From Region Inference to von Neumann Machines via Region Representation Inference

Lars Birkedal, Carnegie Mellon University* Mads Tofte, University of Copenhagen† Magnus Veijstrup, 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 prototype 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:

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].† 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

* 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_Æ_NKT_ELEKTRONIK@dce.dk.

† 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 role 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:

\[
e ::= true \mid false \mid x \mid \lambda x.e \mid e_1 e_2
\]

[\[
\text{if } e \text{ then } e \text{ else } e\]

[\[
\text{let } x = e \text{ in } e \text{ end}
\]

letrec \(f(x) = e\) in \(e\) end

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:

\[
\text{letrec } f(x) =
\]

\[
\text{letrec } \text{facacc}(p) =
\]

\[
\text{let } n = f! p \text{ in let } acc = \text{ snd } p
\]

\[
in \text{ if } n=0 \text{ then } p \text{ else } \text{facacc}(n-1,n*acc)
\]

\[
\text{end end}
\]

\[
in (\lambda y.\text{facacc } y, \text{ facacc}(x+3,1))
\]

\[
\text{end end}
\]

Here we have taken the liberty to extend the skeletal language with pairs, projections (\(f!\) and \(\text{ snd}\)), integer constants, and infix binary operations on integers (+, *, -, *).

Also, we use parent heses 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 denumerable infinite set of region variables, ranged over by \(\rho\). For any syntactic class, \(c\), let \(\bar{c}\) denote the syntactic class defined by:

\[
\bar{c} ::= \text{ empty} \mid c \mid c_1 \cdots c_n \quad (n \geq 2)
\]

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

\[
a ::= \text{ at } \rho \quad b ::= \rho
\]
\[
\begin{align*}
e & := \text{true} \mid \text{false} \mid x \mid (\lambda x.e) a \mid e_1 e_2 \\
& \quad \text{if } e \text{ then } e \text{ else } e \\
& \quad \text{let } x = e \text{ in } e \\
& \quad \text{letregion } b \text{ in } e \\
& \quad \text{letrec } f \{ \hat{a} \} (x) a = e \text{ in } e \\
& \quad f \{ \hat{a} \} a
\end{align*}
\]

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 \text{letrec } \rho \text{ in } e \text{ end} is evaluated thus: first a region is allocated and bound to \rho; then \( e \) is evaluated (probably using the region for storing and retrieving values) and then, when \text{end} is reached, the region is deallocated. An annotation of the form \( \rho \) indicates that the value of the expression preceding the annotation should be put into the region bound to \rho. 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 \text{letrec} is region-poly-morphic: it has a (perhaps empty) list of formal region parameters and may be applied to different actual regions at different call sites. An expression \( f \{ \hat{a} \} a \) creates a function closure in region \( \rho \), in which the formals of \( f \) have been bound to the actual regions \( \hat{a} \).

We write \text{letrec } \rho^2 \text{ in } e \text{ end for}
\[
\text{letrec } \rho_1 \\
\quad \text{in } \cdots \text{letrec } \rho_k \text{ in } e \text{ end } \\
\text{end}
\]
when \( \rho^2 = \rho_1, \ldots, \rho_k \). Further, \( f[\rho_1, \ldots, \rho_k] \ a \) abbreviates \( f[\text{atap}_1, \ldots, \text{atap}_k] \ a \). Expressions of the form
\[
\text{letrec } \rho \text{ in } f[\rho_1, \ldots, \rho_k] \text{ at } \rho \text{ (e) end}
\]
(when \( \rho \notin \{ \rho_1, \ldots, \rho_k \} \) and \( \rho \) 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 Leven[e][1] have developed an analysis which seeks to do the actual region allocation in \text{letrec } \rho \text{ in } e \text{ end} as late as possible after the \text{letrec} 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 \text{letrec} and de-allocation is done at \text{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 \( \text{TyVar} \) of \text{type variables}, ranged over by \( \alpha \), and a denumerably infinite set \( \text{EffectVar} \) of \text{effect variables}, ranged over by \( \epsilon \).

An effect, \( \varphi \), is a finite set of atomic effects. An atomic effect, \( \eta \), is either a token of the form \text{get}(\rho) or \text{put}(\rho), or \( \epsilon \) is an effect variable. Types, \( \tau \), \text{types and places}, \( \mu \), \text{simple type schemes}, \( \sigma \), \text{compound type schemes}, \( \pi \), take the form:
\[
\begin{align*}
\tau & ::= \text{bool} \mid \alpha \mid \mu \rightarrow \tau^2 \rightarrow \\
\mu & ::= (\tau, \rho) \\
\sigma & ::= \tau \mid \forall \alpha. \sigma \mid \forall \epsilon. \sigma \\
\pi & ::= \sum \mid \forall \alpha. \pi \mid \forall \epsilon. \pi \mid \forall \rho. \pi
\end{align*}
\]

An object of the form \( (e, \varphi) \) (formally a pair \((e, \varphi)\)) on a function arrow \( \mu \rightarrow \tau^2 \rightarrow \mu' \) is called an \text{arrow effect}. Here \( \varphi \) is the effect of evaluating the body of the function.

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

A \text{type environment}, \( TE \), is a finite map from program variables to pairs of the form \((\sigma, \rho)\) or \((\pi, \rho)\).

A \text{substitution} \( S \) is a triple \((S', S', S')\), where \( S' \) is a finite map from region variables to region variables, \( S' \) is a finite map from type variables to types and \( S' \) 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
\[
\pi = \forall \rho_1 \cdots \rho_k \alpha_1 \cdots \alpha_n \epsilon_1 \cdots \epsilon_m. \sum
\]
and type $\tau'$, we say that $\tau'$ is an instance of $\pi$ (via $S$), written $\pi \geq \tau'$, if there exists a substitution

$$S = \{ \rho_1 \mapsto \rho'_1, \ldots, \rho_k \mapsto \rho'_k \},$$

$$\{ \alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n \},$$

$$\{ \epsilon_1 \mapsto \epsilon'_1, \ldots, \epsilon_m \mapsto \epsilon'_m, \epsilon_m \}$$

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

$$([\rho_1', \ldots, \rho_k'], [\tau_1, \ldots, \tau_n], [\epsilon'_1, \ldots, \epsilon'_m, \epsilon_m]).$$

More generally, we refer to triples of above form as instance lists and use $\text{id}$ 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 $\text{TE} \vdash e : \mu, \phi$. Formally, an explicitly typed region-annotated term is a term $e$, for which there exist $\mu$ and $\phi$ such that $\text{TE} \vdash e : \mu, \phi$. For given $\text{TE}$ and $e$ there is at most one such $\mu$ and $\phi$ (and at most one derivation proving $\text{TE} \vdash e : \mu, \phi$). 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-Annotted Terms

$$\text{TE} \vdash e : \mu, \phi$$

1. $\text{TE} \vdash \text{true at } \rho : (\text{bool}, \rho), \{\text{put}(\rho)\}$
2. $\text{TE} \vdash \text{false at } \rho : (\text{bool}, \rho), \{\text{put}(\rho)\}$
3. $\text{TE} \vdash \text{x} : \sigma$ via $S$
4. $\text{TE} \vdash \{x \mapsto \mu_1\} \vdash e : \mu_2, \phi \subseteq \phi'$
5. $\text{TE} \vdash (\lambda^x. e : (\mu, \phi)) \at \rho : (\mu_1 \xrightarrow{x = \mu'} \mu_2, \rho), \{\text{put}(\rho)\}$
6. $\text{TE} \vdash e_1 : (\mu', \phi_2)$
7. $\text{TE} \vdash e_2 : \mu, \phi_2$
8. $\text{TE} \vdash e_3 : \mu, \phi_2$
9. $\text{TE} \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \mu, \{\text{get}(\rho)\} \cup \phi_1 \cup \phi_1 \cup \phi_2$
10. $\text{TE} \vdash \text{let } x : (\sigma, \rho_1) = e_1 \text{ in } e_2 : \mu, \phi_1 \cup \phi_2$

### 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 $\text{max}(m_1, m_2)$, and the product of $m_1$ and $m_2$, written $m_1 \otimes m_2$, are defined similarly.

#### 4.1 Untyped Multiplicity-Annnotated Terms

We modify the class of region binders to become:

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

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

$$\text{letrec } f : (\pi, \rho)(x) = e_1 \text{ in } e_2 : \mu, \phi_1 \cup \phi_2$$

the multiplicity $m$ is an upper bound on the number of times a value is put into the region which will be bound to $\rho$ 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

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

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 \( \rho \) — including calls that \( f \) may make to other functions or to itself. Consider a reference to \( f \) (in \( e_1 \) or in \( e_2 \))

\[ \ldots \; f[s, \ldots, \rho', \ldots ] \; \ldots \]

It is possible to have \( \text{mult}(\rho) < \infty \) and \( \text{mult}(\rho') = \infty \), 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 \( f \):

\[ \ldots \; f[s, \ldots, \rho'', \ldots ] \; \ldots \]

we need not have \( \text{mult}(\rho'') = \text{mult}(\rho') \). This flexibility was found to be important in practice — without it, too many regions were ascribed multiplicity \( \infty \). 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 \text{letrec} expression which generates it and it never changes, so \((r, m)\) can be regarded as a region \( r \) with a multiplicity attribute \( m \).

When storing a value into a \text{letrec-bound} \( \rho \) 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 \textit{multiplicity effect} is a finite map from atomic effects to multiplicities; we use \( \psi \) to range over multiplicity effects. The extension to a total map which is \( \psi \) outside the domain of \( \psi \) is denoted \( \psi^+ \). Let \( \psi_1 \) and \( \psi_2 \) be multiplicity effects. The \textit{sum} of \( \psi_1 \) and \( \psi_2 \), written \( \psi_1 \oplus \psi_2 \), is the multiplicity effect which has domain \( \text{Dom}(\psi_1) \cup \text{Dom}(\psi_2) \) and values

\[ (\psi_1 \oplus \psi_2)(\eta) = \psi_1^+(\eta) \oplus \psi_2^+(\eta) \]

Similarly, the \textit{maximum} of \( \psi_1 \) and \( \psi_2 \), written \( \text{max}(\psi_1, \psi_2) \), is defined by

\[ (\text{max}(\psi_1, \psi_2))(\eta) = \text{max}(\psi_1^+(\eta), \psi_2^+(\eta)) \]

Finally, when \( \psi \) is a multiplicity effect and \( m \) is a multiplicity, the \textit{scalar product}, \( m \odot \psi \), is the multiplicity effect with the same domain as \( \psi \) and values \( (m \odot \psi)(\eta) = m \odot (\psi(\eta)) \).

The semantic objects of \textit{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 \( \tau, S \) 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 \( \eta \in \text{Dom}(\psi) \).

The typing rules for multiplicity-annotated terms are:

\textbf{Multiplicity-Annotated Terms} \[ TE \vdash e : \mu, \psi \]

\begin{align*}
TE & \vdash \text{true} \; \rho : \text{bool}, \{ \text{put}(\rho) \mapsto 1 \} \\
TE & \vdash \text{false} \; \rho : \text{bool}, \{ \text{put}(\rho) \mapsto 1 \} \\
TE & \vdash f[s, \ldots, \rho: m, \ldots ](x) a = e_1 \text{ in } e_2 \; \{ \}
\end{align*}

\[ TE + \{ x \mapsto \mu_1 \} \vdash e : \mu_2, \psi' \; \psi \odot \psi'' = \psi' \]

\[ TE \vdash (\lambda^\psi x : \mu_1, e) \; \rho : \mu_1 \mapsto \psi \mapsto \psi_2, \psi \; \{ \} \]

\begin{align*}
TE & \vdash e_1 : \mu_1, \psi_1 \\
TE & \vdash e_2 : \mu_2, \psi_2 \\
TE & \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \\
& \quad : \mu, \{ \text{get}(\rho) \mapsto 1 \} \odot \psi_1 \odot \text{max}(\psi_2, \psi_3) \]
\end{align*}

\[ TE \vdash \text{letrec } f[s, \ldots, \rho: m, \ldots ](x) a = e_1 \text{ in } e_2 \text{ end } : \mu, \psi_1 \odot \psi_2 \]

\[ TE \vdash f[s, \ldots, \rho: m, \ldots ](x) \; \rho : \tau, \psi \]

\[ TE \vdash e : \mu, \psi \; \text{fv}(\psi') \cap \text{fv}(TE, \mu) = \emptyset \]

\[ TE \vdash \text{letrec } \psi' \text{ in } e' \text{ end } : \mu, \psi \sqcup \psi' \]

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 \textit{multiplicity substitution} is a triple \( S = (S', S^0, S^\\infty) \), where \( S^0 \) is a map from type variables to types, \( S^\infty \) is a map from region variables to region variables and \( S' \) is a map from effect variables to multiplicity arrow effects \( e, \psi \). Each of these finite maps extend to total maps — in the case of \( S^\infty \) by mapping each effect variable \( e \) outside the domain of \( S^\infty \) to the multiplicity arrow effect \( e \).
We define

\[ S'(\psi) = \bigoplus \{ \text{put}(S'(\rho)) \mapsto \psi(\text{put}(\rho)) \mid \text{put}(\rho) \in \mathcal{P}\} \]
\[ \oplus \bigoplus \{ \text{get}(S'(\rho)) \mapsto \psi(\text{get}(\rho)) \mid \text{get}(\rho) \in \mathcal{P}\} \]
\[ \oplus \psi \downarrow \text{Effect Var} \]

\[ S''(\psi) = \psi \setminus \text{Effect Var} \oplus \]
\[ \bigoplus \{ \psi(e) \oplus \{ e' \mapsto 1 \} \oplus \psi' \mid e \in \text{Dom}(\psi) \land e'.\psi' = S''(e) \} \]

Moreover, define

\[ S'(e, \psi) = e'.(\psi' \oplus S''(\psi)) \]

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

\[ (S', S', S')(A) = S'(S''(S'(A))) \]

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 \((\sigma, \rho)\) or \((\pi, \rho)\) in the range of \( TE \).

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

### Multiplicity Inference

**Inference Problem:** given \( TE, e, \mu \) and \( \varphi \) 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 \( \infty \). The object is of course to choose multiplicities as small as possible.

Veijrlup'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 \( S''(\max(\psi_1, \psi_2)) \geq \max(S''(\psi_1), S''(\psi_2)) \). In particular, if a lambda-bound variable, \( f \), occurs in two different conditions, unification on the type of \( f \) during the multiplicity inference of the second condition can increase the effect of the first condition:

\[ \lambda f : ((\text{int}, \rho_1) \rightarrow (\text{int}, \rho_2), \rho_3), \]
\[ \text{let } x = \text{if true then } 1 \text{ at } \rho_2 \text{ else } f(1) \text{ at } \rho_1 \]
\[ \text{in if true then } (\lambda y) \text{ at } \rho_2 \text{ else } f \]

Here the effect of evaluating \( x \) will end up having two put effects on \( \rho_2 \), 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:

\[ \text{er} \text{letreg} \psi \text{ in } e \text{ end} = \]
\[ \text{letreg } b_1 \ldots b_k \text{ in } \text{er}(e) \text{ end} \]

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

\[ b_i = \rho_i : \psi^k(\text{put}(\rho_i)) \]

for \( i = 1 \ldots k \).

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

where \( \pi' = \forall \rho_1, \ldots, \rho_k \alpha \varepsilon(\mu_1 \frac{\psi}{\mu_2}) \) and \( b_i = \rho_i : \psi^k(\text{put}(\rho_i)) \), \( i = 1 \ldots k \).

### 4.3 Removal of get-regions

Consider a declaration of the form

\[ \text{letrec } f[B](x) = c \text{ in } e \text{ end} \]

Write \( \tilde{B} \) in the form \( p_1 : m_1, \ldots, p_k : m_k \). If \( p \in \{p_1, \ldots, p_k\} \) is such that there is no \( \text{put}(\rho) \) anywhere in the type of \( f \), then \( f \) does not really need \( \rho \); 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 forms, provided the corresponding actual arguments in calls of \( f \) are removed too.

\[ \text{letreg } p_1 : 1 \text{ in } \text{letrec } f[p_7 : 1, p_9 : \infty, p_10 : \infty, p_11 : \infty, p_12 : 1, p_9 : 0, p_10 : 10](x) \text{ at } p_8 = \]
\[ \text{letrec } f[\text{acc}[p_1 : \infty, p_7 : \infty, p_9 : \infty](p) \text{ at } p_r = \]
\[ \text{let } n = \text{fst } p \text{ in let acc end in } \]
\[ \text{letreg } p_9 : 1 \text{ in } \]
\[ \text{if letreg } p_2 : 1 \text{ in } \]
\[ (n = 0 \text{ at } p_1) \text{ at } p_9 \]
\[ \text{end then } p \]
\[ \text{else fac acc }[p_9 : \infty, p_7 : \infty, p_8 : \infty](p) \text{ at } p_r \]
\[ \text{letreg } p_2 : 1 \text{ in } \]
\[ (n = 1 \text{ at } p_4) \text{ at } p_8 \]
\[ \text{end, } (n * \text{ acc } \text{ at } p_7) \text{ at } p_8 \]
\[ \text{end end end end in } \]
\[ (\lambda y) \text{ fac acc }[p_9 : \infty, p_7 : \infty, p_8 : \infty](y) \text{ at } p_12, (\ast \ast) \]
\[ \text{fac acc }[p_9 : \infty, p_10 : \infty, p_11] \]
\[ \text{letreg } p_7 : 1 \]
\[ (x + 3 \text{ at } p_7) \text{ at } p_11 \]
\[ \text{end, } 1 \text{ at } p_8 \text{ at } p_8 \]
\[ \text{end end end end in } \]
\[ \text{fsl } \text{letreg } p_7 : 1 \text{ in } \]
\[ f[p_3 : 1, p_9 : \infty, p_10 : \infty, p_12 : 1, p_8 : \infty, p_9 : \infty, p_7 : \infty, p_8 : \infty](x) \text{ at } p_7 \]
\[ \text{end end end end } \]
\[ (8 \text{ at } p_7, 1 \text{ at } p_9) \text{ at } p_8 \]

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 ex-
pression 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 $p_0$ removed. The lazy fact accr $y$ in line (*1*) is put into a write-once region ($p_2$), 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 $\text{put} p$ 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 $p$ store word-sized values requires a simple region flow analysis, which we describe in this section. If $p$ is a formal parameter of some letrec-bound function, $f$, and all stores to $p$ are stores of word-sized values, then $p$ 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 $p$ 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

$$a ::= \text{at } p \mid \text{ignore}$$

In examples, we abbreviate $v \text{ ignore}$ to $v$.

In the dynamic semantics, evaluating $v \text{ ignore}$ just results in $v$ without performing any allocation in any region.

5.2 Boxty Constraints

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

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

Ground region types are ordered by $\perp \subseteq \text{word} \subseteq \top$. 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 $p$ for which no at $p$ occurs in the program. (Get-regions of region-polymorphic functions have region type $\perp$, if they are not removed.)

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 $r = \top$ or $r \neq \top$. A finite set of constraints has a minimal solution (with respect to $\subseteq$). 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 $p : m$ is associated with a fresh region type variable, written $r(p)$. For every subexpression $\text{true at } p$ or $\text{false at } p$ of $e$, we generate a constraint $r(p) \supseteq \text{word}$. For all other at $p$ in $e$ we generate a $r(p) \supseteq \top$ constraint. Furthermore, if $f$ is declared by

$$\text{letrec } f[p_1 \ldots p_k : m_b] \text{ at } p = e_1 \text{ in } e_2$$

then for every reference to $f$:

$$f[p_1 \ldots p_k] \text{ at } p$$

we generate the $k$ constraints $r(p_i) \supseteq r(p'_i), 1 \leq i \leq k$.

Once the minimal solution has been found, every region binder $p : m$ 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 $p$ (for the $p$ in question) are changed into ignore. When a formal region type 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 $p_{28}$ and $p_{54}$, 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 di-
rectives and binders
\[ a ::= \text{at} \rho \mid \text{attop} \rho \mid \text{atbot} \rho \mid \text{sat} \rho \]
\[ b ::= \rho : m \]

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

\[ e ::= x_{\rho} \mid v \mid x_{\rho} x_{\rho} \mid f_{\rho}[\alpha] \ a_{0} x_{\rho} \mid \text{if } x_{\rho} \text{ then } e \text{ else } e \mid \text{let } x : (\sigma, \rho) = e \text{ in } e \mid \text{letrec } f : (\pi, \rho_{0})[\beta] (x) a_{0} = e \text{ in } e \mid \text{letregion } b \in e \end{end}

\[ v ::= \text{true} \mid \text{false} \mid \lambda x : \mu. e \]

The key idea is that every intermediate result of the computation is bound to a variable. The type information (\sigma, \pi and \mu) 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 \text{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 \text{letrec}-bound program variable \( f \) with type scheme
\[ \pi = \forall \cdots \rho \cdots, \alpha \cdots, \epsilon_{j} \cdots \]
and whenever there is an applied occurrence of \( f \):
\[ f([\cdot, \cdot \cdot]_{\rho_{1}} \cdots) \]
there is an edge from \( \rho_{1} \) to \( \rho_{1}' \) and from \( \epsilon_{j} \) to \( \epsilon_{j}' \). Similarly for \text{let}-bound variables. Finally, for every effect \( e, \varphi \) occurring anywhere in the program, there is an edge from \( e \) to every region and effect variable which occurs free in \( \varphi \). In the graph that arises thus, \text{letrec} bound variables are always leaf nodes and region variables only lead to region variables. For every node \( n \) in \( G \), let \( (n) \) denote the set of variables that are reachable in \( G \) starting from \( n \), including \( n \) itself.

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

\[ L ::= [] \mid \text{if } x_{\rho} \text{ then } L \text{ else } e_{3} \mid \text{let } x : (\sigma, \rho) = L \text{ in } e_{2} \mid \text{let } x : (\sigma, \rho) = e_{1} \text{ in } L \text{ end} \mid \text{letrec } f : (\pi, \rho_{0})[\beta] (x) a_{0} = e_{1} \text{ in } e_{2} \text{ end} \]

Next, \textit{local allocation contexts}, \( R \), are given by

\[ R ::= L[v][] \mid L[f_{\rho}[\alpha_{1}, \ldots, \alpha_{-1}, []], a_{0} x_{\rho}][] \mid L[f_{\rho}[\alpha]] [x_{\rho}][] \mid L[\text{letrec } f : (\pi, \rho_{0})[\beta] (x) [] = e_{1} \text{ in } e_{2} \text{ end}] \]

The last of the three kinds of context is a \textit{global} expression context, which allows one to single out an arbitrary subexpression:

\[ E ::= L \mid L[\text{let } x : (\sigma, \rho) = \lambda x : \mu. E] a_{0} = E \text{ in } e_{2} \text{ end} \]

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

\[ \text{LV}([]) = \emptyset \]
\[ \text{LV}(\text{if } x_{\rho} \text{ then } L \text{ else } e_{3}) = \text{LV}(L) \]
\[ \text{LV}(\text{if } x_{\rho} \text{ then } e_{2} \text{ else } L) = \text{LV}(L) \]
\[ \text{LV}(\text{let } x : (\sigma, \rho) = L \text{ in } e_{2} \text{ end}) = \text{LV}(L) \cup (\text{LF}(e_{2}) \setminus \{x\}) \]
\[ \text{LV}(\text{let } x : (\sigma, \rho) = e_{1} \text{ in } L \text{ end}) = \text{LV}(L) \]
\[ \text{LV}(\text{letrec } f : (\pi, \rho_{0})[\beta] (x) a_{0} = e_{1} \text{ in } L \text{ end}) = \text{LV}(L) \]
\[ \text{LV}(\text{letregion } b \in L \text{ end}) = \text{LV}(L) \]

Here \text{FV}(e) means the set of program variables that occur free in \( e \).
The definition is extended to local allocation contexts, \( R \), as follows:

\[
L[V][v] = \text{FV}(v) \cup L(V) \tag{22}
\]
\[
L[f_i[a_1,\ldots,a_{i-1},[,a_{i+1},\ldots,a_k] a_i x_id)] = L(V) \tag{23}
\]
\[
L[f_i[a_i]] = \{f,x\} \cup L(V) \tag{24}
\]

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 \( L(V) \), 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 applying \( f \) to the actual regions.) Similarly, in the case for \( \text{letrec} \), \( f \) 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 \( (T,\rho) \) be the type annotation of the binding occurrence of \( x \), where \( T \) takes one of the forms \( \tau, \sigma \lor \pi \), depending on how \( x \) is bound (see the definition of K-normal forms). We define the \( \text{live region variables} \) of \( x \), written \( \text{brv}(x) \), to be the set \( \{(\rho)|\rho \in \text{brv}(T,\rho)\} \cup \{\epsilon\} \cap \text{RegVar} | e \in \text{fev}(T)\} \). Next, when \( X \) is a set of variables occurring in \( e \) we define \( \text{brv}(x) = \bigcup \{\text{brv}(x) | x \in X\} \). Let \( C \) be an allocation context of the form \( E[R] \). We say that a region variable \( \rho \) is \textit{bound non-locally in} \( C \), if \( C \) cannot be written in the form \( E[\text{letrec} \rho \in m \in R]\) or \( E[\text{letrec} \rho \in m \in \text{atop}\rho \in m \in R]\) for any \( E' \) and \( R' \). (In other words, \( \rho \) is bound non-locally in \( C \), if there is an incomplete or incomplete between the binder of \( \rho \) and the hole of the context.) The following rules make it possible to change every \( \text{atp} \rho \) occurring in \( e \) into \( \text{atp} \rho \) or \( \text{atp} \rho \) or \( \text{atp} \rho \).

\[
\begin{array}{c}
\rho \notin \text{brv}(L(V)) \\
\text{E}[\text{letrec} \rho \in m \in R]\Rightarrow \text{E}[\text{letrec} \rho \in m \in R] \tag{25}
\end{array}
\]

\[
\begin{array}{c}
\rho \in \text{brv}(L(V)) \\
\text{E}[\text{letrec} \rho \in m \in R]\Rightarrow \text{E}[\text{letrec} \rho \in m \in R] \tag{26}
\end{array}
\]

\[
\begin{array}{c}
\text{brv}(L(V)) \cap \rho = \emptyset \\
\text{E}[\text{letrec} \{f[p] : m\}x = R]\Rightarrow \text{E}[\text{letrec} \{f[p] : m\}x = R] \tag{27}
\end{array}
\]

\[
\begin{array}{c}
\text{brv}(L(V)) \cap \rho = \emptyset \\
\text{E}[\text{letrec} \{f[p] : m\}x = R]\Rightarrow \text{E}[\text{letrec} \{f[p] : m\}x = R] \tag{28}
\end{array}
\]

Figure 4: After storage mode analysis

\[
\frac{\text{\( \rho \) bound non-locally in \( R \)}}{\text{E}[\text{atp} \rho] \Rightarrow \text{E}[\text{atp} \rho]] \tag{29}
\]

For brevity, we have shortened \( f : (\pi,\rho) \) to \( f \) in (27) and (28). In (25), \( \text{atp} \) is justified by the fact that no value which is used up to the point where the region is de-allocated resides in \( \rho \). (Here it is essential that \( R \) is a local context.) In (27), \( \text{atp} \) is justified by the fact that neither \( \rho \) nor any region with which \( \rho \) may be aliased contains a value which is needed by the rest of the body of \( f \). Finally, in (29), we conservatively use \( \text{atp} \) if \( \rho \) 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 \( *\#* \) notice that we get \( \text{atp} \rho \), by (29). Thus pairs will pile up in \( \rho \) during the evaluation of the application in line \( *\#* \).

By contrast, we get \( \text{sat} \rho \) in line \( *\#* \), by (27). In line \( *\#* \), \( \rho \) is passed as region actual corresponding to \( \rho \). This happens with mode \( \text{atp} \), using (25), so that this "infinite" region \( \rho \) will only ever hold one pair.

7 Physical Size Inference

At every value allocation \( v@p \) (where \( @ \in \{\text{atp}, \text{sat} \} \)), 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.) Thus, 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 \( p \) or at any region variable with which \( p \) 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 low order bits being used for storing the region size (finite/infinite) and the storage mode (atop/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 p : k in e end, where k is a finite physical size, the variable p is bound to the current stack pointer which is then increased by 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 behavior (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 horizontal line.

All benchmarks were executed as stand-alone programs under the MLKit (using the PA-RISC code generator) and Standard ML of New Jersey[3], version 93 on an HP PA-RISC 9000/6700 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 9000/6700. 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.

Considering that the Kit compiles programs very naively, 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 is 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.

<table>
<thead>
<tr>
<th>Life</th>
<th>NJ93</th>
<th>New Jersey</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple</td>
<td>469</td>
<td>56%</td>
</tr>
<tr>
<td>simple</td>
<td>26</td>
<td>21%</td>
</tr>
</tbody>
</table>

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 95% 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 ltrex is not necessarily a good indication of dy-
namic 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 al-
locations 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 multi-
\ifle 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 over-
flow 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.) There are many cases where 
storage mode analysis does not bring down the max-
imal 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 “iter-
\ firefighters, 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

<table>
<thead>
<tr>
<th>stack allocations, $S$</th>
<th>heap allocations, $H$</th>
<th>$S_{100%}$</th>
<th>$H_{100%}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>life</td>
<td>28,359,922</td>
<td>2,184,328</td>
<td>94%</td>
</tr>
<tr>
<td>mandelbrot</td>
<td>158,376,021</td>
<td>340</td>
<td>&gt; 99.9%</td>
</tr>
<tr>
<td>knuth bendix</td>
<td>51,962,334</td>
<td>8,684,832</td>
<td>86%</td>
</tr>
<tr>
<td>simple</td>
<td>96,903,975</td>
<td>728,713</td>
<td>99.2%</td>
</tr>
<tr>
<td>fib</td>
<td>1</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>reynolds2</td>
<td>25,165,846</td>
<td>91</td>
<td>&gt; 99.9%</td>
</tr>
<tr>
<td>reynolds3</td>
<td>42,991,640</td>
<td>4,194,393</td>
<td>91%</td>
</tr>
<tr>
<td>dangle</td>
<td>2,002,002</td>
<td>4,007,008</td>
<td>33%</td>
</tr>
<tr>
<td>tailloop</td>
<td>4,004,004</td>
<td>4,004,006</td>
<td>50%</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Space, $s_T$</th>
<th>Time, $t_T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>life</td>
<td>768</td>
</tr>
<tr>
<td>mandelbrot</td>
<td>352</td>
</tr>
<tr>
<td>knuth bendix</td>
<td>4,620</td>
</tr>
<tr>
<td>simple</td>
<td>2,112</td>
</tr>
<tr>
<td>fib</td>
<td>92</td>
</tr>
<tr>
<td>reynolds2</td>
<td>96</td>
</tr>
<tr>
<td>reynolds3</td>
<td>40,000</td>
</tr>
<tr>
<td>dangle</td>
<td>240</td>
</tr>
<tr>
<td>tailloop</td>
<td>-</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Space, $s_T$</th>
<th>Time, $t_T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>life</td>
<td>768</td>
</tr>
<tr>
<td>mandelbrot</td>
<td>352</td>
</tr>
<tr>
<td>knuth bendix</td>
<td>4,620</td>
</tr>
<tr>
<td>simple</td>
<td>2,112</td>
</tr>
<tr>
<td>fib</td>
<td>92</td>
</tr>
<tr>
<td>reynolds2</td>
<td>96</td>
</tr>
<tr>
<td>reynolds3</td>
<td>40,000</td>
</tr>
<tr>
<td>dangle</td>
<td>240</td>
</tr>
<tr>
<td>tailloop</td>
<td>-</td>
</tr>
</tbody>
</table>

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.

The most obvious candidate is to allow more than one function 
argument register and more than one function result regis-
ter. Also, the Kit (quite unnecessarily) represents every 
function by a closure, even when all the call sites are known.
Finally, improved inlining 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.

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:

data type '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 \( n \)
isa the argument to \( \text{mk\_tree} \). \( (n=20 \text{ in the example) In} \)
reynolds2, the polymorphic recursion of region inference
separates the lifetimes of \( p \text{ and } y=x \text{ or else } p \). In
reynolds3, however, \( p \text{ and } x::p \) 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 \( \text{mk\_list} 2000 \)
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 \( \text{cycle} \) and \( \text{mk\_list} \), 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)

in

output(std_out, "\ndone\n")
in output(std_out, "unlooping...\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 undecidability 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 Leiven 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. scholarship for the first author and the DART
grant for the second author.

References

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.


ML compiler. In Gilles Kahn, editor, Functional
Programming Languages and Computer Architecture.

[4] Lars Birkerdal, 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.

on Principles of Programming Languages, pages 207–


