 : \mu, \varphi_2 \quad TE \vdash e_3 : \mu, \varphi_3}{\begin{array}{l} TE \vdash \ \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \\ \qquad : \mu, \{\mathbf{get}(\rho)\} \cup \varphi_1 \cup \varphi_1 \cup \varphi_3 \end{array}} \tag{6}$$

$$\frac{\begin{array}{c} TE \vdash e_1 : (\tau_1, \rho_1), \varphi_1 \\ \sigma_1 = \forall \vec{\alpha}\, \vec{\epsilon}.\tau_1 \quad \mathrm{fv}(\vec{\alpha}, \vec{\epsilon}) \cap \mathrm{fv}(TE, \varphi_1) = \emptyset \\ TE + \{x \mapsto (\sigma_1, \rho_1)\} \vdash e_2 : \mu, \varphi_2 \end{array}}{TE \vdash \ \texttt{let } x : (\sigma_1, \rho_1) \texttt{ = } e_1 \texttt{ in } e_2 \texttt{ end} : \mu, \varphi_1 \cup \varphi_2} \tag{7}$$

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

$$\frac{\begin{array}{c} TE(f) = (\pi, \rho') \\ \pi \geq \tau \text{ via } S \quad \varphi = \{\mathbf{get}(\rho'), \mathbf{put}(\rho)\} \end{array}}{TE \vdash f_{il(S)} \texttt{ at } \rho : (\tau, \rho), \varphi} \tag{9}$$

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

For any semantic object A, frv(A) denotes the set of region variables that occur free in A, ftv(A) denotes the set of type variables that occur free in A, fev(A) denotes the set of effect variables that occur free in A and fv(A) denotes the union of all of the above.

The *erasure* of an explicitly typed region-annotated expression e, written $er(e)$, is an untyped, region-annotated expression obtained by erasing type and effect information. We show a couple of the defining equations:

$er(\texttt{letregion } \varphi \texttt{ in } e \texttt{ end}) =$
$\quad \texttt{letregion frv}\,\varphi \texttt{ in } er(e) \texttt{ end}$
$er(\texttt{letrec } f : (\forall \vec{\rho}\vec{\epsilon}\vec{\alpha}.\tau, \rho) \texttt{ (x) =} e_1 \texttt{in } e_2 \texttt{ end}) =$
$\quad \texttt{letrec } f\texttt{[}\vec{\rho}\texttt{]at}\rho\texttt{=}er(e_1) \texttt{ in } er(e_2) \texttt{ end}$
$er(f_{(\_,[\rho_1,\ldots,\rho_k],\_)} \texttt{at } \rho) = f\texttt{[}\rho_1, \ldots, \rho_k\texttt{] at } \rho$

4 Multiplicity Inference

Multiplicity Inference is concerned with inferring for each region, how many times a value is put into that region. We introduce a syntactic class of *multiplicities*, ranged over by m:

$$m ::= 0 \mid 1 \mid \infty$$

Addition of multiplicities is defined by:

$$m_1 \oplus m_2 = \begin{cases} 0 & \text{if } m_1 = m_2 = 0; \\ 1 & \text{if } m_1 = 0 \wedge m_2 = 1 \text{ or vice versa} \\ \infty & \text{otherwise} \end{cases}$$

The *maximum* of m_1 and m_2, written $max(m_1, m_2)$, and the *product* of m_1 and m_2, written $m_1 \otimes m_2$, are defined similarly.

4.1 Untyped Multiplicity-Annotated Terms

We modify the class of region binders to become:

$$b ::= \rho : m$$

Let us assume that every region variable ρ is only bound once in any given expression. We then define the *multiplicity of* ρ, written $mul(\rho)$, to be the multiplicity which occurs in the binder which binds ρ, and ∞ otherwise (i.e., if ρ is free).

Evaluation of multiplicity-annotated expressions can be defined using an operational semantics which has two region stacks, namely a stack of regions each of which can accept at most one write and a stack of regions each of which can accept an unbounded number of writes. (The dynamic semantics for region annotated terms in [17] has only the second kind of region stack.)

In an expression of the form

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

the multiplicity m is an upper bound on the number of times a value is put into the region which will be bound to ρ at runtime. Thus, if $m = \infty$ we allocate a region on the stack of unbounded regions and otherwise on the stack of write-once regions.

In an expression

```
letrec f[...,ρ:m,...](x) a = e1 in e2
```

the multiplicity m is an upper bound on how many times the evaluation of the body of f (i.e., e_1) puts a value into ρ — including calls that f may make to other functions or to itself. Consider a reference to f (in e_1 or in e_2)

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

It is possible to have $mul(\rho) < \infty$ and $mul(\rho') = \infty$, signifying that f contributes a finite number of allocations to an unbounded region. Also, f is polymorphic in multiplicities, in the sense that if we have some other call of f:

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

we need not have $mul(\rho'') = mul(\rho')$. This flexibility was found to be important in practice — without it, too many regions were ascribed multiplicity ∞. However, it means that the dynamic region environment has to map region variables to pairs of the form (r, m), where r is a region name (identifying the region) and m is the multiplicity of the region. At runtime, the multiplicity of a region is determined by the `letregion` expression which generates it and it never changes, so (r, m) can be regarded as a region name r with a multiplicity attribute m.

When storing a value into a letrec-bound ρ it is now necessary to test at runtime to see what kind of store operation should be performed. Allocation in the two kinds of regions is done differently; for unbounded regions we first have to allocate new space within the region, but for write-once regions, we can write directly knowing that there will be space for one write.

4.2 Typed Multiplicity-Annotated Terms

A *multiplicity effect* is a finite map from atomic effects to multiplicities; we use ψ to range over multiplicity effects. The extension of ψ to a total map which is 0 outside the domain of ψ is denoted ψ^+. Let ψ_1 and ψ_2 be multiplicity effects. The *sum* of ψ_1 and ψ_2, written $\psi_1 \oplus \psi_2$, is the multiplicity effect which has domain $\mathrm{Dom}(\psi_1) \cup \mathrm{Dom}(\psi_2)$ and values

$$(\psi_1 \oplus \psi_2)(\eta) = \psi_1^+(\eta) \oplus \psi_2^+(\eta)$$

Similarly, the *maximum* of ψ_1 and ψ_2, written $max(\psi_1, \psi_2)$, is defined by

$$(max(\psi_1, \psi_2))(\eta) = max(\psi_1^+(\eta), \psi_2^+(\eta))$$

Finally, when ψ is a multiplicity effect and m is a multiplicity, the *scalar product*, $m \otimes \psi$, is the multiplicity effect with the same domain as ψ and values $(m \otimes \psi)(\eta) = m \otimes (\psi(\eta))$.

The semantic objects of typed multiplicity-annotated terms are as those for typed region-annotated terms, except that effects are replaced by multiplicity effects everywhere. We shall also use τ, S etc. to range over semantic objects with multiplicities, and then use vertical bars ($|\tau|, |S|, \ldots$) to refer to the semantic objects obtained by replacing every multiplicity effect with its domain, which is an effect. We write $\psi' \geq \psi$ to mean $|\psi'| = |\psi|$ and $\psi'(\eta) \geq \psi(\eta)$, for all $\eta \in \mathrm{Dom}(\psi)$.

The typing rules for multiplicity-annotated terms are:

Multiplicity-Annotated Terms $\boxed{TE \vdash e : \mu, \psi}$

$$\frac{}{TE \vdash \texttt{true}\ \texttt{at}\ \rho : (\texttt{bool}, \rho), \{\mathbf{put}(\rho) \mapsto 1\}} \tag{11}$$

$$\frac{}{TE \vdash \texttt{false}\ \texttt{at}\ \rho : (\texttt{bool}, \rho), \{\mathbf{put}(\rho) \mapsto 1\}} \tag{12}$$

$$\frac{TE(x) = (\sigma, \rho) \quad \sigma \geq \tau \ via\ S}{TE \vdash x_{il(S)} : (\tau, \rho), \{\}} \tag{13}$$

$$\frac{TE + \{x \mapsto \mu_1\} \vdash e : \mu_2, \psi \qquad \psi \oplus \psi'' = \psi'}{TE \vdash (\lambda^{\epsilon.\psi'} x : \mu_1.e)\ \texttt{at}\ \rho : (\mu_1 \xrightarrow{\epsilon.\psi'} \mu_2, \rho), \{\mathbf{put}(\rho) \mapsto 1\}} \tag{14}$$

$$\frac{\begin{array}{c} TE \vdash e_1 : (\mu' \xrightarrow{\epsilon.\psi_0} \mu, \rho), \psi_1 \\ TE \vdash e_2 : \mu', \psi_2 \\ \psi = \psi_0 \oplus \psi_1 \oplus \psi_2 \oplus \{\epsilon \mapsto 1\} \oplus \{\mathbf{get}(\rho) \mapsto 1\} \end{array}}{TE \vdash e_1\ e_2 : \mu, \psi} \tag{15}$$

$$\frac{TE \vdash e_1 : (\texttt{bool}, \rho), \psi_1 \quad TE \vdash e_2 : \mu, \psi_2 \quad TE \vdash e_3 : \mu, \psi_3}{\begin{array}{l} TE \vdash\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\ \quad : \mu, \{\mathbf{get}(\rho) \mapsto 1\} \oplus \psi_1 \oplus max(\psi_2, \psi_3) \end{array}} \tag{16}$$

$$\frac{\begin{array}{c} TE \vdash e_1 : (\tau_1, \rho_1), \psi_1 \\ \sigma_1 = \forall \vec{\alpha}\,\vec{\epsilon}.\tau_1 \quad \mathrm{fv}(\vec{\alpha}, \vec{\epsilon}) \cap \mathrm{fv}(TE, \psi_1) = \emptyset \\ TE + \{x \mapsto (\sigma_1, \rho_1)\} \vdash e_2 : \mu, \psi_2 \end{array}}{TE \vdash\ \texttt{let}\ x : (\sigma_1, \rho_1)\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end} : \mu, \psi_1 \oplus \psi_2} \tag{17}$$

$$\frac{\begin{array}{c} \pi = \forall \vec{\rho}\,\vec{\epsilon}.\underline{\tau} \quad \pi' = \forall \vec{\alpha}.\pi \quad \mathrm{fv}(\vec{\alpha}, \vec{\rho}, \vec{\epsilon}) \cap \mathrm{fv}(TE, \psi_1) = \emptyset \\ TE + \{f \mapsto (\pi, \rho)\} \vdash (\lambda^{\epsilon'.\psi'} x : \mu_x.e_1)\ \texttt{at}\ \rho : (\tau, \rho), \psi_1 \\ TE + \{f \mapsto (\pi', \rho)\} \vdash e_2 : \mu, \psi_2 \end{array}}{TE \vdash\ \texttt{letrec}\ f : (\pi', \rho)(x)\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end} : \mu, \psi_1 \oplus \psi_2} \tag{18}$$

$$\frac{\begin{array}{c} TE(f) = (\pi, \rho') \\ \pi \geq \tau \text{ via } S \quad \psi = \{\mathbf{get}(\rho') \mapsto 1, \mathbf{put}(\rho) \mapsto 1\} \end{array}}{TE \vdash\ f_{il(S)}\ \ \texttt{at}\ \ \rho : (\tau, \rho), \psi} \tag{19}$$

$$\frac{TE \vdash e : \mu, \psi \quad \mathrm{fv}(\psi') \cap \mathrm{fv}(TE, \mu) = \emptyset}{TE \vdash \texttt{letregion}\ \psi'\ \texttt{in}\ e'\ \texttt{end} : \mu, \psi \setminus\!\!\setminus |\psi'|} \tag{20}$$

$$\frac{TE \vdash e : \mu, \psi \qquad \psi' \geq \psi}{TE \vdash e : \mu, \psi'} \tag{21}$$

Note that union of effects has turned into sum of multiplicity effects, except at the conditional, where maximum is used. A more substantial change is in the definition of what it means to apply substitutions (rules 13 and 19 rely on this)

A *(multiplicity) substitution* is a triple $S = (S^r, S^t, S^e)$, where S^t is a map from type variables to types, S^r is a map from region variables to region variables and S^e is a map from effect variables to multiplicity arrow effects $(\epsilon.\psi)$. Each of these finite maps extend to total maps — in the case of S^e by mapping each effect variable ϵ outside the domain of S^e to the multiplicity arrow effect $\epsilon.\{\}$.

We define

$$
\begin{aligned}
S^r(\psi) \;&=\; \bigoplus\{\mathbf{put}(S^r(\rho)) \mapsto \psi(\mathbf{put}(\rho)) \mid \mathbf{put}(\rho) \in |\psi|\} \\
&\oplus\; \bigoplus\{\mathbf{get}(S^r(\rho)) \mapsto \psi(\mathbf{get}(\rho)) \mid \mathbf{get}(\rho) \in |\psi|\} \\
&\oplus\; \psi \downarrow \mathrm{EffectVar}
\end{aligned}
$$

$$
\begin{aligned}
S^e(\psi) \;=\; & \psi \setminus\!\setminus \mathrm{EffectVar} \oplus \\
& \bigoplus\{\psi(\epsilon) \otimes (\{\epsilon' \mapsto 1\} \oplus \psi') \mid \\
& \quad \epsilon \in \mathrm{Dom}(\psi) \wedge \epsilon'.\psi' = S^e(\epsilon)\}
\end{aligned}
$$

Moreover, define

$$S^e(\epsilon.\psi) = \epsilon'.(\psi' \oplus S^e(\psi))$$

where $\epsilon'.\psi' = S^e(\epsilon)$. Finally, we define

$$(S^r, S^t, S^e)(A) = S^t(S^e(S^r(A)))$$

where A can be an arrow effect, a type or a type and place. Substitutions can also be applied to type schemes, after renaming of bound variables to avoid capture, when necessary. Finally, a substitution can be applied to a type environment TE by applying it to every pair (σ, ρ) or (π, ρ) in the range of TE.

We say that a multiplicity-annotated expression e is *well-annotated* in TE if there exists a μ and a ψ such that $TE \vdash e : \mu, \psi$. For given TE and e, there exists at most one such μ and ψ.

Multiplicity Inference is the following problem: given TE, e, μ and φ with $TE \vdash e : \mu : \varphi$ according to rules (1)–(10) and given a multiplicity type environment TE' with $|TE'| = TE$, find a multiplicity-annotated term e' which is well-annotated in TE' and satisfies $|e'| = e$.

When e is closed, there is a trivial solution to the Multiplicity Inference Problem: choose all multiplicities to be ∞. The object is of course to choose multiplicities as small as possible.

Vejlstrup's M.Sc. thesis[18] contains a multiplicity inference algorithm and a proof that it is correct and always terminates. The algorithm does not always find minimal multiplicities. One problem is that substitution and maximum do not commute; in general one only has $S^e(max(\psi_1, \psi_2)) \geq max(S^e(\psi_1), S^e(\psi_2))$. In particular, if a lambda-bound variable, f, occurs in two different conditionals, unification on the type of of f during the multiplicity inference of the second conditional can increase the effect of the first conditional:

$\lambda\, f : ((\mathtt{int}, \rho_1) \xrightarrow{\epsilon_1.\{\mathbf{put}(\rho_2)\}} (\mathtt{int}, \rho_2), \rho_3)$.
  $\mathtt{let}\ x$ = $\mathtt{if\ true\ then\ 1\ at}\ \rho_2\ \mathtt{else}\ f(\mathtt{1\ at}\ \rho_1)$
  $\mathtt{in\ if\ true\ then}\ (\lambda y.1\ \mathtt{at}\ \rho_2)\ \mathtt{at}\ \rho_3\ \mathtt{else}\ f$

Here the effect of evaluating x will end up having two put effects on ρ_2, although one would be sound.

Judging from experience, however, the algorithm is usually good at detecting finite regions (see Section 9).

Erasure of a typed multiplicity-annotated term gives an untyped multiplicity-annotated term. We show some of the defining equations:

$er(\mathtt{letregion}\ \psi\ \mathtt{in}\ e\ \mathtt{end}) =$
  $\mathtt{letregion}\ b_1 \cdots b_k\ \mathtt{in}\ er(e)\ \mathtt{end}$

where $\{\rho_1, \ldots, \rho_k\} = \mathrm{frv}(\psi)$ and

$$b_i = \rho_i : \psi^+(\mathbf{put}(\rho_i))$$

for $i = 1 \ldots k$.

$er(\mathtt{letrec}\ f : (\pi', \rho)(x)\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2\ \mathtt{end}) =$
  $\mathtt{letrec}\ f[b_1, \ldots, b_k]\mathtt{at}\rho\mathtt{=}er(e_1)\ \mathtt{in}\ er(e_2)\ \mathtt{end}$

where $\pi' = \forall \rho_1, \ldots, \rho_k \vec{\alpha}\vec{\epsilon}.(\mu_1 \xrightarrow{\epsilon.\psi} \mu_2)$ and $b_i = \rho_i : \psi^+(\mathbf{put}(\rho_i))$, $i = 1 \ldots k$.

4.3 Removal of get-regions

Consider a declaration of the form

$$\mathtt{letrec}\ f[\vec{b}](x)\ a\ \mathtt{=}\ e\ \mathtt{in}\ e\ \mathtt{end}$$

Write $\vec{b}$ in the form $\rho_1 : m_1, \ldots, \rho_k : m_k$. If $\rho \in \{\rho_1, \ldots, \rho_k\}$ is such that there is no $\mathbf{put}(\rho)$ anywhere in the type of f, then f does not really need ρ: putting a value into a region requires region information, but reading a value does not. Such region variables are called **get**-regions (of f). They can be eliminated from the list of region formals, provided the corresponding actual arguments in calls of f are removed too.

```
letregion ρ6:1 in
 letrec f[ρ7:1,ρ8:1,ρ9:∞, ρ10:∞,ρ11:∞,
          ρ12:1,ρ13:0,ρ14:0,ρ15:0](x) at ρ6 =
  letrec facacc[ρ16:∞,ρ17:∞,ρ18:∞](p) at ρ7=
    let n = fst p in  let acc = snd p in
     letregion ρ19:1 in
      if letregion ρ21:1 in
         (n = 0 at ρ21) at ρ19
         end then p
      else facacc[ρ16,ρ17,ρ18]
             (letregion ρ24:1 in
               (n - 1 at ρ24) at ρ18
              end, (n * acc) at ρ17) at ρ16
     end end end
  in
   ((λ y.facacc[ρ13,ρ14,ρ15]y) at ρ12,(*1*)
    facacc[ρ9,ρ10,ρ11]
       (letregion ρ27:1
         (x + 3 at ρ27) at ρ11
        end, 1 at ρ10) at ρ9) at ρ8
  end
 in letregion ρ28:∞,ρ29:∞,ρ30:∞,ρ31:1,ρ32:1
    in (letregion ρ33:1,ρ34:∞,ρ35:∞,ρ36:∞
        in fst (letregion ρ37:1 in
                f[ρ31,ρ33,ρ34,ρ35,ρ36,
                  ρ32,ρ28,ρ29,ρ30] 7 at ρ37
               end)
        end) (8 at ρ30, 1 at ρ29) at ρ28
    end
 end
end
```

Figure 2: After Multiplicity Inference and elimination of **get**-regions

In what follows, we always use the more aggressive erase operations which removes both type information and **get**-regions. The erasure of a typed multiplicity-annotated expression which corresponds to the region annotated example in Figure 1 is shown in Figure 2. Notice that most region binders have been given finite multiplicity and that f has had the **get**-region ρ_0 removed. The $\lambda y.facacc\, y$ in line `(*1*)` is put into a write-once region (ρ_{32}), which eventually is stack-allocated, even though the closure "escapes".

5 Unboxed Values

In the plain region inference scheme[17], every value is represented "boxed", i.e., by a pointer to the actual value, which resides in a region. However, it is not necessary to box values whose natural size is not bigger than what a register can hold. Let us refer to such values as *word*-sized. In the ML Kit, the word-sized values are conservatively defined to be precisely the integers and the booleans. Storing a word-sized value allocates no space in memory; it just stores the value in a register.

Let r be a region at runtime. If all put operations on r are putting word-sized values, then no values at all are put into r, and r could be eliminated altogether. *This holds, even if there are multiple put operations to the region.* For every storage operation v `at` ρ in the program, enough of the type of v is known statically to decide the appropriate representation (boxed/unboxed). This relies on the fact that v is a syntactic value. Detecting whether *all* storage operations to ρ store word-sized values requires a simple region flow analysis, which we describe in this section. If ρ is a formal parameter of some `letrec`-bound function, f, and all stores to ρ are stores of word-sized values, then ρ is removed from the list of formal parameters of f, and all the corresponding actuals in applications of f are removed too. *This is true even if the multiplicity in the binder of ρ is not finite.* This removes many region parameters in practice.

In ML, all functions take one argument; "multiple" arguments are represented by a tuple which in the Kit always is a boxed tuple. This is simple but inefficient. No doubt, careful data representation analysis[10, 15, 8] would be very useful with regions. This has not yet been explored, however.

5.1 Modified Syntax

We extend allocation directives to become

$a ::=$ `at` ρ | `ignore`

In examples, we abbreviate v `ignore` to v.

In the dynamic semantics, evaluating v `ignore` just results in v without performing any allocation in any region.

5.2 Boxity Constraints

Let RegionTyVar be a denumerably infinite set of *region type variables*, ranged over by r. We introduce region types rt:

$$rt ::= \perp \mid \texttt{word} \mid \top \mid r$$

Ground region types are ordered by $\perp \sqsubseteq$ `word` $\sqsubseteq \top$. Intuitively, a region can be given type `word` if all the values stored in it are word-sized. (The region need not have finite multiplicity.) Top ($\top$) stands for all types that are not of word size, e.g., record types and function types. Bottom ($\perp$) is the type of region variables ρ for which no `at` ρ occurs in the program. (Get-regions of region-polymorphic functions have region type $\perp$, if they are not removed.)

```
letregion ρ6:1 in
 letrec f[ρ7:1,ρ8:1,ρ9:∞,
            ρ12:1,ρ13:0](x) at ρ6 =
  letrec facacc[ρ16:∞](p) at ρ7 =
   let n= fst p in let acc= snd p in
     if n = 0 then p
     else facacc[ρ16]((n-1,n*acc)at ρ16)
   end end
  in
   ((λy.facacc[ρ13]y) at ρ12,
    facacc[ρ9]((x + 3, 1)at ρ9))at ρ8
  end
 in letregion ρ28:∞,ρ31:1,ρ32:1 in
      (letregion ρ33:1,ρ34:∞ in
        fst(f[ρ31,ρ33,ρ34,ρ32,ρ28] 7)
       end)((8,1)at ρ28)
    end
 end
end
```

Figure 3: After elimination of `word`-typed regions

The type system of region types is monomorphic in that every region variable is assigned a ground region type.

The analysis which assigns region types to region binders is a simple constraint-based analysis. A *constraint* takes one of the two forms $r \sqsupseteq rt$ or $r \sqsupseteq r'$. A finite set of constraints has a minimal solution (with respect to $\sqsubseteq$). It can be shown that this solution can be found in time which is linear in the number of constraints in the set.

Constraints are generated as follows: every binder $\rho : m$ is associated with a fresh region type variable, written $r(\rho)$. For every subexpression `true at` ρ or `false at` ρ of e, we generate a constraint $r(\rho) \sqsupseteq$ `word`. For all other `at` ρ in e we generate a $r(\rho) \sqsupseteq \top$ constraint. Furthermore, if f is declared by

`letrec` f`[`$\rho'_1 : m_1, \ldots, \rho'_k : m_k$`]` (x) `at` ρ `=` e_1 `in` e_2

then for every reference to f:

$$f\texttt{[}\rho_1, \ldots, \rho_k\texttt{]}\ \texttt{at}\ \rho$$

we generate the k constraints $r(\rho_i) \sqsupseteq r(\rho'_i)$, $1 \le i \le k$.

Once the minimal solution has been found, every region binder $\rho : m$ which has been assigned region type `word` or less is removed from the program, thus reducing the number of letregions and the number of parameters to region-polymorphic functions. Furthermore, all allocation directives `at` ρ (for the ρ in question) are changed into `ignore`. When a formal region parameter is removed, all corresponding actuals must of course be removed too.

The result of removing `word` regions from Figure 2 is shown in Figure 3. Notice that by now, all `letregion`-bound region variables with infinite multiplicity, except ρ_{28} and ρ_{34}, have been eliminated. At runtime, there will be just two infinite regions.

6 Storage Mode Analysis

The purpose of storage mode analysis was explained in the Introduction. It operates with the following allocation directives and binders

$$\begin{array}{rcl} a & ::= & \texttt{at}\,\rho \mid \texttt{attop}\,\rho \mid \texttt{atbot}\,\rho \mid \texttt{sat}\,\rho \\ b & ::= & \rho : m \end{array}$$

In the input expression, all allocation directives take the form $\texttt{at}\rho$; in the output, every `at` has been turned into `attop`, `atbot` or `sat`. The idea is that one can transform $\texttt{at}\,\rho$ into $\texttt{atbot}\,\rho$ at some program point p, if and only if, whenever p is reached during evaluation, the rest of the evaluation does not use a value which has already been stored into the region to which ρ is bound. Storage mode `attop` should be used when it is certain that the region will contain live values; `sat` ("somewhere at") should be used when the decision about storage mode should be delayed till runtime (typically when ρ is `letrec`-bound).

Storage Mode Analysis is based on statically inferred liveness properties. Liveness analysis has to take temporary values into account. Inspired by the A-Normal Form of Flanagan *et al*[7], we shall therefore assume that the input expression to the storage mode analysis conforms to the following grammar of *region annotated K-Normal Form expressions*:

$$\begin{array}{rcl} e & ::= & x_{il} \mid v\,a \mid x_{il}\,x_{il} \mid f_{il}\,\texttt{[}\vec{a}\texttt{]}\ a_0\,x_{il} \\ & \mid & \texttt{if}\ x_{il}\ \texttt{then}\ e\ \texttt{else}\ e \\ & \mid & \texttt{let}\ x:(\sigma,\rho)\ \texttt{=}\ e\ \texttt{in}\ e \\ & \mid & \texttt{letrec}\ f:(\pi,\rho_0)\texttt{[}\vec{b}\texttt{]}\ (x)\ a_0\ \texttt{=}\ e \\ & & \texttt{in}\ e\ \texttt{end} \\ & \mid & \texttt{letregion}\ b\ \texttt{in}\ e\ \texttt{end} \\ v & ::= & \texttt{true} \mid \texttt{false} \mid \lambda x:\mu.e \end{array}$$

The key idea is that every intermediate result of the computation is bound to a variable. The type information (σ,π and μ) in K-Normal Forms is provided by region inference. Transformation into K-Normal Form can be done in linear time and does not affect the runtime behaviour of the expression. (Unlike Flanagan *et al* we do not linearise `let` bindings, as this would affect region inference in a negative way.)

To enable region polymorphic functions to be applied in contexts that allow different degrees of region overwriting, we pass the storage mode itself along with the region at runtime. Thus we have not only multiplicity polymorphism (Section 4) but also *storage mode polymorphism* (since we found that not having it made too many regions too big).

At runtime, a region may be accessible via more than one region variable, if it is passed as actual argument to a region polymorphic function. This is called *region aliasing.* Storage Mode Analysis must take region aliasing into account. We propose the following global, higher-order region flow analysis. A directed graph G is built. There is one node in G for every region variable and every effect variable which occurs in the (K-normalised) program. (Thus, we can identify variables with nodes.) Whenever the program has a `letrec`-bound program variable f with type scheme

$$\pi = \forall \cdots \rho_i \cdots, \vec{\alpha}, \cdots \epsilon_j \cdots . \underline{\tau}$$

and whenever there is an applied occurrence of f:

$$f_{([\cdots \rho'_i \cdots],[\vec{\tau}],[\cdots \epsilon'_j.\varphi'_j \cdots])}$$

there is an edge from ρ_i to ρ'_i and from ϵ_j to ϵ'_j. Similarly for `let`-bound variables. Finally, for every effect $\epsilon.\varphi$ occurring anywhere in the program, there is an edge from ϵ to every region and effect variable which occurs free in φ. In the graph that arises thus, `letregion` bound variables are always leaf nodes and region variables only lead to region variables. For every node n in G, let $\langle n \rangle$ denote the set of variables that are reachable in G starting from n, including n itself.

Let ρ be a region variable and let e be a region annotated expression which first binds ρ and then refers to ρ. The storage mode analysis depends on a distinction between whether there is a λ between the binder of ρ and the use of ρ, or not. To be able to make this distinction precise, we introduce three kinds of contexts. Two of these are *local*, meaning that they do not allow going under lambda (or `letrec`). A *local expression context*, L, takes the form

$$\begin{array}{rcl} L & ::= & [\,] \\ & \mid & \texttt{if}\ x_{il}\ \texttt{then}\ L\ \texttt{else}\ e_3 \\ & \mid & \texttt{if}\ x_{il}\ \texttt{then}\ e_2\ \texttt{else}\ L \\ & \mid & \texttt{let}\ x:(\sigma,\rho)\ \texttt{=}\ L\ \texttt{in}\ e_2 \\ & \mid & \texttt{let}\ x:(\sigma,\rho)\ \texttt{=}\ e_1\ \texttt{in}\ L \\ & \mid & \texttt{letrec}\ f:(\pi,\rho_0)\texttt{[}\vec{b}\texttt{]}\ (x)\ a_0\ \texttt{=}\ e_1 \\ & & \texttt{in}\ L\ \texttt{end} \\ & \mid & \texttt{letregion}\ b\ \texttt{in}\ L\ \texttt{end} \end{array}$$

Next, *local allocation contexts*, R, are given by

$$\begin{array}{rcl} R & ::= & L[v\,[\,]] \\ & \mid & L[f_{il}\,\texttt{[}a_1,\ldots,a_{i-1},[\,],a_{i+1},\ldots,a_k\texttt{]}\ a_0\,x_{il}] \\ & \mid & L[f_{il}\,\texttt{[}\vec{a}\texttt{]}\ [\,]\,x_{il}] \\ & \mid & L[\texttt{letrec}\ f:(\pi,\rho_0)\texttt{[}\vec{b}\texttt{]}\ (x)\ [\,]\ \texttt{=}\ e_1 \\ & & \quad \texttt{in}\ e_2\ \texttt{end}] \end{array}$$

The last of the three kinds of context is a *(global) expression context*, which allows one to single out an arbitrary subexpression:

$$\begin{array}{rcl} E & ::= & L \\ & \mid & L[\texttt{let}\ x:(\sigma,\rho)\ \texttt{=}\ (\lambda x:\mu.E)\,a\ \texttt{in}\ e\ \texttt{end}] \\ & \mid & L[\texttt{letrec}\ f:(\pi,\rho)\texttt{[}\vec{b}\texttt{]}\ (x)\ a_0\ \texttt{=}\ E\ \texttt{in}\ e_2\ \texttt{end}] \end{array}$$

Given a local context L we say that a program variable x is *live at the hole of the context* if it is a member of the set $\mathrm{LV}(L)$, defined by:

$\mathrm{LV}([\,]) = \emptyset$
$\mathrm{LV}(\texttt{if}\ x_{il}\ \texttt{then}\ L\ \texttt{else}\ e_3) = \mathrm{LV}(L)$
$\mathrm{LV}(\texttt{if}\ x_{il}\ \texttt{then}\ e_2\ \texttt{else}\ L) = \mathrm{LV}(L)$
$\mathrm{LV}(\texttt{let}\ x:(\sigma,\rho)\ \texttt{=}\ L\ \texttt{in}\ e_2\ \texttt{end}) =$
$\quad \mathrm{LV}(L) \cup (\mathrm{FV}(e_2) \setminus \{x\})$
$\mathrm{LV}(\texttt{let}\ x:(\sigma,\rho)\ \texttt{=}\ e_1\ \texttt{in}\ L\ \texttt{end}) = \mathrm{LV}(L)$
$\mathrm{LV}(\texttt{letrec}\ f:(\pi,\rho_0)\texttt{[}\vec{b}\texttt{]}\ (x)\ a_0\ \texttt{=}\ e_1$
$\quad \texttt{in}\ L\ \texttt{end}) = \mathrm{LV}(L)$
$\mathrm{LV}(\texttt{letregion}\ b\ \texttt{in}\ L\ \texttt{end}) = \mathrm{LV}(L)$

Here $\mathrm{FV}(e)$ means the set of program variables that occur free in e.

The definition is extended to local allocation contexts, R, as follows:

$$\mathrm{LV}(L[v\;[\,]]) = \mathrm{FV}(v) \cup \mathrm{LV}(L) \tag{22}$$

$$\mathrm{LV}(L[f_{il}\,\texttt{[}a_1,\ldots,a_{i-1},[\,],a_{i+1},\ldots,a_k\texttt{]}\;\;a_0\,x_{il}]) = \mathrm{LV}(L) \tag{23}$$

$$\mathrm{LV}(L[f_{il}\,\texttt{[}\vec{a}\texttt{]}\;\;[\,]\,x_{il}]) = \{f,x\} \cup \mathrm{LV}(L) \tag{24}$$

$\mathrm{LV}(L[\texttt{letrec}\;f:(\pi,\rho_0)\texttt{[}\vec{b}\texttt{]}\,\texttt{(}x\texttt{)}\,[\,]\texttt{=}e_1\,\texttt{in}\;e_2\;\;\texttt{end}]) =$
$(\mathrm{FV}(e_1)\setminus\{f,x\}) \cup (\mathrm{FV}(e_2)\setminus\{f\}) \cup \mathrm{LV}(L)$

Intuitively, a variable is live at a hole in a local context, if the variable is in scope at the hole and is used by the computation up to end of the context. In (22), v can be a lambda abstraction; the free variables of v are considered live at the allocation point, since they must be put into the closure for v after memory for the closure has been allocated. In (23), the set of live variables is just $\mathrm{LV}(L)$, since the storage mode which is passed to f indicates whether the region contains values that are used after f returns. In (24), however, f is considered live at the allocation point: at runtime, first space for the closure is allocated and then the closure is created by appling f to the actual regions.) Similarly, in the case for `letrec`, f is not considered live at the hole, since the space for the region closure representing f is allocated before the closure is created.

Let e be an expression in K-normal form. For simplicity, we assume that e has no free program variables, that all bound program variables are distinct and that every region-polymorphic function has at precisely one region parameter. (The generalisation to many region parameters is straightforward.) Let x be a program variable which occurs in e. Let (T,ρ) be the type annotation of the binding occurrence of x, where T takes one of the forms τ, σ or π, depending on how x is bound (see the definition of K-normal forms). We define the *live region variables of x*, written $lrv(x)$, to be the set $\{\langle\rho\rangle \mid \rho \in \mathrm{frv}(T,\rho)\} \cup \{\langle\epsilon\rangle \cap \mathrm{RegVar} \mid \epsilon \in \mathrm{fev}(T)\}$. Next, when X is a set of variables occurring in e we define $lrv(X) = \bigcup\{lrv(x) \mid x \in X\}$. Let C be an allocation context of the form $E[R]$. We say that a region variable ρ is *bound non-locally in C*, if C cannot be written in the form $E'[\texttt{letregion}\;\rho:m\;\;\texttt{in}\;\;R'\texttt{end}]$ or $E'[\texttt{letrec}\;\;f:(\pi,\rho_0)\texttt{[}\rho:m\texttt{]}\,\texttt{(}x\texttt{)}\,a\texttt{=}R'\;\;\texttt{in}\;\;e_2\;\;\texttt{end}]$ for any E' and R'. (In other words, ρ is bound non-locally in C, if there is an incomplete λ or `letrec` between the binder of ρ and the hole of the context.) The following rules make it possible to change every $\texttt{at}\rho$ occurring in e into $\texttt{attop}\,\rho$, $\texttt{atbot}\,\rho$ or $\texttt{sat}\,\rho$.

$$\frac{\rho \notin lrv(\mathrm{LV}(R))}{\begin{array}{c}E[\texttt{letregion}\;\rho:m\;\;\texttt{in}\;\;R[\texttt{at}\;\;\rho]\;\;\texttt{end}] \Rightarrow\\ E[\texttt{letregion}\;\rho:m\;\;\texttt{in}\;\;R[\texttt{atbot}\;\;\rho]\;\;\texttt{end}]\end{array}} \tag{25}$$

$$\frac{\rho \in lrv(\mathrm{LV}(R))}{\begin{array}{c}E[\texttt{letregion}\;\rho:m\;\;\texttt{in}\;\;R[\texttt{at}\;\;\rho]\;\;\texttt{end}] \Rightarrow\\ E[\texttt{letregion}\;\rho:m\;\;\texttt{in}\;\;R[\texttt{attop}\;\;\rho]\;\;\texttt{end}]\end{array}} \tag{26}$$

$$\frac{lrv(\mathrm{LV}(R)) \cap \langle\rho\rangle = \emptyset}{\begin{array}{c}E[\texttt{letrec}\;\;f\texttt{[}\rho:m\texttt{]}\,\texttt{(}x\texttt{)}\,a\texttt{=}R[\texttt{at}\;\;\rho]\;\;\texttt{in}\;\;e_2\;\;\texttt{end}] \Rightarrow\\ E[\texttt{letrec}\;\;f\texttt{[}\rho:m\texttt{]}\,\texttt{(}x\texttt{)}\,a\texttt{=}R[\texttt{sat}\;\;\rho]\;\;\texttt{in}\;\;e_2\;\;\texttt{end}]\end{array}} \tag{27}$$

$$\frac{lrv(\mathrm{LV}(R)) \cap \langle\rho\rangle \neq \emptyset}{\begin{array}{c}E[\texttt{letrec}\;\;f\texttt{[}\rho:m\texttt{]}\,\texttt{(}x\texttt{)}\,a\texttt{=}R[\texttt{at}\;\;\rho]\;\;\texttt{in}\;\;e_2\;\;\texttt{end}] \Rightarrow\\ E[\texttt{letrec}\;\;f\texttt{[}\rho:m\texttt{]}\,\texttt{(}x\texttt{)}\,a\texttt{=}R[\texttt{attop}\;\;\rho]\;\;\texttt{in}\;\;e_2\;\;\texttt{end}]\end{array}} \tag{28}$$

```
letregion ρ6:1 in
 letrec f[ρ7:1,ρ8:1,ρ9:∞,
          ρ12:1,ρ13:0](x) atbot ρ6 =
  letrec facacc[ρ16:∞](p) sat ρ7 =
    let n= fst p in let acc= snd p in
       if n = 0 then p
       else facacc[sat ρ16]
              ((n-1,n*acc) sat ρ16)
    end end
  in
(*1*) ((λy.facacc[attop ρ13]y)sat ρ12,
(*2*)  facacc[sat ρ9]((x + 3, 1)sat ρ9)
      ) at ρ8
  end
 in letregion ρ28:∞,ρ31:1,ρ32:1 in
      (letregion ρ33:1,ρ34:∞ in
        fst (f [atbot ρ31,atbot ρ33,
(*3*)           atbot ρ34,atbot ρ32,
                atbot ρ28] 7)
(*4*)  end)((8,1) attop ρ28)
    end
 end
end
```

Figure 4: After storage mode analysis

$$\frac{\rho\text{ bound non-locally in }E[R]}{E[R[\texttt{at}\,\rho]] \Rightarrow E[R[\texttt{attop}\,\rho]]} \tag{29}$$

For brevity, we have shortened $f:(\pi,\rho_0)$ to f in (27) and (28). In (25), `atbot` is justified by the fact that no value which is used up to the point where the region is de-allocated resides in ρ. (Here it is essential that R is a *local* context.) In (27), `sat` is justified by the fact that neither ρ nor any region with which it may be aliased contains a value which is needed by the rest of the body of f. Finally, in (29), we conservatively use `attop`, if ρ is bound outside the closest surrounding function.

Rules 25–29 have been implemented and tested in the Kit, but not proved correct.

Figure 4 shows the result of applying storage mode analysis to the expression in Figure 3. At line `(*1*)` notice that we get `attop` ρ_{13}, by (29). Thus pairs will pile up in ρ_{28} during the evaluation of the application in line `(*4*)`. By contrast, we get `sat` ρ_9 in line `(*2*)`, by (27). In line `(*3*)`, ρ_{34} is passed as region actual corresponding to ρ_9. This happens with mode `atbot`, using (25), so that this "infinite" region ρ_{34} will only ever hold one pair.

7 Physical Size Inference

At every value allocation $v@\rho$ (where $@ \in \{\texttt{attop},\texttt{atbot},\texttt{sat}\}$), the size of the value can be computed statically. (Every function is represented by a "flat" closure which contains the values of the free variables of the function.) Also, the multiplicity of the region is known. In case the multiplicity is finite, the physical size of the region is to be the maximum size of values that may be stored at ρ or at any region variable with which ρ can be aliased. This maximum can be found using the graph G computed in Section 6.

8 The Kit Abstract Machine

The KAM has a *runtime stack*, an infinite number of *registers* and a *region heap*. The operations of the KAM are similar to those of Appel[2], extended with operations for allocating and deallocating regions and for allocating memory in regions. Region names (Section 4.1) are represented as 32 bit words, with the two low order bits being used for storing the region size (finite/infinite) and the storage mode (attop/atbot). The KAM has operations for setting and testing these bits. The region operations are implemented by a runtime system written in C.

A region of unbounded size is represented by a linked list of fixed-size blocks of contiguous memory in the region heap. Regions with finite size are implemented on the runtime stack. That is, upon evaluating `letregion` $\rho : k$ `in` e `end`, where k is a finite physical size, the variable ρ is bound to the current stack pointer which is then increased with k words. Then e is evaluated and the stack pointer decreased by k words.

9 Experimental Results

The purposes of the experiments were (a) to assess the feasibility of region-based execution by comparing the time and space requirements of object programs produced by the Kit to time and space requirements of object programs produced by a first-rate ML compiler, namely Standard ML of New Jersey, and (b) to assess the importance of multiplicity inference and storage mode analysis.

The benchmarks fall into two categories: (1) small programs designed to exhibit extreme behaviour (`fib`, `reynolds2`, `reynolds3`, `dangle` and `tailloop`); and (2) non-trivial programs based on the Standard ML of New Jersey distribution benchmarks (`life`, `mandelbrot`, `knuth-bendix` and `simple`); the largest benchmark is `simple` (approx. 1150 lines of SML). The smallest benchmarks are shown in Section 9.3. In tables, we separate small benchmarks from other benchmarks by a horisontal line.

All benchmarks were executed as stand-alone programs under the ML Kit (using the PA-RISC code generator) and Standard ML of New Jersey[3], version 93 on an HP PA-RISC 9000s700 computer. All running times are in seconds (user time, measured by the UNIX `time` program). Space is maximum resident memory in kilobytes (measured by the UNIX `top` program).

9.1 Comparison with Standard ML of New Jersey

The numbers presented here must be read with caution, since the two compilers are very different. However, the numbers do give a rough indication of the feasibility of region-based execution.

Figure 5 shows a comparison of space usage. There can be dramatic differences between using region inference and using a (reference tracing) garbage collector. These differences will be explained in Section 9.3.

Figure 6 shows running times in seconds, still on the HP PA-RISC 9000s700. The numbers are Unix "user time". The relatively poor performance of the Kit on `simple` is probably due to the fact that this benchmark makes intensive use of floating point numbers, which are implemented very inefficiently in the Kit.

	Kit, s	**NJ93,** s_{93}	$\frac{s*100\%}{s_{93}}$
`life`	376	1,952	24%
`mandelbrot`	352	852	41%
`knuth-bendix`	4,000	2,300	174%
`simple`	2,100	2,200	95%
`fib`	92	1,000	9%
`reynolds2`	96	1,212	8%
`reynolds3`	40,000	1,204	3322%
`dangle`	224	45,000	0.5%
`tailloop`	96	880	11 %

Figure 5: Comparison of space between the ML Kit and SML/NJ version 93. All numbers are in kilobytes and indicate maximum resident memory used.

	Kit, t	**NJ93,** t_{93}	$\frac{t*100\%}{t_{93}}$
`life`	14.2	12.3	115.4 %
`mandelbrot`	43.4	24.9	174.3 %
`knuth-bendix`	32.4	27.8	116.5 %
`simple`	62.2	17.4	357.5 %
`fib`	10.8	27.9	38.7 %
`reynolds2`	16.7	29.2	57.2%
`reynolds3`	23.8	27.7	85.9%
`dangle`	1.56	14.4	10.8%
`tailloop`	4.79	1.96	244 %

Figure 6: Comparison of running times (in seconds)

Considering that the Kit compiles programs very naïvely, apart from everything that has to do with regions, it appears that neither the extra cost associated with allocating into multiple regions nor the overhead of runtime region parameters are prohibitive in practice.

9.2 Region Representation Inference

Figure 7 summarises the static results of region representation inference. In all the benchmarks, except `tailloop`, at least three out of four region variables were found not to belong on the region heap.

	letregions	**word**	**stack**	**heap**
`life`	469	23%	56 %	20%
`mandelbrot`	112	27%	58 %	14%
`knuth bendix`	1014	17%	66 %	16%
`simple`	2648	21%	66 %	11%
`fib`	14	72%	28%	0%
`reynolds2`	85	21%	58 %	20%
`reynolds3`	85	20%	57 %	22%
`dangle`	13	38%	46 %	15%
`tailloop`	9	0%	66%	33%

Figure 7: For each program, the table shows how many letregion binders the region-annotated program contains, and the partitioning of these according to how they will be allocated at runtime.

Figure 8 shows the distribution of allocations amongst stack and heap at runtime. In all cases, except `dangle`, at least 85% of allocations were stack allocations. Remarkably, the largest of the programs, `simple`, had more than 99% of all allocations happen on the stack. The difference between the number of heap allocations for reynolds2 and reynolds3 shows that the static frequency of infinite letregions is not necessarily a good indication of dynamic behaviour (compare Figure 7). Notice that although reynolds3 "space leaks" in the Kit, the space leak is on the heap and the vast majority of allocations are still stack allocations and cause no space problems. This fits with our general experience that space leaks with region inference tend to stem from few isolated spots in the program. (This experience is based on the fact that we have built a region profiler which can trace region sizes.)

To assess the importance of multiplicity inference, the benchmarks were also compiled and run on a version of the Kit in which all multiplicities were set to infinity (while all other analyses were left enabled), see Figure 9. For all the benchmarks, multiplicity inference gives speedups of more than 200%: allocation into a region of finite multiplicity is cheaper than allocation into a region of unbounded multiplicity. Multiplicty inference does not always yield big space savings; it depends on whether many regions exist at the same time.

To assess the importance of storage mode analysis, the benchmarks were then compiled and run on a version of the Kit in which all storage modes were selected to attop (while all other analyses were left enabled), see Figure 10. With storage mode analysis enabled, tailloop runs in constant space, but without storage mode analysis, a memory overflow occurs. For life, the storage mode analysis ensures that at most two generations of the game are alive at the same time. (Without storage mode analysis, all generations pile up in the same regions.) That there are many cases where storage mode analysis does not bring down the maximal space usage is not surprising: maximal space usage is not necessarily reached by the kind of iterative computations for which storage mode analysis is intended.

Multiplicity inference appears to give significant time savings, across all benchmarks. Storage mode analysis is more erratic: it serves an important purpose for some "iterative" computations, but these do not necessarily dominate overall space usage.

Judging from the very high proportion of allocations that happen on the stack in the Kit, optimisations that move stackable regions into registers could be very important. The most obvious candidate is to allow more than one function argument register and more than one function result register. Also, the Kit (quite unnecessarily) represents every function by a closure, even when all the call sites are known. Finally, improved in-lining might help. The Kit evaluates a comparison like i=0 by building a tagged tuple, passing it to the equality function of the prelude, which takes apart the tuple, calls the polymorphic equality function in the runtime system, which eventually returns an integer, which is then compared against an integer, resulting in a branch and store in a register. We believe that this could be improved.

	stack allocations, S	**heap allocations**, H	$\frac{S*100\%}{H+S}$
life	28,269,922	2,184,329	93%
mandelbrot	158,376,021	340	> 99.9%
knuth bendix	51,962,334	8,684,852	86%
simple	96,903,575	728,713	99.2%
fib	1	0	100%
reynolds2	25,165,846	91	> 99.9%
reynolds3	42,991,640	4,194,393	91%
dangle	2,002,002	4,007,008	33%
tailloop	4,004,004	4,004,006	50%

Figure 8: For each program, the table shows how many allocations of objects were done in total at runtime (not including objects of runtime type word) and the partitioning of these according to whether they were done on the stack or the heap.

	Space, s_∞	$\frac{s_\infty*100\%}{s}$	**Time**, t_∞	$\frac{t_\infty*100\%}{t}$
life	548	138%	50.4	355%
mandelbrot	9,988	2,837%	223	514%
knuth-bendix	6,612	165%	77.9	240%
simple	3,860	184%	296	476%
fib	116	129%	32.4	300%
reynolds2	120	125%	50.4	301%
reynolds3	40,000	100%	86.8	365%
dangle	1,732	687%	4.83	310%
tailloop	96	100%	10.2	208%

Figure 9: Space and time used in the Kit when all multiplicities are set to ∞. The numbers are compared with the results from Figures 5 and 6.

	Space, $s_\top$	$\frac{s_\top*100\%}{s}$	**Time**, $t_\top$	$\frac{t_\top*100\%}{t}$
life	768	204%	14.8	104%
mandelbrot	352	100%	45.9	106%
knuth-bendix	4,620	116%	38.3	118%
simple	2,112	101%	76.2	123%
fib	92	100%	11.4	105%
reynolds2	96	100%	18.5	111%
reynolds3	40,000	100%	25.2	106%
dangle	240	107%	1.70	109%
tailloop	-	-	-	-

Figure 10: Space and time used in the Kit when all storage modes are set to attop. The numbers are compared with the results from Figures 5 and 6. tailloop crashes with memory overflow.

9.3 Discussion of extreme behaviour

In this section we analyse some of the small benchmarks, which were designed to exhibit extreme behaviour. Here is reynolds2:

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

fun mk_tree 0 = Lf
  | mk_tree n = let val t = mk_tree(n-1)
                in Br(n,t,t)
                end

fun search p Lf = false
  | search p (Br(x,t1,t2)) =
      if p x then true
      else search (fn y => y=x orelse p y) t1
           orelse
           search (fn y => y=x orelse p y) t2
```

```
val it = search (fn _ => false) (mk_tree 20)
```

The program `reynolds3` is obtained by replacing the `search` function of `reynolds2` by:

```
fun member(x,[]) = false
  | member(x,x'::rest) =
      x=x' orelse member(x, rest)

fun search p Lf = false
  | search p (Br(x,t1,t2)) =
      if member(x,p) then true
      else search (x::p) t1 orelse
           search (x::p) t2
```

Irrespective of whether region inference or garbage collection is used, the running time is exponential in n, where n is the argument to `mk_tree`. (n is 20 in the example.) In `reynolds2`, the polymorphic recursion of region inference separates the lifetimes of `p` and `(fn y => y=x orelse p y)`. In `reynolds3`, however, `p` and `x::p` are put in the same region, for region inference does not distinguish between a list and its tail. With region inference, space consumption is linear in running time with `reynolds3` and logarithmic in running time with `reynolds2`. With garbage collection, it is logarithmic in both cases.

Here is `dangle`:

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

fun cycle(p as (m,f)) =
  if m=0 then p
  else cycle(m-1,
         let val x = [(m, mklist 2000)]
         in fn () => #1(hd x) + f()
         end)

val r = cycle(1000, fn() => 0);
```

Region inference ensures that the list l produced by `mklist 2000` is discarded immediately after the closure for `fn () => #1(hd x) + f()` is produced; note that the function will not access l — in fact the closure will contain a dangling pointer[17]. In garbage collected systems which do not allow dangling pointers, the space usage is $O(m \times n)$, where m and n are the arguments to `cycle` and `mklist`, respectively (here $m = 1000$ and $n = 2000$). With region inference, the space usage is just $O(m)$.

Finally, here is `tailloop`:

```
val x =
let
  val maxint = 2000
  val zero = (0,0)
  fun is_zero(0,0) = true
    | is_zero _ = false
  fun sub (m,n) =
      if n=0 then (m-1, maxint)
      else (m, n-1)
  fun loop (x as (m,n)) =
      if is_zero x then x
      else loop(sub x)
  fun loop' p = (loop p;
                 "\ndone\n")
in
  output(std_out,
         "\nlooping...\n");
  output(std_out,
         loop'(maxint,maxint))
end;
```

Integers themselves are unboxed, but without storage mode analysis, the integer pairs fill up the memory.

10 Conclusion

We have presented a series of region-based analyses for mapping an abstract stack of regions onto real machines. All of these analyses were devised to solve needs which became evident from practical experiments. The combination of analyses presented here often works well in practice, but we have also shown examples which suggest that it might be useful to provide garbage collection as a supplement to region inference, to handle those cases where the various static analyses cannot cope. (Such cases will always exist, for undecidablity reasons.) It is noteworthy, however, that all the benchmarks we tried from the SML/NJ test suite could be made to run relatively well, even without garbage collection and without many of the optimisations one expects to find in a mature compiler.

Acknowledgements

We wish to thank Martin Elsman and Niels Hallenberg for their work on the Kit, Raph Levien for finding mistakes in earlier versions of the storage mode analysis and Greg Morrisett for good advice on code generation. This work is funded by the Danish National Research Council, in the form of a Ph.D. scolarship for the first author and the DART grant for the second author.

References

[1] Alexander Aiken, Manuel Fähndrich, 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.

[2] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[3] Andrew W. Appel and David B. MacQueen. A Standard ML compiler. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture.* ACM, Springer-Verlag, Sept 1987.

[4] 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.

[5] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proc. 9th Annual ACM Symp. on Principles of Programming Languages*, pages 207–212, Jan. 1982.

[6] 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.

[7] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proc. of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation (PLDI)*, June 1993.

[8] Fritz Henglein and Jesper Jørgensen. Formally optimal boxing. In *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 213–226. ACM Press, January 1994.

[9] 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.

[10] Xavier Leroy. Unboxed objects and polymorphic typing. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 177–188. ACM Press, January 1992.

[11] J. M. Lucassen. *Types and Effects, towards the integration of functional and imperative programming.* PhD thesis, MIT Laboratory for Computer Science, 1987. MIT/LCS/TR-408.

[12] J.M. Lucassen and D.K. Gifford. Polymorphic effect systems. In *Proceedings of the 1988 ACM Conference on Principles of Programming Languages*, 1988.

[13] R. Milner. A theory of type polymorphism in programming. *J. Computer and System Sciences*, 17:348–375, 1978.

[14] Hanne Riis Nielson and Flemming Nielson. Higher-order concurrent programs with finite communication topology. In *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97. ACM Press, January 1994.

[15] Zhong Shao. *Compiling Standard ML for Efficient Execution on Modern Machines.* PhD thesis, Princeton University, 1994. (Also available as Research Report CS-TR-475-94).

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

[17] 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.

[18] Magnus Vejlstrup. Multiplicity inference. Master's thesis, Dept. of Computer Science, Univ. of Copenhagen, September 1994.