Title: **Region-Based Memory Management**[1]

Authors: **Mads Tofte and Jean-Pierre Talpin**

Affiliation: **Mads Tofte: Department of Computer Science, University of Copenhagen; Jean-Pierre Talpin: IRISA, Campus de Beaulieu, France**

[1]An earlier version of this work was presented at the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, January 1994

Abstract

This paper describes a memory management discipline for programs that perform dynamic memory allocation and de-allocation. At runtime, all values are put into *regions*. The store consists of a stack of regions. All points of region allocation and deallocation are inferred automatically, using a type and effect based program analysis. The scheme does not assume the presence of a garbage collector.

The scheme was first presented by Tofte and Talpin (1994); subsequently, it has been tested in The ML Kit with Regions, a region-based, garbage-collection free implementation of the Standard ML Core language, which includes recursive datatypes, higher-order functions and updatable references (Birkedal *et al.* 96, Elsman and Hallenberg 95).

This paper defines a region-based dynamic semantics for a skeletal programming language extracted from Standard ML. We present the inference system which specifies where regions can be allocated and de-allocated and a detailed proof that the system is sound with respect to a standard semantics.

We conclude by giving some advice on how to write programs that run well on a stack of regions, based on practical experience with the ML Kit.

1 Introduction

Computers have finite memory. Very often, the total of memory allocated by a program as it is run far exceeds the size of the memory. Thus, a practical discipline of programming must provide some form of memory recycling.

One of the key achievements of early work in programming languages is the invention of the notion of block structure and the associated implementation technology of stack-based memory management for recycling of memory. In block-structured languages, every point of allocation is matched by a point of deallocation and these points can easily be identified in the source program(Naur, 1963; Dijkstra 1960). Properly used, the stack discipline can result in very efficient use of memory, the maximum memory usage being bounded by the depth of the call stack rather than the number of memory allocations.

The stack discipline has its limitations, however, as witnessed by restrictions in the type systems of block-structured languages. For example, procedures are typically prevented from returning lists or procedures as results. There are two main reasons for such restrictions.

First, for the stack-discipline to work, the size of a value must be known at the latest when space for that value is allocated. This allows for example arrays which are local to a procedure and have their size determined by the arguments of the procedure; by contrast, it is not in general possible to determine how big a list is going to become, when generation of the list begins.

Second, for the stack-discipline to work, the life-time of values must comply with the allocation and deallocation scheme associated with block structure. When procedures are values, there is a danger that a procedure value refers to values which have been deallocated. For example, consider the following program:

```
(let x = (2, 3)
 in (fn y => (#1 x, y))
 end
)(5)
```

This expression is an application of a function (denoted by `(let···end)`) to the number 5. The function has formal parameter y and body `(#1` x`,` y`)`, where `#1` stands for first projection. (`fn` is pronounced λ in SML.) Thus the operator expression is supposed to evaluate to `(fn` y `=> (#1` x`,` y`))`, where x is bound to the pair $(2, 3)$, so that the whole expression evaluates to the pair $(2, 5)$. However, if we regard the `let···end` construct as a block construct (rather than just a lexical scope), we see why a stack-based implementation would not work: we cannot de-allocate the space for x at the `end`, since the first component of x is still needed by the function which is returned by the entire `let` expression.

One way to ease the limitations of the stack discipline is to allow programmer controlled allocation and de-allocation of memory, as is done in C. (C has

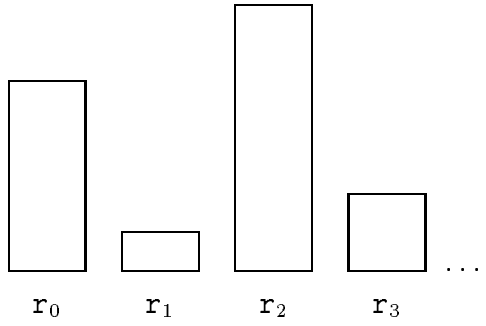


Figure 1: The store is a stack of regions; every region is uniquely identified by a *region name* (e.g., $\mathtt{r}_0$) and is depicted by a box in the picture.

two operations `malloc` and `free` for allocation and de-allocation, respectively.) Unfortunately, it is in general very hard for a programmer to know when a block of memory does not contain any live values and may therefore be freed; consequently, this solution very easily leads to so-called *space leaks*, i.e., to programs that use much more memory than expected.

Functional languages (like Haskell and Standard ML) and some object-oriented languages (e.g., JAVA) instead let a separate routine in the runtime system, the *garbage collector*, take care of de-allocation of memory (Knuth 1972; Baker 1978; Lieberman 1983). Allocation is done by the program, often at a very high rate. In our example, the three expressions `(2, 3)`, `(fn` y `=> (#1` x`,` y`))` and `(#1` x`,` y`)` each allocate memory, each time they are evaluated. The part of memory used for holding such values is called the *heap*; the rôle of the garbage collector is to recycle those parts of the heap that hold only dead values, i.e., values which are of no consequence to the rest of the computation.

Garbage collection can be very fast, provided the computer has enough memory. Indeed, there is a much quoted argument that the amortized cost of copying garbage collection tends to zero, as memory tends to infinity (Appel 1992, page 206). It is not the case, however, that languages such as Standard ML frees the programmer completely from having to worry about memory management. To write efficient SML programs, one must understand the potential dangers of, for example, accidental copying or survival of large data structures. If a program is written without concern for space usage, it may well use much more memory than one would like; even if the problem is located (using a space profiler, for example), turning a space-wasting program into a space-efficient one may require major changes to the code.

The purpose of the work reported in this paper is to advocate a compromise between the two extremes (completely manual vs completely automatic memory

management). We propose a memory model which the user may use when programming; memory can be thought of as a stack of regions, see Figure 1. Each region is like a stack of unbounded size which grows upwards in the picture, until the region in its entirety is popped off the region stack. For example, a typical use of a region is to hold a list. A program analysis automatically identifies program points where entire regions can be allocated and de-allocated and decides, for each value-producing expression, into which region the value should be put.

More specifically, we translate every well-typed source language expression, e, into a target language expression, e', which is identical with e, except for certain region annotations. The evaluation of e' corresponds, step for step, to the evaluation of e. Two forms of annotations are

$$e_1 \texttt{ at } \rho$$
$$\texttt{letregion } \rho \texttt{ in } e_2 \texttt{ end}$$

The first form is used whenever e_1 is an expression which directly produces a value. (Constant expressions, λ-abstractions and tuple expressions fall into this category.) The ρ is a *region variable*; it indicates that the value of e_1 is to be put in the region bound to ρ.

The second form introduces a region variable ρ with local scope e_2. At runtime, first an unused region, identified by a *region name*, r, is allocated and bound to ρ. Then e_2 is evaluated (probably using the region named r). Finally, the region is de-allocated. The `letregion` expression is the only way of introducing and eliminating regions. Hence regions are allocated and de-allocated in a stack-like manner.

The target program which corresponds to the above source program is

$e' \equiv$
```
letregion ρ4,ρ5
in letregion ρ6
   in let x = (2 at ρ2, 3 at ρ6) at ρ4
      in (λy.(#1 x, y) at ρ1) at ρ5
      end
   end
   5 at ρ3
end
```

We shall step through the evaluation of this expression in detail in Section 4. Briefly, evaluation starts in a region stack with three regions (ρ_1, ρ_2 and ρ_3); evaluation then allocates and de-allocates three more regions (ρ_4, ρ_5 and ρ_6) and at the end, ρ_1, ρ_2 and ρ_3 contain the final result.

The scheme forms the basis of the ML Kit with Regions, a compiler for the Standard ML Core language, including higher-order functions, references and recursive datatypes. The region inference rules we describe in this paper address life times only. A solution to the other problem, handling values of unknown size,

is addressed in (Birkedal *et al.* 96). An important optimisation turns out to be to distinguish between regions, whose size can be determined statically and those that cannot. The former can be allocated on a usual stack.

Using C terminology, region analysis infers where to insert calls to `malloc` and `free` — but beware that the analysis has only been developed in the context of Standard ML and relies on the fact that SML is rather more strongly typed than C. For a strongly typed imperative language like JAVA, region inference might be useful for freeing memory (unlike C, JAVA does not have `free`). For readers who are interested in code generation, Appendix A shows the three-address program which the ML Kit produces from the above program, using both region inference and the additional optimisations described in (Birkedal *et al.* 1996). However, this paper is primarily about the semantics of regions, not their implementation.

Experience with the Kit is that, properly used, the region scheme is strong enough to execute demanding benchmarks and to make considerable space savings, compared to a garbage-collected system (Birkedal *et al.* 96). We have found that most of the allocation is handled well by the automatic region analysis; occasionally it is too conservative and here a garbage collector would probably be useful, especially if the programmer does not know the region inference rules; for now, we have chosen instead to make (usually small) transformations to the source programs to make them more "region friendly". We shall describe some of those transformations towards the end of this paper.

A very important property of our implementation scheme is that programs are executed "as they are written", with no additional costs of unbounded size (see Appendix A for a detailed example). The memory management directives which are inserted are each constant time operations. This opens up the possibility of using languages with the power of Standard ML for applications where guarantees about time and space usage are crucial, for example in real time programming or embedded systems.

The key problem which is addressed in this paper is to prove that the region inference system is safe, in particular, that de-allocation really is safe, when the analysis claims that it is safe.

We do this as follows. We first define a standard operational semantics for our skeletal source language, giving both a static and a dynamic semantics (Section 3). We then define a region-based operational semantics for a target language; the target language is identical to the source language, except that programs have been annotated with region information (Section 4). In the dynamic semantics or the source language, there is no notion of store; in the target language semantics, however, there is a store which is organised as a stack of regions. We then specify the translation from source language to target language in the form of an inference system (Section 5). We then define a representation relation between values in a standard semantics for our skeletal language and values in a region-based semantics (Section 7) and show that, for every subexpression e of the original program, as far as the rest of the computation (after the evaluation of

e) is concerned, e and its image in the target program evaluate to related values, when evaluated in related environments (Section 9). Restricting attention to what the rest of the computation can observe turns out to be crucial: some connections between values in the source language semantics and in the region-based semantics are lost when memory is re-used in the region-based semantics. The key point is that on that part of target machine which can be observed by the rest of the computation, every value used in the source language is faithfully represented by a value in the target language.

This representation relation is defined as the maximal fixed point of a certain monotonic operator. Properties of the relation are proved using a method of proof which we call *rule-based co-induction* (Section 8.1).

Algorithms for region inference are beyond the scope of this paper; however, we shall give some hints about how the region inference rules we present can be implemented (Section 10).

2 Related Work

The main differences between the region stack and the traditional stack discipline for block-structured languages are as follows. First, when a value is created in our scheme, it is not necessarily put into the topmost region. In the case of function closures, for example, the closure is put as far down the stack as is necessary in order to be sure that the closure still exists should it ever be accessed. Second, not all regions have a size which can be determined at the time the region is allocated. Finally, the scheme works for higher-order functions and recursive datatypes and allocation is based on the basis of the type system of the language, not the grammar.

Ruggieri and Murtagh (1988) propose a stack of regions in conjunction with a traditional heap. Each region is associated with an activation record (this is not necessarily the case in our scheme). They use a combination of interprocedural and intraprocedural data-flow analysis to find suitable regions to put values in. We use a type-inference based analysis, and this is crucial for the handling of polymorphism and higher-order functions.

Inoue *et al.* (1988) present an interesting technique for compile-time analysis of runtime garbage cells in lists. Their method inserts pairs of HOLD and RECLAIMη instructions in the target language. HOLD holds on to a pointer, p say, to the root cell of its argument and RECLAIMη collects those cells that are reachable from p and fit the path description η. HOLD and RECLAIM pairs are nested, so the HOLD pointers can be held in a stack, not entirely unlike our stack of regions. In our scheme, however, the unit of collection is one entire region, i.e., there is no traversal of values in connection with region collection. The path descriptions of Inoue *et al.* make it possible to distinguish between the individual members of a list. This is not possible in our scheme, as we treat all the

elements of the same list as equal. Inoue *et al.* report a 100% reclamation rate for garbage *list* cells produced by Quicksort (Inoue 1988, page 575). We obtain a 100% reclamation rate (but for 1 word) for *all* garbage produced by Quicksort, without garbage collection (Tofte and Talpin, 1994).

Hudak (1986) describes a reference counting scheme for a first-order call-by-value functional language. Turner, Wadler and Mossin (1995) use a type system inspired by linear logic to distinguish between variables which are used at most once and variables which may be used more than once. These analyses provide somewhat different information from ours: we only distinguish between "no use" and "perhaps some use."

Georgeff (1984) describes an implementation scheme for typed lambda expressions in so-called simple form together with a transformation of expressions into simple form. The transformation can result in an increase in the number of evaluation steps by an arbitrarily large factor (Georgeff 1984, page 618). Georgeff also presents an implementation scheme which does not involve translation, although this relies on not using call-by-value reduction, when actual parameters are functions.

The device we use for grouping values according to regions is unification of region variables, using essentially the idea of Baker (1990), namely that two value-producing expressions e_1 and e_2 should be given the same "`at` ρ" annotation, if and only if type checking, directly or indirectly, unifies the type of e_1 and e_2. Baker does not prove safety, however, nor does he deal with polymorphism.

To obtain good separation of lifetimes, we use *explicit region polymorphism*, by which we mean that regions can be given as arguments to functions at runtime. For example, a declaration of the successor function `fun` $succ(x)$`=`$x+1$ is compiled into

$$\begin{array}{l}\texttt{fun}\ succ[\rho,\rho'](x)\texttt{=letregion}\ \rho'' \\ \qquad\qquad\qquad\qquad\texttt{in}\ (x + (1\ \texttt{at}\ \rho''))\ \texttt{at}\ \rho' \\ \qquad\qquad\qquad\qquad\texttt{end}\end{array}$$

Note that *succ* has been decorated with two extra formal region parameters (enclosed in square brackets to distinguish them from value variables such as x). The new *succ* function has type scheme

$$\forall\rho,\rho'.(\texttt{int},\rho) \xrightarrow{\{\mathbf{get}(\rho),\mathbf{put}(\rho')\}} (\texttt{int},\rho')$$

meaning that, for any ρ and ρ', the function accepts an integer at ρ and produces an integer at ρ' (performing a **get** operation on region ρ and a **put** operation on region ρ' in the process). Now *succ* will put its result in different regions, depending on the context:

$$\cdots\ succ[\rho_{12},\rho_9](5\ \texttt{at}\ \rho_{12})\ \cdots\ succ[\rho_1,\rho_4](y)$$

We make the additional provision that a recursive function, f, can call itself with region arguments which are different from its formal region parameters and which may well be local to the body of the recursive function. Such local regions resemble the activation records of the classical stack discipline.

We use ideas from effect inference (Lucassen 1987; Lucassen and Gifford 1988; Jouvelot and Gifford 1991) to find out where to wrap `letregion` ρ `in` ... `end` around an expression. Most work on effect inference uses the word "effect" with the meaning "side-effect" or, in concurrent languages, "communication effect"(Nielson and Nielson 1994). However, our effects are side-effects relative to the underlying region-based store model, irrespective of whether these effects stem from imperative features or not.

The idea that effect inference makes it possible to delimit regions of memory and delimit their lifetimes goes back to early work on effect systems (Lucassen *et al.* 1987). Lucassen and Gifford (1988) call it *effect masking*; they prove that (side-) effect masking is sound with respect to a store semantics where regions are not reused. Talpin (1993) and Talpin and Jouvelot (1992) present a polymorphic effect system with (side-) effect masking and prove that it is sound, with respect to a store semantics where regions are not reused.

The first version of the proof of the present paper was recorded in a technical report (Tofte and Talpin 1993), which in turn was used as the basis for the proof outline in (Tofte and Talpin, 1994). In order to simplify the proofs, several modifications to the early proofs have been done. The main differences are: (a) we have adopted value polymorphism, which simplifies the proofs in various ways; in particular a difficult lemma — Lemma 4.5 in (Tofte and Talpin 1993) — could be eliminated; (b) the dynamic semantics of the target language has been extended with region environments; (c) the definition of consistency has been strengthened to prevent closures with free region variables (these used to complicate the proof) (d) the proofs have been rewritten and reorganised around the idea of rule-based co-induction.

Aiken *et al.* (1995) have developed a program analysis which can be used as a post-pass to the analysis described in the present paper. Their analysis makes it possible to delay the allocation of regions and to promote the de-allocation, sometimes leading to asymptotic improvements in space usage and never leading to worse results than region inference without their analysis added.

3 The source language, SExp

The skeletal language treated in this paper is essentially Milner's polymorphically typed lambda calculus (Milner 78). We assume a denumerably infinite set Var of *(program) variables*. We use x and f to range over variables. Finally, c ranges over integer constants. The grammar for the source language is

$$e \;::=\; c \mid x \mid \lambda x.e \mid e_1 e_2 \mid \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end}$$

$\mid$ `letrec` $f(x)$ `=` e_1 `in` e_2 `end`

Let SExp denote the set of source language expressions. The addition of pairs and tuples to the theory is straightforward. (References, exceptions and recursive datatypes have been added in the implementation, but correctness of the translation of these constructs has not been proved.) Call-cc, concurrency primitives and other substantial extensions of Standard ML have not been studied. Nor is it clear whether region inference can be made to bear on lazy functional languages. The fact that ML is typed is essential; the fact that it has polymorphism is not essential for what follows.

3.1 Notation

In the rest of this paper we shall use the following terminology. A *finite* map is a map with finite domain. Given sets A and B, the set of finite maps from A to B is denoted $A \xrightarrow{\text{fin}} B$. 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. For any map f and set A, we write $f \downarrow A$ to mean the restriction of f to A. We sometimes write a tuple of region variables, for example, in the form $\rho_1 \cdots \rho_k$, i.e, without parentheses and commas.

We often need to select components of tuples — for example the region name of an address. In such cases, we rely on variable names to indicate which component is being selected. For example, "*r of a*" means "the region name component of *a*". (As we shall see, an address is a pair of the form (r, o), where r is a region name and o is an offset.)

3.2 Static Semantics for Source

Following Damas and Milner (1982), we have *ML types* and *ML type schemes* defined by:

τ^{ML}	$::= \mathtt{int} \mid \alpha \mid \tau^{ML} \to \tau^{ML}$	ML type
σ^{ML}	$::= \forall\alpha_1 \cdots \alpha_n.\tau^{ML}$	ML type scheme ($n \geq 0$)

where α ranges over a denumerably infinite set TyVar of *type variables*. An ML type τ_0^{ML} is *an instance* of an ML type scheme $\sigma^{ML} = \forall\alpha_1 \cdots \alpha_n.\tau^{ML}$, written $\sigma^{ML} \geq \tau_0^{ML}$, if there exist $\tau_1^{ML}, \ldots, \tau_n^{ML}$ such that $\tau^{ML}[\tau_1^{ML}/\alpha_1, \ldots, \tau_n^{ML}/\alpha_n] = \tau_0^{ML}$. An *ML type environment* is a finite map from program variables to ML type schemes. We use TE^{ML} to range over type environments. When o is an ML type, type scheme or type environment, $\mathrm{ftv}(o)$ denotes the set of type variables that occur free in o.

In Milner's original type discipline, polymorphism is associated with `let`. It has turned out that there are advantages to restricting polymorphism so that in

`let` x `=` e_1 `in` e_2 `end`, x only gets a type scheme if e_1 is a syntactic value. (In the present language, a syntactic value is an integer constant or a lambda abstraction.) Besides making it easier to prove soundness in connection with references and other language extensions, imposing this restriction also makes the proofs of correctness of region inference simpler (we have done both). In fact, we shall take the restriction one step further, and only allow polymorphism in connection with `letrec`. Any program which satisfies the value restriction can be turned into an equivalent program which only has `letrec`-polymorphism, by simply turning every `let` x `=` e_1 `in` e_2 `end` into `letrec` x'`(`z`)` `=` e_1 `in` $e_2[x'(0)/x]$ `end` where x' and z are fresh variables. In the theory that follows we therefore only have polymorphism in connection with `letrec`; with this convention, `let` x `=` e_1 `in` e_2 `end` is just syntactic sugar for $(\lambda x.e_1)(e_2)$; we show the rules for `let` even so, to make it easier to follow the examples.

$$\frac{TE^{ML}(x) = \sigma^{ML} \quad \sigma^{ML} \geq \tau^{ML}}{TE^{ML} \vdash x : \tau^{ML}}$$

$$\frac{TE^{ML} + \{x \mapsto \tau_1^{ML}\} \vdash e : \tau_2^{ML}}{TE^{ML} \vdash \lambda x.e : \tau_1^{ML} \to \tau_2^{ML}}$$

$$\frac{TE^{ML} \vdash e_1 : \tau_0^{ML} \to \tau^{ML} \qquad TE^{ML} \vdash e_2 : \tau_0^{ML}}{TE^{ML} \vdash e_1\ e_2 : \tau^{ML}}$$

$$\frac{TE^{ML} \vdash e_1 : \tau_1^{ML} \qquad TE^{ML} + \{x \mapsto \tau_1^{ML}\} \vdash e_2 : \tau^{ML}}{TE^{ML} \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end} : \tau^{ML}}$$

$$\frac{\begin{array}{c} TE^{ML} + \{f \mapsto \tau^{ML}\} \vdash \lambda\, x.e_1 : \tau^{ML} \quad \{\alpha_1, \ldots, \alpha_n\} \cap \mathrm{ftv}(TE^{ML}) = \emptyset \\ TE^{ML} + \{f \mapsto \forall \alpha_1 \cdots \alpha_n.\tau^{ML}\} \vdash e_2 : \tau_2^{ML} \end{array}}{TE^{ML} \vdash \texttt{letrec}\ f\texttt{(}x\texttt{)}\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end} : \tau_2^{ML}}$$

3.3 Dynamic Semantics for Source

A *non-recursive closure* is a triple $\langle x, e, E\rangle$, where E is an *environment*, i.e. a finite map from variables to values. We use E to range over environments; the set of environments is denoted Env. A *recursive closure* takes the form $\langle x, e, E, f\rangle$ where f is the name of the recursive function in question. A *value* is either an integer constant or a closure. We use v to range over values; the set of values is denoted Val.

Evaluation rules appear below. They allow one to infer statements of the form $E \vdash e \to v$, read: *in environment E the expression e evaluates to value v.* A closure representing a recursive function is "unrolled" just before it is applied (rule (5).

Expressions $\boxed{E \vdash e \to v}$

$$\frac{}{E \vdash c \to c} \tag{1}$$

$$\frac{E(x) = v}{E \vdash x \to v} \tag{2}$$

$$\frac{}{E \vdash \lambda x.e \to \langle x, e, E \rangle} \tag{3}$$

$$\frac{E \vdash e_1 \to \langle x_0, e_0, E_0 \rangle \qquad E \vdash e_2 \to v_2 \qquad E_0 + \{x_0 \mapsto v_2\} \vdash e_0 \to v}{E \vdash e_1\ e_2 \to v} \tag{4}$$

$$\frac{\begin{array}{c} E \vdash e_1 \to \langle x_0, e_0, E_0, f \rangle \qquad E \vdash e_2 \to v_2 \\ E_0 + \{f \mapsto \langle x_0, e_0, E_0, f \rangle\} + \{x_0 \mapsto v_2\} \vdash e_0 \to v \end{array}}{E \vdash e_1\ e_2 \to v} \tag{5}$$

$$\frac{E \vdash e_1 \to v_1 \qquad E + \{x \mapsto v_1\} \vdash e_2 \to v}{E \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \to v} \tag{6}$$

$$\frac{E + \{f \mapsto \langle x, e_1, E, f \rangle\} \vdash e_2 \to v}{E \vdash \texttt{letrec}\ f(x)\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \to v} \tag{7}$$

4 The target language, TExp

We assume a denumerably infinite set RegVar $= \{\rho_1, \rho_2, \ldots\}$ of *region variables*; we use ρ to range over region variables. The grammar for the target language, TExp, is

$$\begin{array}{rcl} e & ::= & c \mid x \mid f\ \texttt{[}\rho_1\texttt{, }\cdots\texttt{, }\rho_n\texttt{]}\ \texttt{at}\ \rho \mid \lambda x.e\ \texttt{at}\ \rho \\ & \mid & e_1\, e_2 \mid \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end} \\ & \mid & \texttt{letrec}\ f\texttt{[}\rho_1, \cdots, \rho_k\texttt{]}(x)\ \texttt{at}\ \rho\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end} \\ & \mid & \texttt{letregion}\ \rho\ \texttt{in}\ e \end{array}$$

As is common, functions are represented by closures; but region-polymorphic functions (introduced by `letrec` f`[`$\cdots$`](`x`) =` $\cdots$) are represented by so-called region function closures, which are different from closures. In the expression form $\lambda x.e$ `at` ρ , the ρ indicates the region into which the closure representing $\lambda x.e$ should be put. (Hence, the `at`ρ qualifies $\lambda x.e$, not e.) In

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

the ρ indicates where the region function closure for f should be put. A subsequent application f `[`ρ_1`, ` $\cdots$`, ` ρ_n`]` `at` ρ' extracts this region function closure from the store, applies it to actual arguments $\rho_1,\ldots,\rho_k$ and creates a function closure in ρ'.

For any finite set $\{\rho_1,\ldots,\rho_k\}$ of region variables ($k \geq 0$), we write `letregion` $\rho_1,\ldots,\rho_k$ `in` e for `letregion` ρ_1 `in` $\cdots$ `letregion` ρ_k `in` e.

We shall not present a separate static semantics for the target language, for such a semantics can be extracted from the translation rules in Section 5. We thus proceed to the dynamic semantics.

4.1 Dynamic Semantics for Target

Assume a denumerably infinite set RegName = {`r1`, `r2`, ...} of *region names*; we use r to range over region names. Region names serve to identify regions at runtime. Further, assume a denumerable infinite set, OffSet, of *offsets*; we use o to range over offsets.

A *region* is a finite map from offsets to storable values. A *storable value* is either an integer constant, a function closure, or a region function closure. We use sv to range over storable values; the set of storable values is denoted StoreVal. A *variable environment* is a finite map from program variables to values. We use VE to range over variable environments; the set of variable environments is denoted TargetEnv. A *region environment* is a finite map from region variables to region names. We use R to range over region environments; the set of region environments is denoted RegEnv. A *function closure* is a quadruple $\langle x, e', VE, R\rangle$, where x is a program variable, e' is a target language expression, and VE and R give meaning to the free program and region variables of $\lambda x.e'$. A *region function closure* is a tuple of the form $\langle \rho_1 \cdots \rho_k, x, e, VE, R\rangle$. Region function closures represent region-polymorphic functions; the region variables $\rho_1,\ldots,\rho_k$ are required to be distinct and are referred to as the *formal parameters* of the region function closure.

An *address* is a pair (r, o) of a region name and an offset. We use a to range over addresses and Addr to denote over the set of addresses. For any address a, we write r *of* a to mean the first component (i.e., the region name) of a. A *store* is a finite map from region names to regions. We use s to range over stores; the set of stores is denoted Store.

A *value* is an address. We use v to range over values; the set of values is denoted TargetVal.

We shall be brief about indirect addressing: whenever $a = (r, o)$ is an address, we write $s(a)$ to mean $s(r)(o)$. Similarly, we write $s + \{(r, o) \mapsto sv\}$ as a shorthand for $s + \{r \mapsto (s(r) + \{o \mapsto sv\})\}$. Moreover, we define the *planar domain of* s, written $\mathrm{Pdom}(s)$, to be the finite set $\{(r, o) \in \mathrm{Addr} \mid r \in \mathrm{Dom}(s) \wedge o \in \mathrm{Dom}(s(r))\}$. Finally, we write "$s \setminus\setminus \{r\}$" (read: s *without* r) to mean the store $s \downarrow (\mathrm{Dom}(s) \setminus \{r\})$.

The inference rules for the dynamic semantics of TExp are shown below. They allow one to infer sentences of the form $s, VE, R \vdash e' \to v', s'$, read: *In store* s*, variable environment* VE *and region environment* R*, the target expression* e' *evaluates to value* v' *and (a perhaps modified) store* s'*.*

Rule 10 is the evaluation rule for application of a region function closure. A function closure is created from the region closure. One can imagine that a runtime-error occurs if the premises cannot be satisfied (for example because $\rho'_i \notin \mathrm{Dom}(R)$, for som ρ'_i). However, the correctness proof shows that the premises always can be satisfied for programs that result from the translation.

Rule 14 concerns region-polymorphic and (possibly) recursive functions. To keep down the number of constructs in the target language, we have chosen to combine the introduction of recursion and region polymorphism in one language construct. Functions defined with `letrec` need not be recursive, so one can also use the `letrec` construct to define region functions that produce non-recursive functions. Rule 14 creates a region closure in the store and handles recursion by creating a cycle in the store: first a "fresh address" is chosen (by side-conditions $r = R(\rho) \qquad o \notin \mathrm{Dom}(s(r))$; the environment $VE' = VE + \{f \mapsto (r, o)\}$ is stored in the region function closure $\langle \rho_1, \cdots, \rho_k, x, e_1, VE', R \rangle$, which in turn is stored in the fresh address chosen earlier. Any reference to f in e_1 will then yield the region function closure itself, by rule 10, as desired (since `letrec` introduces recursion). Moreover, in any function application, the operator expression will evaluate to a pointer to an ordinary function closure $\langle \rho_1, \ldots, \rho_k, x, e, VE_0, R_0 \rangle$, even if the operator expression is of the the form f `[`ρ'_1`, ` $\cdots$`, `ρ'_k`]` `at` ρ. Consequently, a single rule for function application suffices.

Finally, the pushing and popping of the region stack is seen in Rule 15.

Expressions $\boxed{s, VE, R \vdash e \to v, s'}$

$$\frac{R(\rho) = r \quad o \notin \mathrm{Dom}(s(r))}{s, VE, R \vdash c \ \mathtt{at}\ \rho \to (r, o), s + \{(r, o) \mapsto c\}} \tag{8}$$

$$\frac{VE(x) = v}{s, VE \vdash x \to v, s} \tag{9}$$

$$\frac{\begin{array}{c} VE(f) = a, \quad s(a) = \langle \rho_1, \ldots, \rho_k, x, e, VE_0, R_0 \rangle \\ r = R(\rho) \qquad o \notin \mathrm{Dom}(s(r)) \qquad sv = \langle x, e, VE_0, R_0 + \{\rho_i \mapsto R(\rho_i') \ ; \ 1 \leq i \leq k\} \rangle \end{array}}{s, VE, R \vdash f \ \texttt{[}\rho_1', \ \cdots, \ \rho_k'\texttt{]} \ \texttt{at} \ \rho \rightarrow (r, o), \ s + \{(r, o) \mapsto sv\}} \tag{10}$$

$$\frac{r = R(\rho) \qquad o \notin \mathrm{Dom}(s(r))}{s, VE, R \vdash \lambda x.e \ \texttt{at} \ \rho \rightarrow (r, o), \ s + \{(r, o) \mapsto \langle x, e, VE, R \rangle\}} \tag{11}$$

$$\frac{\begin{array}{c} s, VE, R \vdash e_1 \rightarrow a_1, s_1 \qquad s_1(a_1) = \langle x_0, e_0, VE_0, R_0 \rangle \\ s_1, VE, R \vdash e_2 \rightarrow v_2, s_2 \qquad s_2, VE_0 + \{x_0 \mapsto v_2\}, R_0 \vdash e_0 \rightarrow v, s' \end{array}}{s, VE, R \vdash e_1 \ e_2 \rightarrow v, s'} \tag{12}$$

$$\frac{s, VE, R \vdash e_1 \rightarrow v_1, s_1 \qquad s_1, VE + \{x \mapsto v_1\}, R \vdash e_2 \rightarrow v, s'}{s, VE, R \vdash \texttt{let} \ x \ \texttt{=} \ e_1 \ \texttt{in} \ e_2 \rightarrow v, s'} \tag{13}$$

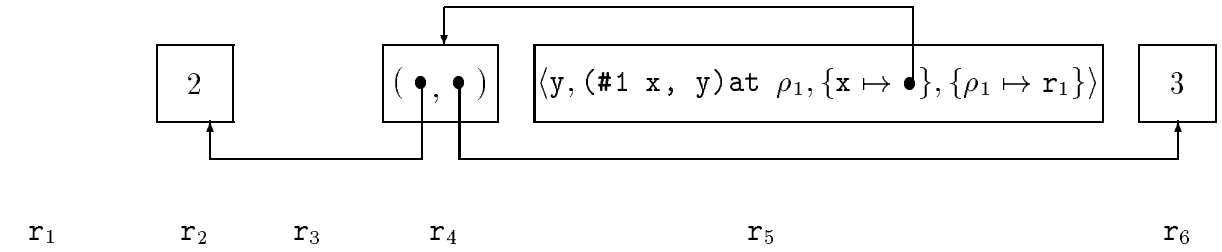
$$\frac{\begin{array}{c} r = R(\rho) \qquad o \notin \mathrm{Dom}(s(r)) \qquad VE' = VE + \{f \mapsto (r, o)\} \\ s + \{(r, o) \mapsto \langle \rho_1, \cdots, \rho_k, x, e_1, VE', R \rangle\}, VE', R \vdash e_2 \rightarrow v, s' \end{array}}{s, VE, R \vdash \texttt{letrec} \ f\texttt{[}\rho_1, \ldots, \rho_k\texttt{](}x\texttt{)} \ \texttt{at} \ \rho \ \texttt{=} \ e_1 \ \texttt{in} \ e_2 \rightarrow v, s'} \tag{14}$$

$$\frac{r \notin \mathrm{Dom}(s) \qquad s + \{r \mapsto \{\}\}, VE, R + \{\rho \mapsto r\} \vdash e \rightarrow v, s_1}{s, VE, R \vdash \texttt{letregion} \ \rho \ \texttt{in} \ e \rightarrow v, s_1 \setminus\!\setminus \{r\}} \tag{15}$$

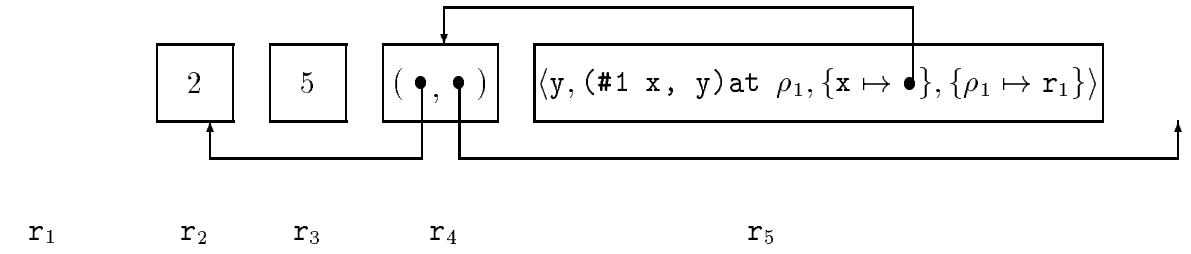
We now illustrate the use of the rules by two examples, comment on the design decisions embodied in the rules and finally prove some properties about the semantics.

4.2 Example: Function Values

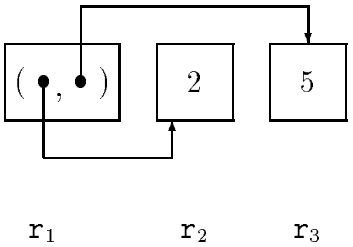
Let us consider the evaluation of the expression e' from Section 1. Since ρ_1, ρ_2 and ρ_3 occur free in e', they must be allocated before the evaluation of e' begins. We show three snapshots from the evaluation of e', namely (a) just after the closure has been allocated; (b) just before the closure is applied and (c) at the end; we assume six regions with names $\texttt{r}_1, \ldots, \texttt{r}_6$, which become bound to $\rho_1, \ldots, \rho_6$, respectively. Notice the dangling, but harmless, pointer at (b).



(a)



(b)



(c)

4.3 Example: Region polymorphism

This example illustrates region polymorphism and the use of polymorphic recursion. Consider the following source expression, which computes the 15th Fibonacci number.

```
letrec fib(x) = if x=0 then 1
                else if x=1 then 1
                else fib(x-2)+fib(x-1)
in fib(15) end
```

The corresponding target expression is shown in Figure 2. In the target expression, the `fib` function takes two arguments, namely ρ_3, which is the region where x is located, and ρ_4, which is the place where `fib` is supposed to put its result. Due to the presense of polymorphic recursion in the region inference

system, the recursive calls of `fib` use regions *different* from ρ_3 and ρ_4 (and the two recursive calls use separate regions). For example, the first call first reserves space for the result of the call (ρ_5), then reserves space for the actual argument (ρ_8), then creates the actual argument, performs the call, deallocates the actual argument and uses the result, till it can be discarded (after the `+`).

```
letregion ρ2
in letrec fib[ρ3,ρ4] at ρ2 (x:  (int, ρ3)) =
    if ···
    then 1 at ρ4
    else if ··· then 1 at ρ4
    else letregion ρ5,ρ6
         in (letregion ρ7,ρ8
             in fib[ρ8,ρ5] at ρ7
                letregion ρ9 in (x - (2 at ρ9)) at ρ8 end
             end +
             letregion ρ10,ρ11
             in fib[ρ11,ρ6] at ρ10
                letregion ρ12 in (x - (1 at ρ12)) at ρ11 end
             end) at ρ4
         end
   in letregion ρ12,ρ13
      in fib[ρ13,ρ1] at ρ12
         (15 at ρ13)
      end
   end
end
```

Figure 2: The Fibonacci function annotated with regions. The result will be a single integer in ρ_1

The `letrec` stores the following cyclic region function closure in the store at some new address, a:

$$\langle \rho_3\rho_4, \mathtt{x}, \mathtt{if}\ \cdots, \{\mathtt{fib} \mapsto a\}, \{\rho_1 \mapsto \mathtt{r}_1, \rho_2 \mapsto \mathtt{r}_2\}\rangle$$

Assuming that ρ_{13} is bound to $\mathtt{r}_3$, the application of `fib` to 15 near the end of the program stores the following function closure in the region denoted by ρ_{12}:

$$\langle \mathtt{x}, \mathtt{if}\ \cdots, \{\mathtt{fib} \mapsto a\}, \{\rho_1 \mapsto \mathtt{r}_1, \rho_2 \mapsto \mathtt{r}_2, \rho_3 \mapsto \mathtt{r}_3, \rho_4 \mapsto \mathtt{r}_1\}\rangle$$

We see that region inference has produced allocations and deallocations very similar to those of a traditional stack-based implementation. Indeed, the maximal

memory usage in this example is proportional to the maximum depth of the recursion, as it would be in a pure stack discipline.

4.4 Design Choices

The region-based semantics relies on a number of design choices, some of which are crucial.

First, it is crucial that the sets RegName and OffSet can be any (denumerable) sets. We do not assume that these sets are ordered or that there is any notion of address locality. Thus no particular physical implementation of the region stack is built into the theory. This is essential since real computers have a flat address space, whereas the region stack conceptually is two-dimensional. The particular implementation choice used in the ML Kit is described in (Birkedal *et al.* 1996)

Second, it is crucial that the semantics uses so-called "flat environments"; the alternative ("linked environments") is to represent the environment as a linked list of environment frames. This is a popular representation in block-structured languages and in some functional languages. With linked environments, closure creation is cheap, but it does not work with regions, at least if the environment frames are interspersed with regions on one stack! In example 4.2, it is essential that we copy the environment into the closure for λ`y.(#1 x, y) at` ρ_1 so that the binding for `x` is not destroyed when we leave the scope of `x` and ρ_6 and hence pop the stack.

There are also some inessential choices. There is no need to represent all objects boxed (in the ML Kit, integers and other values that fit in one machine word are represented unboxed). Recursion could probably have been implemented using unfolding of closures rather than cycles in the store. Finally, there is no deep need to keep the region environment and the variable environment separate in closures (the ML Kit merges the two) but we do so to make it clear that region names are not values.

4.5 Properties of region-based evaluation

We can now state formally that the complete evaluation of an expression does not decrease the store. For arbitrary finite maps f_1 and f_2, we say that f_2 *extends* f_1, written $f_1 \subseteq f_2$, if $\mathrm{Dom}(f_1) \subseteq \mathrm{Dom}(f_2)$ and for all $x \in \mathrm{Dom}(f_1)$, $f_1(x) = f_2(x)$. We then say that s_2 *succeeds* s_1, written $s_2 \sqsupseteq s_1$ (or $s_1 \sqsubseteq s_2$), if $\mathrm{Dom}(s_1) \subseteq \mathrm{Dom}(s_2)$ and $s_1(r) \subseteq s_2(r)$, for all $r \in \mathrm{Dom}(s_1)$.

Lemma 4.1 *If* $s, VE, R \vdash e \rightarrow v, s'$ *then* $\mathrm{Dom}(s) = \mathrm{Dom}(s')$ *and* $s \sqsubseteq s'$.

The proof is a straightforward induction on the depth of inference of $s, VE, RE \vdash e \rightarrow v, s'$. The formula $\mathrm{Dom}(s) = \mathrm{Dom}(s')$ in Lemma 4.1 expresses that the store resulting from the elaboration has neither more nor less regions than the store

in which the evaluation begins, although other regions may have been allocated temporarily during the evaluation. The evaluation of e may write values in existing regions, so it is possible to have $s(r) \subset s'(r)$, for some r. However, e never removes or overwrites any of the values that are in s.

4.6 Syntactic Equality of Expressions

Let e' be a target expression. The set of program variables that occur free in e' is written $\mathrm{fpv}(e')$. The set of region variables that occur free in e' is $\mathrm{frv}(e')$.

Both in the source language and in the target language, we shall consider two expressions equal, if they can be obtained from each other by renaming of bound variables. This extends to closures. For example, $\langle x_1, e_1, VE_1 \rangle$ and $\langle x_2, e_2, VE_2 \rangle$ are considered equal if $VE_1 = VE_2$ and $\lambda x_1.e_1$ and $\lambda x_2.e_2$ are equal in the above sense. Moreover, we even allow that the free variables of $\lambda x_2.e_2$ may be a renaming of the free variables of $\lambda x_1.e_1$, provided of course that the corresponding change has been made in the domain of VE_1 to obtain VE_2. (Loosely speaking, this corresponds to admitting value environments as declarations and then allowing the usual renamings permitted in an expression of the form `let` VE_1 `in` $\lambda x_1.e_1$ `end` .) Finally, we consider $\langle x, e, VE_1 \rangle$ and $\langle x, e, VE_2 \rangle$ equal, if $VE_1 \downarrow \mathrm{fpv}(\lambda x.e) = VE_2 \downarrow \mathrm{fpv}(\lambda x.e)$. This allows us to introduce and delete unused program variables in the domains of environments inside closures.

Similarly, for any region closure $\langle \vec{\rho}, x, e, VE, R \rangle$ we allow the renamings of $\vec{\rho}$, x, $\mathrm{fpv}(e)$ and $\mathrm{frv}(e)$ and the introduction or elimination of unused program variables that one would expect if the closure were written `let` VE, R `in` $\lambda \vec{\rho}, x_1.e_1$ `end` .

Equality on semantic objects in each of the two dynamic semantics is then defined to be the smallest equivalence relation which is closed under the three transformations described above.

5 Region inference

The rules that specify which translations are legal are called the *region inference rules*. In Section 5.1 we present region types and other semantic objects that occur in the region inference rules; the rules themselves are presented in Section 5.2. In Sections 5.3 and 5.4 we state and prove properties of the region inference system, for example that the translation is a refinement of Milner's type discipline.

5.1 Semantic objects

Region types

We assume three denumerably infinite, pairwise disjoint sets:

$\alpha \in \mathrm{TyVar}$ type variables

ρ or $p \in$ RegVar — region variables
$\epsilon \in$ EffectVar — effect variables

To avoid too many subscripts and primes, we use both p (for "place") and ρ to range over region variables. An *atomic effect* is a term of the form

$$\eta \ ::= \mathbf{put}(\rho) \mid \mathbf{get}(\rho) \mid \epsilon \quad \text{atomic effect}$$

We use η to range over atomic effects. An *effect* is a finite set of atomic effects. We use φ to range over effects. For a concrete example, the effect of expression e' in Example 4.2 is $\{\mathbf{put}(\rho_1), \mathbf{put}(\rho_2), \mathbf{put}(\rho_3)\}$.

Types and types with places are given by:

$\tau \ ::= \mathtt{int} \mid \alpha \mid \mu \xrightarrow{\epsilon.\varphi} \mu$ — type
$\mu \ ::= (\tau, \rho)$ — type with place

In a function type

$$\mu \xrightarrow{\epsilon.\varphi} \mu' \tag{16}$$

the object $\epsilon.\varphi$ is called an *arrow effect*. Formally, an arrow effect is a pair of an effect variable and an effect; we refer to ϵ and φ as the *handle* and the *latent effect*, respectively. If a function f has type (16) then the latent effect φ is to be interpreted as the effect of evaluating the body of f. Effect variables are useful for expressing dependencies between effects. For example, the target expression

$$e' \equiv (\lambda \mathtt{f}.(\lambda \mathtt{x}.\mathtt{f}(\mathtt{x}))\ \mathtt{at}\ \rho_4)\ \mathtt{at}\ \rho_5$$

can be given type

$$\tau_{e'} = \begin{array}{l} ((\alpha_1, \rho_1) \xrightarrow{\epsilon_1.\emptyset} (\alpha_2, \rho_2), \rho_3) \xrightarrow{\epsilon_2.\{\mathbf{put}(\rho_4)\}} \\ \qquad ((\alpha_1, \rho_1) \xrightarrow{\epsilon_3.\{\mathbf{get}(\rho_3), \epsilon_1\}} (\alpha_2, \rho_2), \rho_4) \end{array} \tag{17}$$

In (17) the last occurrence of ϵ_1 indicates that for all e_1 and e_2 of the appropriate type, if e_1 evaluates to some function, g, and e_2 evaluates to some value, v, then the evaluation of $(e'e_1)e_2$ may involve an application of g. (As it happens, the evaluation would indeed involve an application of g, but the type does not express that.)

Equality of types is defined by term equality, as usual, but up to set equality of latent effects. For example, the arrow effects $\epsilon.\{\mathbf{put}(\rho), \mathbf{get}(\rho')\}$ and $\epsilon.\{\mathbf{get}(\rho'), \mathbf{put}(\rho)\}$ are considered equal.

One might wonder why we have a pair $\epsilon.\varphi$ on the function arrow rather than just, say, an effect φ. The reason is that the region inference algorithms we use rely on unification, just as ML type inference does (Damas and Milner 1982). Thus the effect sets on function arrows pose a problem for the existence of principal unifiers. A solution is to use arrow effects together with certain invariants about the use of effect variables. The basic idea is that effect variables uniquely "stand for" effects: if $\epsilon_1.\varphi_1$ and $\epsilon_2.\varphi_2$ both occur in a proof tree formed

by the inference algorithm and $\epsilon_1 = \epsilon_2$ then it will also be the case that $\varphi_1 = \varphi_2$. Moreover, if two arrow effects $\epsilon_1.\varphi_1$ and $\epsilon_2.\varphi_2$ both occur in a proof tree and $\epsilon_2 \in \varphi_1$ then $\varphi_2 \subseteq \varphi_1$: the presence of ϵ_2 in φ_1 implies that φ_2 subsumes the entire effect φ_1 which ϵ_1 stands for. With these representation invariants and using the special notion of substitution defined below, one can prove the existence of principal unifiers, even though types "contain" effects (which are sets). A detailed account of how this is done is beyond the scope of this paper. Also, the invariants mentioned above are not needed for proving the soundness of region inference, so we shall not consider them in what follows.

Substitution

A *type substitution* is a map from type variables to types; we use S_t to range over type substitutions. A *region substitution* is a map from region variables to region variables; we use S_r to range over region substitutions. An *effect substitution* is a map from effect variables to arrow effects; we use S_e to range over effect substitutions. A *substitution* is a triple (S_t, S_r, S_e); we use S to range over substitutions. Substitution on types, region variables and effects is defined as follows. Let $S = (S_t, S_r, S_e)$; then

Effects

$$\begin{aligned} S(\varphi) \;=\; & \{\mathbf{put}(S_r(\rho)) \mid \mathbf{put}(\rho) \in \varphi\} \cup \\ & \{\mathbf{get}(S_r(\rho)) \mid \mathbf{get}(\rho) \in \varphi\} \cup \\ & \{\eta \mid \exists \epsilon, \epsilon', \varphi'.\ \epsilon \in \varphi \wedge \epsilon.\varphi' = S_e(\epsilon) \wedge \eta \in \{\epsilon'\} \cup \varphi'\} \end{aligned}$$

Types and Region Variables

$$S(\mathtt{int}) = \mathtt{int} \quad S(\alpha) = S_t(\alpha) \quad S(\rho) = S_r(\rho)$$

$$S(\tau, \rho) = (S(\tau), S(\rho))$$

$$S(\mu \xrightarrow{\epsilon.\varphi} \mu') = S(\mu) \xrightarrow{\epsilon'.(\varphi' \cup S(\varphi))} S(\mu'), \quad \text{where } \epsilon'.\varphi' = S_e(\epsilon)$$

For a concrete example, consider the substitution $S = (S_r, S_t, S_e)$, where

$$\begin{aligned} S_e(\epsilon) \;&=\; \begin{cases} \epsilon_8.\{\mathbf{get}(\rho_1), \mathbf{put}(\rho_2)\} & \text{if } \epsilon = \epsilon_1; \\ \epsilon & \text{otherwise} \end{cases} \\ S_t(\alpha) \;&=\; \begin{cases} \mathtt{int} & \text{if } \alpha = \alpha_1 \text{ or } \alpha = \alpha_2; \\ \alpha & \text{otherwise} \end{cases} \\ S_r(\rho) \;&=\; \rho \quad \text{for all } \rho \end{aligned}$$

where ϵ_1, ρ_1, ρ_2, α_1 and α_2 refer to (17). Now we have

$$\begin{aligned} S(\tau_{e'}) = \quad & ((\mathtt{int}, \rho_1) \xrightarrow{\epsilon_8.\{\mathbf{get}(\rho_1), \mathbf{put}(\rho_2)\}} (\mathtt{int}, \rho_2), \rho_3) \xrightarrow{\epsilon_2.\{\mathbf{put}(\rho_4)\}} \\ & ((\mathtt{int}, \rho_1) \xrightarrow{\epsilon_3.\{\mathbf{get}(\rho_1), \mathbf{get}(\rho_3), \mathbf{put}(\rho_2), \epsilon_8\}} (\mathtt{int}, \rho_2), \rho_4) \end{aligned} \tag{18}$$

This more specific type for e' is appropriate if e' occurs in the following application expression:

$$e'((\lambda \mathtt{n} : (\mathtt{int}, \rho_1).(\mathtt{n+1})\ \mathtt{at}\ \rho_2)\ \mathtt{at}\ \rho_3) \tag{19}$$

for which one will then be able to infer the type and place

$$((\mathtt{int}, \rho_1) \xrightarrow{\epsilon_3.\{\mathbf{get}(\rho_1),\mathbf{get}(\rho_3),\mathbf{put}(\rho_2),\epsilon_8\}} (\mathtt{int}, \rho_2), \rho_4)$$

In applying substitutions to semantic objects with bound names (e.g., a type scheme) bound variables are first renamed to avoid capture, when necessary. Substitutions compose; Id is the identity substitution.

The *support* of a type substitution S_t, written $\mathrm{Supp}(S_t)$, is the set $\{\alpha \in \mathrm{TyVar} \mid S_t(\alpha) \neq \alpha\}$. Similarly for region substitutions. The *support* of an effect substitution S_e, written $\mathrm{Supp}(S_e)$, is the set $\{\epsilon \in \mathrm{EffectVar} \mid S_e(\epsilon) \neq \epsilon.\emptyset\}$. The support of a substitution $S = (S_t, S_r, S_e)$, written $\mathrm{Supp}(S)$, is defined as $\mathrm{Supp}(S_t) \cup \mathrm{Supp}(S_r) \cup \mathrm{Supp}(S_e)$. Whenever S_t, S_r and S_e are finite maps of the appropriate types we take the liberty to consider the triple (S_t, S_r, S_e) a substitution, without explicitly extending the finite maps to total maps.

Type schemes

Type schemes resemble the type schemes of Damas and Milner (1982) but with additional quantification over region variables and effect variables:

$$\begin{array}{lll} \sigma & ::= \forall().\tau & \text{simple type scheme} \\ & \mid\ \forall\rho_1\cdots\rho_k\alpha_1\cdots\alpha_n\epsilon_1\cdots\epsilon_m.\underline{\tau} & \text{compound type scheme} \end{array}$$

where $n \geq 0$, $k \geq 0$ and $m \geq 0$. The following definitions are stated for compound type schemes but are easily extended to simple type schemes. For a type scheme $\sigma = \forall\rho_1\cdots\rho_k\alpha_1\cdots\alpha_n\epsilon_1\cdots\epsilon_m.\underline{\tau}$, the *bound variables of* σ, written $\mathrm{bv}(\sigma)$, is the set

$$\{\rho_1, \ldots, \rho_k, \alpha_1, \ldots, \alpha_n, \epsilon_1, \ldots, \epsilon_m\}$$

We sometimes write the sequences of bound variables as vectors: $\vec{\alpha}$, $\vec{\rho}$ and $\vec{\epsilon}$, respectively. Two type schemes are *equivalent* if they can be obtained from each other by renaming and reordering of bound variables. A type τ' is an *instance of* σ, written $\sigma \geq \tau'$, if there exists a substitution S such that $\mathrm{Supp}(S) \subseteq \mathrm{bv}(\sigma)$ and $S(\tau) = \tau'$. When we want to make S explicit, we say that τ' is an instance of σ *via* S, written $\sigma \geq \tau'$ *via* S. Equivalent type schemes have the same instances.

We sometimes write τ as a shorthand for the simple type scheme $\forall().\tau$, not to be confused with the compound type scheme $\forall().\underline{\tau}$, since compound type schemes have a special significance: they are used exclusively as types of region-polymorphic functions, even for those region-polymorphic functions that take an empty list of actual region parameters. The underlining serves to make it clear whether a type scheme is to be regarded as simple or compound.

$$
\begin{aligned}
\alpha &\in \text{TyVar} \\
p \text{ or} \rho &\in \text{RegVar} \\
\epsilon &\in \text{EffectVar} \\
\varphi &\in \text{Effect} = \text{Fin}(\text{AtomicEffect}) \\
&\quad \text{AtomicEffect} = \text{EffectVar} \cup \text{PutEffect} \cup \text{GetEffect} \\
\mathbf{put}(\rho) &\in \text{PutEffect} = \text{RegVar} \\
\mathbf{get}(\rho) &\in \text{GetEffect} = \text{RegVar} \\
\tau &\in \text{Type} = \text{TyVar} \cup \text{ConsType} \cup \text{FunType} \\
&\quad \text{SimpleTypeScheme} = \text{Type} \\
&\quad \text{CompoundTypeScheme} = \\
&\qquad \cup_{k \geq 0} \text{RegVar}^k \times \cup_{n \geq 0} \text{TyVar}^n \times \cup_{m \geq 0} \text{EffectVar}^m \times \text{Type} \\
\sigma &\in \text{TypeScheme} = \text{SimpleTypeScheme} \cup \text{CompoundTypeScheme} \\
&\quad \text{ConsType} = \{\texttt{int}\} \\
\mu \xrightarrow{\epsilon.\varphi} \mu &\in \text{FunType} = \text{TypeWithPlace} \times \text{ArrowEffect} \times \text{TypeWithPlace} \\
\epsilon.\varphi &\in \text{ArrowEffect} = \text{EffectVar} \times \text{Effect} \\
\mu &\in \text{TypeWithPlace} = \text{Type} \times \text{RegVar} \\
TE &\in \text{TyEnv} = \text{Var} \xrightarrow{\text{fin}} (\text{TypeScheme} \times \text{RegVar})
\end{aligned}
$$

Figure 3: Semantic objects of region inference

A *type environment* is a finite map from program variables to pairs of the form (σ, ρ). We use TE to range over type environments.

The semantic objects are summarised in Figure 3. The notion of free variables extend to larger semantic objects, such as type environments. (For example, a type variable is said to occur free in TE if it occurs free in $TE(x)$, for some x.) For any semantic object A, $\text{frv}(A)$ denotes the set of region variables that occur free in A; $\text{ftv}(A)$ denotes the set of type variables that occur free in A; $\text{fev}(A)$ denotes the set of effect variables that occur free in A; and $\text{fv}(A)$ denotes the union of the above.

5.2 The inference system

The inference rules allow the inference of statements of the form

$$TE \vdash e \Rightarrow e' : \mu, \varphi$$

read: *in TE, e translates to e′, which has type and place μ and effect φ.* The region inference rules are non-deterministic: given TE and e, there may be infinitely many e', μ and φ satisfying $TE \vdash e \Rightarrow e' : \mu, \varphi$. This non-determinism is convenient to express type-polymorphism, but we also use it to express freedom in the choice of region variables. Indeed, the region inference rules allow one to put all values in a single region, although, in practice, this would be the worst possible choice.

Region-based translation of expressions $\boxed{TE \vdash e \Rightarrow e' : \mu, \varphi}$

$$\frac{}{TE \vdash c \Rightarrow c\,\mathtt{at}\,\rho : (\mathtt{int}, \rho), \{\mathbf{put}(\rho)\}} \tag{20}$$

$$\frac{TE(x) = (\tau, \rho)}{TE \vdash x \Rightarrow x : (\tau, \rho), \emptyset} \tag{21}$$

$$\frac{\begin{array}{c} TE(f) = (\sigma, \rho') \quad \sigma = \forall \rho_1 \cdots \rho_k \vec{\alpha} \vec{\epsilon}.\underline{\tau_1} \\ \sigma \geq \tau \text{ via } S \qquad \varphi = \{\mathbf{get}(\rho'), \mathbf{put}(\rho)\} \end{array}}{TE \vdash f \Rightarrow f\mathtt{[}S(\rho_1), \ldots, S(\rho_k)\mathtt{]}\ \mathtt{at}\ \rho : (\tau, \rho), \varphi} \tag{22}$$

$$\frac{\begin{array}{c} TE + \{x \mapsto \mu_1\} \vdash e \Rightarrow e' : \mu_2, \varphi \\ \varphi \subseteq \varphi' \quad \tau = \mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2 \quad \mathrm{frv}(e') \subseteq \mathrm{frv}(TE, \tau) \end{array}}{TE \vdash \lambda x.e \Rightarrow \lambda x.e'\ \mathtt{at}\ \rho : (\tau, \rho), \{\mathbf{put}(\rho)\}} \tag{23}$$

$$\frac{TE \vdash e_1 \Rightarrow e_1' : (\mu' \xrightarrow{\epsilon.\varphi} \mu, \rho), \varphi_1 \qquad TE \vdash e_2 \Rightarrow e_2' : \mu', \varphi_2}{TE \vdash e_1\ e_2 \Rightarrow e_1'\ e_2' : \mu, \varphi \cup \varphi_1 \cup \varphi_2 \cup \{\epsilon, \mathbf{get}(\rho)\}} \tag{24}$$

$$\frac{\begin{array}{c} TE \vdash e_1 \Rightarrow e_1' : (\tau_1, \rho_1), \varphi_1 \\ TE + \{x \mapsto (\tau_1, \rho_1)\} \vdash e_2 \Rightarrow e_2' : \mu, \varphi_2 \end{array}}{TE \vdash \mathtt{let}\ x\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2\ \mathtt{end} \Rightarrow \mathtt{let}\ x\ \mathtt{=}\ e_1'\ \mathtt{in}\ e_2'\ \mathtt{end} : \mu, \varphi_1 \cup \varphi_2} \tag{25}$$

$$\frac{\begin{array}{c} TE + \{f \mapsto (\forall \vec{\rho} \vec{\epsilon}.\underline{\tau}, \rho_0)\} \vdash \lambda\, x.e_1 \Rightarrow \lambda\, x.e_1'\ \mathtt{at}\ \rho_0 : (\tau, \rho_0), \varphi_1 \\ \mathrm{fv}(\vec{\alpha}, \vec{\rho}, \vec{\epsilon}) \cap \mathrm{fv}(TE, \varphi_1) = \emptyset \\ TE + \{f \mapsto (\forall \vec{\alpha}\, \vec{\rho}\, \vec{\epsilon}.\underline{\tau}, \rho_0)\} \vdash e_2 \Rightarrow e_2' : \mu, \varphi_2 \end{array}}{\begin{array}{l} TE \vdash \mathtt{letrec}\ f(x)\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2\ \mathtt{end} \Rightarrow \\ \qquad \mathtt{letrec}\ f\mathtt{[}\vec{\rho}\mathtt{]}(x)\ \mathtt{at}\ \rho_0\ \mathtt{=}\ e_1'\ \mathtt{in}\ e_2'\ \mathtt{end} : \mu, \varphi_1 \cup \varphi_2 \end{array}} \tag{26}$$

$$\frac{TE \vdash e \Rightarrow e' : \mu, \varphi \qquad \rho \notin \mathrm{frv}(TE, \mu)}{TE \vdash e \Rightarrow \mathtt{letregion}\ \rho\ \mathtt{in}\ e'\ \mathtt{end} : \mu, \varphi \setminus \{\mathbf{put}(\rho), \mathbf{get}(\rho)\}} \tag{27}$$

$$\frac{TE \vdash e \Rightarrow e' : \mu, \varphi \qquad \epsilon \notin \mathrm{fev}(TE, \mu)}{TE \vdash e \Rightarrow e' : \mu, \varphi \setminus \{\epsilon\}} \tag{28}$$

In Rule 21, note that the effect of referring to x is empty; this is because the effects only relate to access of the region store s, not the environments VE and R. In Rule 22 the instances of the bound region variables become actual region

parameters in the target expression. The resulting effect includes $\mathbf{get}(\rho')$ and $\mathbf{put}(\rho)$, for we access the region closure in ρ' and create an ordinary function closure in ρ.

In rule 23, the effect of creating the function closure at region ρ is simply $\{\mathbf{put}(p)\}$. Following Talpin and Jouvelot (1992), one is allowed to make the information about the function less precise by increasing the latent effect. This is useful in cases where two expressions must have the same functional type (including the latent effects on the arrows) but may evaluate to different closures. The freedom to increase effects is also useful when one wants to prove that every well-typed Exp-program of Milner (1978, 1982) can be translated with the region inference rules — see Lemma 5.2 below. We shall explain the side-condition $\mathrm{frv}(e') \subseteq \mathrm{frv}(TE, \tau)$ in a moment.

In Rule 24 we see that the latent effect is brought out when the function is applied. The $\mathbf{get}(\rho)$ in the resulting effect is due to the fact that we must access the closure at ρ in order to perform the function application.

In Rule 26, note that f is region-polymorphic, but not type-polymorphic, inside e_1, its own body. In e_2, however, f is polymorphic in types, regions and effects. Without the limitation on type-polymorphism inside e_1, region inference would not be decidable.

Rule 27 concerns the introduction of `letregion` expressions. The basic idea, which goes back to early work on effect systems (Lucassen *et al*, 1987), is this. Suppose $TE \vdash e \Rightarrow e' : \mu, \varphi$ and assume that ρ is a region variable which does not occur free in TE or in μ (typically, ρ occurs free in φ, indicating that ρ is used in the computation of e'). *Then ρ is purely local to the evaluation of e', in the sense that the rest of the computation will not access any value stored in ρ.*

Example

Once again, consider the expression e' from Section 1. Let e'_0 be the subexpression

$$
\begin{array}{rcl}
e'_0 & \equiv & \texttt{let x = (2 at } \rho_2\texttt{, 3 at } \rho_6\texttt{) at } \rho_4 \\
 & & \texttt{in } (\lambda\texttt{y.(\#1 x, y) at } \rho_1\texttt{) at } \rho_5 \\
 & & \texttt{end}
\end{array}
$$

The type environment in force when this expression is produced is $TE_0 = \{\}$, the type and place of e'_0 is

$$\mu_0 = ((\mathtt{int}, \rho_3) \xrightarrow{\epsilon_1.\{\mathbf{get}(\rho_4), \mathbf{put}(\rho_1)\}} ((\mathtt{int}, \rho_2) * (\mathtt{int}, \rho_3), \rho_1), \rho_5)$$

and the effect of e'_0 is $\varphi_0 = \{\mathbf{put}(\rho_2), \mathbf{put}(\rho_6), \mathbf{put}(\rho_4), \mathbf{put}(\rho_5)\}$. Note that ρ_6 is the only region variable which occurs free in φ_0 but occurs free neither in TE_0 nor in μ_0. Rule 27 allows us to discharge ρ_6, resulting in the effect $\{\mathbf{put}(\rho_2), \mathbf{put}(\rho_4), \mathbf{put}(\rho_5)\}$ and the `letregion` ρ_6 in e'. □

Next, Rule 28 allows one to discharge an effect variable from the effect; no `letregion` is introduced, since the discharge does not influence evaluation.

We owe an explanation for the side-condition $\mathrm{frv}(e') \subseteq \mathrm{frv}(TE, \tau)$ in Rule 23. Often, but not always it is the case that every region variable which occur free in a translated expression occurs free either in the type or in the effect of the expression. Here is an example where this fails to hold:

$$\{\} \vdash (\lambda f.1)(\lambda x.2) \Rightarrow ((\lambda f.1\ \mathtt{at}\ \rho_1)\ \mathtt{at}\ \rho_2)((\lambda x.2\ \mathtt{at}\ \rho_3)\ \mathtt{at}\ \rho_4) : (\mathtt{int}, \rho_1), \varphi$$

where $\varphi = \{\mathbf{put}(\rho_2), \mathbf{put}(\rho_3), \mathbf{get}(\rho_2), \mathbf{put}(\rho_1)\}$. Here we see that ρ_3 is free in the target expression but occurs free neither in the effect nor in the resulting type and place. The reason is that $2\ \mathtt{at}\ \rho_3$ is "dead code": it can never be executed. In Rule 23 we demand that there be no such free region variables in the body of the lambda abstraction. This side-condition can always be satisfied by applying Rule 27 repeatedly, if necessary, just before applying rule 27. However, the side-condition simplifies the soundness proof (specifically the proof of Lemma 8.3).

As mentioned earlier, the region inference rules give rise to a static semantics for the target language: one just consistency replaces sentences of the $TE \vdash e \Rightarrow e' : \mu, \varphi$ by $TE \vdash e' : \mu, \varphi$. However, we prefer the present form, which emphasises the the rules specify a translation.

5.3 Region inference is a refinement of Milner's type system

In this section we prove that the region inference system is a refinement of Milner's type discipline (Milner 1978) in the sense that an expression can be translated with the region rules if and only if it is well-typed according to Milner's type discipline, as defined in Section 3.2. In particular, this shows that the problem of determining whether a closed expression can be region-annotated is decidable.

We first show that an expression can be translated only if it is well-typed. To this end, we define a function, π, (for "projection") from semantic objects in the region rules to the semantic objects in the Milner rules:

$$\pi(\alpha) = \alpha; \quad \pi(\mathtt{int}) = \mathtt{int}; \quad \pi(\mu \xrightarrow{\epsilon.\varphi} \mu') = \pi(\mu) \to \pi(\mu')$$

$$\pi(\tau, \rho) = \pi(\tau); \quad \pi(\forall \vec{\rho}\,\vec{\alpha}\,\vec{\epsilon}.\tau) = \forall \vec{\alpha}.\pi(\tau); \quad \pi(\sigma, \rho) = \pi(\sigma); \quad \pi(TE) = \pi \circ TE$$

Lemma 5.1 *If* $TE \vdash e \Rightarrow e' : \mu, \varphi$ *then* $\pi(TE) \vdash e : \pi(\mu)$.

The proof is a straightforward induction on the depth of $TE \vdash e \Rightarrow e' : \mu, \varphi$. □

Next we show that every well-typed term can be translated. To this end we define a relation, R, between Milner's objects and ours. Let ρ_0 be some fixed

region variable and let ϵ_0 be some fixed effect variable. The basic idea is to choose ρ_0 everywhere we need a region variable in the translation and to choose $\epsilon_0.\{\mathbf{get}(\rho_0), \mathbf{put}(\rho_0), \epsilon_0\}$ everywhere we need an arrow effect in the translation. Unfortunately, we cannot simply make R a map, because of the distinction between simple and compound type schemes. So we define R inductively as follows:

$$\frac{}{\alpha\ R\ \alpha} \qquad \frac{}{\mathtt{int}\ R\ \mathtt{int}} \qquad \frac{\tau\ R\ \mu \qquad \tau'\ R\ \mu'}{(\tau \to \tau')\ R\ (\mu \xrightarrow{\epsilon_0.\{\mathbf{get}(\rho_0),\mathbf{put}(\rho_0),\epsilon_0\}} \mu')}$$

$$\frac{\tau\ R\ \tau'}{\forall().\tau\ R\ \forall().\tau'} \qquad \frac{\tau\ R\ \tau'}{\forall\vec{\alpha}.\tau\ R\ \forall\vec{\alpha}.\underline{\tau'}} \qquad \frac{\tau\ R\ \tau'}{\tau\ R\ (\tau', \rho_0)} \qquad \frac{\sigma\ R\ \sigma'}{\sigma\ R\ (\sigma', \rho_0)}$$

$$\frac{\mathrm{Dom}(TE) = \mathrm{Dom}(TE') \qquad \forall x \in \mathrm{Dom}(TE).TE(x)\ R\ TE'(x)}{TE\ R\ TE'}$$

Clearly, for every TE there exists a TE' such that $TE\ R\ TE'$.

Lemma 5.2 *If $TE \vdash e : \tau$ and $TE\ R\ TE'$ then $TE' \vdash e \Rightarrow e' : \mu, \varphi$ for some e', μ and φ which satisfy $\tau\ R\ \mu$, $\mathrm{frv}(\mu) = \{\rho_0\}$, $\mathrm{frv}(e') \subseteq \{\rho_0\}$ and $\varphi \subseteq \{\mathbf{get}(\rho_0), \mathbf{put}(\rho_0), \epsilon_0\}$.*

Proof By induction on the depth of inference of $TE \vdash e : \tau$. We show only two cases, as the rest are straightforward.

$\boxed{e \equiv x}$ By assumption we have $TE(x) = \sigma$ and $\sigma \geq \tau$. Since $TE\ R\ TE'$ we then have $TE'(x) = (\sigma', \rho_0)$ for some σ' which satisfies $\sigma\ R\ \sigma'$. Now σ' may be simple or compound, but if it is compound it has no quantified region variables. Let $\mu = (\tau', \rho_0)$ be the unique type with place satisfying $\tau\ R\ \mu$. Then $\sigma' \geq \tau'$ and the desired conclusion follows either by Rule 21 or by Rule 22.

$\boxed{e \equiv \lambda x.e_1}$ Here $\tau = \tau_1 \to \tau_2$ for some τ_1 and τ_2 and $TE \vdash \lambda x.e_1 : \tau$ must have been inferred from the premise $TE + \{x \mapsto \tau_1\} \vdash e_1 : \tau_2$. We have $(TE + \{x \mapsto \tau_1\})\ R\ (TE' + \{x \mapsto \mu_1\})$, where μ_1 is the unique type with place related to τ_1. By induction there exist e_1', μ_2 and φ_0 such that $TE' + \{x \mapsto \mu_1\} \vdash e_1 \Rightarrow e_1' : \mu_2, \varphi_0$, $\mathrm{frv}(\mu_2) = \{\rho_0\}$, $\mathrm{frv}(e_1') \subseteq \{\rho_0\}$ and $\varphi_0 \subseteq \{\mathbf{get}(\rho_0), \mathbf{put}(\rho_0), \epsilon_0\}$. Now Rule 23 conveniently allows us to use this inclusion to prove $TE' \vdash \lambda x.e_1 \Rightarrow \lambda x.e_1'\ \mathtt{at}\ \rho_0 : (\mu_1 \xrightarrow{\epsilon_0.\{\mathbf{get}(\rho_0),\mathbf{put}(\rho_0),\epsilon_0\}} \mu_2, \rho_0), \{\mathbf{put}(\rho_0)\}$ from which the desired results follows. □

5.4 Substitution lemma

Lemma 5.3 *For all substitutions S, if $TE \vdash e \Rightarrow e' : \mu, \varphi$ then $S(TE) \vdash e \Rightarrow S(e') : S(\mu), S(\varphi)$.*

The proof is a straightforward induction on the depth of the inference of $TE \vdash e \Rightarrow e' : \mu, \varphi$, using appropriate variants of S in the case for `letrec`.

Next, we shall state a lemma to the effect that the operation of making type schemes in the type environment more type-polymorphic does not decrease the set of possible translations. Formally, we say that σ_1 is at least as type-polymorphic as σ_2, written $\sigma_1 \sqsupseteq \sigma_2$, if σ_1 and σ_2 are identical, or σ_1 and σ_2 are both compound and $\sigma_1 = \forall\vec{\alpha}.\sigma_2$, for some $\vec{\alpha}$. Furthermore, we write $TE_1 \sqsupseteq TE_2$ if $\mathrm{Dom}(TE_1) = \mathrm{Dom}(TE_2)$ and, for all $x \in \mathrm{Dom}(TE_1)$, if $(\sigma_1, \rho_1) = TE_1(x)$ and $(\sigma_2, \rho_2) = TE_2(x)$ then $\sigma_1 \sqsupseteq \sigma_2$ and $\rho_1 = \rho_2$.

Lemma 5.4 *If* $TE \vdash e \Rightarrow e' : \mu, \varphi$ *and* $TE' \sqsupseteq TE$ *then* $TE' \vdash e \Rightarrow e' : \mu, \varphi$.

We omit the proof, which is a straightforward induction on the depth of inference of $TE \vdash e \Rightarrow e' : \mu, \varphi$. We note, however, that the similar statement concerning region polymorphism (replacing $\sigma = \forall\vec{\alpha}\,\vec{\epsilon}.\underline{\tau}$ by $\sigma' = \forall\vec{\rho}\,\vec{\alpha}\,\vec{\epsilon}.\underline{\tau}$) is not true, because applications of region functions in the target expression can be affected by such a change.

Fortunately, it is precisely the ability to make assumed type schemes more type-polymorphic that we need.

6 Using Effects to Describe Continuations

For the proof of the soundness of the translation scheme, we need to relate the values of the dynamic semantics of the source and target language. We refer to this relation as the *consistency* relation.

Since all values are addresses in the target language semantics, the consistency relation must involve stores. Consistency also naturally depends on types: at type `int`, source level integers can only be consistent with pointers to integers in the target; at a functional type, only closures can be related, and so on. The region inference rules yield expressions, types with places, and effects — all of which can contain free occurrences of region variables. To relate these region variables to the region names which identify regions at runtime, we need a region environment, R, and the following definition:

Definition 6.1 *A region environment* R connects effect φ to store s, *if* $\mathrm{frv}(\varphi) \subseteq \mathrm{Dom}(R)$ *and for all* $\rho \in \mathrm{frv}(\varphi)$, $R(\rho) \in \mathrm{Dom}(s)$.

Based on these considerations, assume that we have defined consistency as a relation

$$\mathcal{C} \subseteq \mathrm{RegEnv} \times \mathrm{TypeWithPlace} \times \mathrm{Val} \times \mathrm{Store} \times \mathrm{TargetVal}$$

where $\mathcal{C}(R, \mu, v, s, v')$ is read: *in region environment* R *and store* s, *source value* v *is consistent with target value* v' *at type with place* μ. The obvious idea

would now be somehow to lift this relation first from types with places to type schemes, $\mathcal{C}(R, \sigma, v, s, v')$, and then, by pointwise extension, to environments, $\mathcal{C}(R, TE, E, s, VE)$. We might then try to prove the following statement:

Conjecture 6.1 *If* $TE \vdash e \Rightarrow e' : \mu, \varphi$ *and* $E \vdash e \to v$ *and* $\mathcal{C}(R, TE, e, s, VE)$ *and* R *connects* φ *to* s *then there exists a store* s' *and a target value* v' *such that* $s, VE, R \vdash e' \to v', s'$ *and* $\mathcal{C}(R, \mu, v, s', v')$.

However, there is a problem with this conjecture. Informally, it states that consistency is preserved by evaluation. Unfortunately, we cannot expect that to hold! To see what the problem is, consider example 4.2 once more. According to the conjecture, at point (b) we should have that the source language closure $\langle \texttt{y}, \texttt{(\#1 x, y)}, \{\texttt{x} \mapsto (2,3)\}\rangle$ and the closure found in region $\texttt{r}_5$ are consistent. In a sense they are consistent: application of the two closures map consistent arguments to consistent results. But notice that the consistency which used to exist between the source environment $\{x \mapsto (2,3)\}$ and its representation in the target semantics was partly destroyed when the region $\texttt{r}_6$ was popped from the region stack. Thus we see that, intuitively speaking, consistency gradually deteriorates during computation. The saving factor, it turns out, is that there is always enough consistency left for the rest of the computation to succeed, without running into any of the inconsistencies!

To make these intuitions precise, we need some notion of "consistency with respect to the rest of the computation". One possibility is to work explicitly with continuations or evaluation contexts. However, we have not explored this possibility, since all we need for the purpose of the soundness proof is a very simple summary of which regions are accessed by the rest of the computation. Specifically, it suffices to summarise the rest of the computation by an effect, φ', which describes which of the currently existing regions are accessed by the rest of the computation. Thus we define a relation

$$\mathcal{C} \subseteq \text{RegEnv} \times \text{TypeWithPlace} \times \text{Val} \times \text{Store} \times \text{TargetVal} \times \text{Effect}$$

where $\mathcal{C}(R, \mu, v, s, v', \varphi')$, also written $\mathcal{C}(R, \mu, v, s, v')$ w.r.t. φ', is read: *at type with place* μ*, in region environment* R *and store* s*, source value* v *is consistent with target value* v' *with respect to the effect* φ' (where φ' represents the effect of the rest of the computation). In our example, φ' is $\{\mathbf{put}(\rho_3), \mathbf{get}(\rho_5), \mathbf{put}(\rho_1)\}$, connected via the region environment to regions $\texttt{r}_3$, $\texttt{r}_5$ and $\texttt{r}_1$. The fact that the rest of the computation does not access the current contents of $\texttt{r}_6$ is evident from the fact that no region variable free in φ' is connected to $\texttt{r}_6$! That is why the environments in the two closures are consistent with respect to the rest of the computation. The second version of our conjecture becomes:

Conjecture 6.2 *If* $TE \vdash e \Rightarrow e' : \mu, \varphi$ *and* $E \vdash e \to v$ *and* $\mathcal{C}(R, TE, e, s, VE)$ w.r.t. $(\varphi \cup \varphi')$ *and* R *connects* $\varphi \cup \varphi'$ *to* s *then there exists a store* s' *and a target value* v' *such that* $s, VE, R \vdash e' \to v', s'$ *and* $\mathcal{C}(R, \mu, v, s', v')$ w.r.t. φ'.

In other words, if we start out with consistency to cover both the evaluation of e' (whose effect is φ) and the rest of the computation (whose effect is φ') then after the computation of e', we will have enough consistency left for the rest of the computation.

However, Conjecture 6.2 is not quite strong enough to be proved by induction. Consider a source language closure $\langle x, e, E\rangle$ and a target closure $\langle x, e', VE, R\rangle$, which we think of as representing $\langle x, e, E\rangle$. When the source closure is applied, the body e will be evaluated in an environment $E + \{x \mapsto v_2\}$, where v_2 is the argument to the function. Assuming that v_2' is some target value consistent with v_2, the corresponding evaluation in the target language takes the form $s, VE + \{x \mapsto v_2'\}, R \vdash e' \rightarrow \cdots$. However, the region environment in which e' is evaluated is not necessarily the same as the region environment R' which is in force at the point where the application takes place, for more regions may have been allocated since the closure was created. Moreover, R' is important for establishing that $E + \{x \mapsto v_2\}$ and $VE + \{x \mapsto v_2'\}$ are consistent, since v_2 and v_2' will be known to be consistent in R', not in R. And we must establish consistency of $E + \{x \mapsto v_2\}$ and $VE + \{x \mapsto v_2'\}$ in order to use induction to prove that the results of the function applications are consistent.

Example

Consider the target expression

```
letregion ρ1
in let x = 3 at ρ1
   in letregion ρ2
      in let f = (λy.(x+y) at ρ0) at ρ2
         in letregion ρ3
            in f(4 at ρ3)
            end
         end
      end
   end
end
```

Consider the point of the evaluation just after the closure for `f` has been created. Let us say that the region environment is $R_1 = \{\rho_0 \mapsto \mathtt{r}_0, \rho_1 \mapsto \mathtt{r}_1, \rho_2 \mapsto \mathtt{r}_2\}$. Then the store is

$$s_1 = \{\mathtt{r}_0 \mapsto \{\}, \mathtt{r}_1 \mapsto \{o_x \mapsto 3\}, \mathtt{r}_2 \mapsto \{o_f \mapsto \langle \mathtt{y}, (\mathtt{x+y})\ \mathtt{at}\ \rho_0, \{\mathtt{x} \mapsto (\mathtt{r}_1, o_x)\}, R_1\rangle\}$$

We can reasonable expect to have

$$\mathcal{C}(R_1, \{\mathtt{x} \mapsto (\mathtt{int}, \rho_1)\}, \{\mathtt{x} \mapsto 3\}, s_1, \{\mathtt{x} \mapsto (\mathtt{r}_1, o_x)\}) \text{ w.r.t. } \varphi_1 \qquad (29)$$

where $\varphi_1 = \{\mathbf{get}(\rho_1), \mathbf{get}(\rho_2), \mathbf{put}(\rho_0)\}$, which is the net-effect of the remainder of the computation at that point. ("Expect" because we have not defined $\mathcal{C}$ yet.) Next, consider the point where the actual argument 4 to `f` has been stored, the closure for `f` has been fetched and we are just about to evaluate the body of `f`. Now the region environment has become $R_2 = R_1 + \{\rho_3 \mapsto \mathtt{r}_3\}$, the store has become $s_2 = s_1 + \{\mathtt{r}_3 \mapsto \{o_4 \mapsto 4\}\}$ and we can reasonably expect to have

$$\mathcal{C}(R_2, (\mathtt{int}, \rho_3), 4, s_2, (\mathtt{r}_3, o_4)) \text{ w.r.t. } \varphi_2 \tag{30}$$

where $\varphi_2 = \{\mathbf{get}(\rho_1), \mathbf{get}(\rho_3), \mathbf{put}(\rho_0)\}$, i.e., the effect of the continuation at that point. From (29) and (30) we can reasonably expect to obtain

$$\begin{aligned}\mathcal{C}(R_2, \quad & \{\mathtt{x} \mapsto (\mathtt{int}, \rho_1), \mathtt{y} \mapsto (\mathtt{int}, \rho_3)\}, \\ & \{\mathtt{x} \mapsto 3, \mathtt{y} \mapsto 4\}, s_2, \{\mathtt{x} \mapsto (\mathtt{r}_1, o_x), \mathtt{y} \mapsto (\mathtt{r}_3, o_4)\}) \text{ w.r.t. } \varphi_2\end{aligned}$$

But evaluation of the function body is going to take place in R_1 (see rule 12). Thus the theorem needs to be strong enough to handle the situation that the region environment in which consistency is established is not the same as the region environment in which the expression is evaluated. Incidentally, this is similar to the situation in block-structured languages, where an an inner block can call a function declared in an enclosing block. (Indeed, it appears that although the variable environments do not obey a stack discipline, the region environments do.) □

We therefore prove that the theorem holds not just for R but also for other region environments R' which "agree" with R:

Definition 6.2 *Let R and R' be region environments and let φ be an effect. We say that R and R'* agree on φ, *if $R \downarrow \mathrm{frv}(\varphi) = R' \downarrow \mathrm{frv}(\varphi)$.*

We are now able to state the main theorem, which we shall prove, once we have defined the consistency relation:

Theorem 6.1 *If $TE \vdash e \Rightarrow e' : \mu, \varphi$ and $\mathcal{C}(R, TE, E, s, VE)$* w.r.t. *$\varphi \cup \varphi'$ and $E \vdash e \to v$ and R connects $\varphi \cup \varphi'$ to s and R' and R agree on $\varphi \cup \varphi'$ and $\mathrm{frv}(e') \subseteq \mathrm{Dom}\, R'$ then there exist s' and v' such that $s, VE, R' \vdash e' \to v', s'$ and $\mathcal{C}(R', \mu, v, s', v')$* w.r.t. *$\varphi'$.*

The premise "$\mathrm{frv}(e') \subseteq \mathrm{Dom}\, R'$" is included only to make the proof simpler; it helps to ensure that closures in the target language will not contain free region variables.

Note that we use the effect of the rest of the computation as an approximation to what data is "live"; the notion usually employed by garbage collectors (namely

that data is live, if it is reachable in the memory graph) is incomparable: we have already seen that data which is reachable in the memory graph is actually dead and can be deallocated using region inference; conversely, sometimes data which we keep alive in a region is not actually used by the rest of the computation and a garbage collector would detect it.

7 Consistency

For simplicity, we first present the consistency relation in the form of inference rules without reference to the underlying mathematics. We shall later explain that the rules can be viewed as describing a maximal fixed point of a certain monotonic operator. For now, it suffices to read the rules as follows: the conclusion of a rule holds if and only if the premises hold.

Rules 31–35 characterize consistency between source values and storable target values sv (defined in Section 4.1). These rules are used in Rules 36 and 37, to characterize consistency between source and target values (recall that target values are addresses). It is precisely in rules Rule 36 and 37 we see the significance of the idea of representing the rest of the computation by the effect φ: if $\mathbf{get}(\rho) \notin \varphi$, then any claim about consistency of values at region ρ is allowed, for ρ then denotes "garbage". However, by rule 36, if $v' = (r, o) \in \mathrm{Pdom}(s)$ and $r = R(\rho)$ then the value stored at address v' has to be consistent with the source value, v, as described by rules 34 and 35. (Recall that $(r, o) \in \mathrm{Pdom}(s)$ abbreviates $r \in \mathrm{Dom}(s) \wedge o \in \mathrm{Dom}(s(r))$.) Rule 38 says that consistency of environments is the pointwise extension of consistency of values.

Rule 31 should be straightforward. In rule 32, note that TE does not occur in the conclusion of the rule: one has to "invent" a TE which can justify the target expression as a compilation result of the source expression. Also, the environments E and VE must be consistent at TE. The region environment R may be regarded as the region environment which is in force when the closures are applied; as we saw earlier, this is not necessarily the same as the region environment which was in force when the target closure was created (R' in the rule). For the purpose of the soundness theorem, we clearly need to know that R and R' are related somehow, and it turns out that it suffices to require that they agree on φ. The condition $\mathrm{frv}(e') \subseteq R'$ ensures that the target closure contains no free region variables; the two first premises of the rule already ensure that $\mathrm{fpv}(e') \subseteq \mathrm{Dom}(VE)$, i.e., that the closure contains no free program variables. Again this is good hygiene which is useful in the proofs (specifically of Lemma 8.3).

Rule 33 is similar to Rule 32, but deals with recursion. For the premises to be satisfied, TE mush have f in its domain. Moreover, since recursion is handled by unfolding in the source language semantics, it is $E + \{f \mapsto \langle x, e, E, f\rangle\}$ and VE that have to be consistent, rather than just E and VE.

Rule 34 is similar to Rule 33, but it relates recursive closures and region function closures at compound type schemes. For simple type schemes, one uses Rule 35 together with rules 31–33.

Types and Storable Values $\boxed{\mathcal{C}(R, \mu, v, s, sv) \text{ w.r.t. } \varphi}$

$$\frac{i \in \mathtt{Int}}{\mathcal{C}(R, (\mathtt{int}, \rho), i, s, i) \text{ w.r.t. } \varphi} \tag{31}$$

$$\frac{\begin{array}{c} TE \vdash \lambda x.e \Rightarrow \lambda x.e' \,\mathtt{at}\, \rho : (\tau, \rho), \{\mathbf{put}(\rho)\} \\ \mathcal{C}(R', TE, E, s, VE) \text{ w.r.t. } \varphi \\ R' \text{ and } R \text{ agree on } \varphi \quad \mathrm{frv}(e') \subseteq \mathrm{Dom}(R') \end{array}}{\mathcal{C}(R, (\tau, \rho), \langle x, e, E\rangle, s, \langle x, e', VE, R'\rangle) \text{ w.r.t. } \varphi} \tag{32}$$

$$\frac{\begin{array}{c} TE \vdash \lambda x.e \Rightarrow \lambda x.e' \,\mathtt{at}\, \rho : (\tau, \rho), \{\mathbf{put}(\rho)\} \\ \mathcal{C}(R', TE, E + \{f \mapsto \langle x, e, E, f\rangle\}, s, VE) \text{ w.r.t. } \varphi \\ R' \text{ and } R \text{ agree on } \varphi \quad \mathrm{frv}(e') \subseteq \mathrm{Dom}(R') \end{array}}{\mathcal{C}(R, (\tau, \rho), \langle x, e, E, f\rangle, s, \langle x, e', VE, R'\rangle) \text{ w.r.t. } \varphi} \tag{33}$$

Type Schemes and Storable Values $\boxed{\mathcal{C}(R, (\sigma, \rho), v, s, sv) \text{ w.r.t. } \varphi}$

$$\frac{\begin{array}{c} TE + \{f \mapsto (\sigma, \rho)\} \vdash \lambda x.e \Rightarrow \lambda x.e' \;\mathtt{at}\; \rho : (\tau, \rho), \{\mathbf{put}(\rho)\} \\ \sigma = \forall \rho_1 \cdots \rho_k \alpha_1 \cdots \alpha_n \epsilon_1 \cdots \epsilon_m . \underline{\tau} \qquad \mathrm{bv}(\sigma) \cap \mathrm{fv}(TE, \rho) = \emptyset \\ R' \text{ and } R \text{ agree on } \varphi \quad \mathrm{frv}(e') \subseteq \mathrm{Dom}(R') \cup \{\rho_1, \ldots, \rho_k\} \\ \mathcal{C}(R', TE + \{f \mapsto (\sigma, \rho)\}, E + \{f \mapsto \langle x, e, E, f\rangle\}, s, VE) \text{ w.r.t. } \varphi \end{array}}{\mathcal{C}(R, (\sigma, \rho), \langle x, e, E, f\rangle, s, \langle \rho_1, \ldots, \rho_k, x, e', VE, R'\rangle) \text{ w.r.t. } \varphi} \tag{34}$$

$$\frac{\mathcal{C}(R, (\tau, \rho), v, s, sv) \text{ w.r.t. } \varphi}{\mathcal{C}(R, (\forall().\tau, \rho), v, s, sv) \text{ w.r.t. } \varphi} \tag{35}$$

Type Schemes and addresses $\boxed{\mathcal{C}(R, (\sigma, \rho), v, s, v') \text{ w.r.t. } \varphi}$

$$\frac{v' = (r, o) \quad R(\rho) = r \quad v' \in \mathrm{Pdom}(s) \qquad \mathcal{C}(R, (\sigma, \rho), v, s, s(v')) \text{ w.r.t. } \varphi}{\mathcal{C}(R, (\sigma, \rho), v, s, v') \text{ w.r.t. } \varphi} \tag{36}$$

$$\frac{\mathbf{get}(\rho) \notin \varphi}{\mathcal{C}(R, (\sigma, \rho), v, s, v') \text{ w.r.t. } \varphi} \tag{37}$$

Environments $\boxed{\mathcal{C}(R, TE, E, s, VE) \text{ w.r.t. } \varphi}$

$$\frac{\begin{array}{c}\text{Dom } TE = \text{Dom } E = \text{Dom } VE \\ \forall x \in \text{Dom } TE.\, \mathcal{C}(R, TE(x), E(x), s, VE(x)) \text{ w.r.t. } \varphi\end{array}}{\mathcal{C}(R, TE, E, s, VE) \text{ w.r.t. } \varphi} \tag{38}$$

The relation $\mathcal{C}$ is defined as the maximal fixed point of an operator $\mathcal{F} : \mathcal{P}(\mathcal{C}) \to \mathcal{P}(\mathcal{C})$, where $\mathcal{P}$ means powerset and C is defined by:

$$\begin{array}{rcl} C & = & \text{RegEnv} \times \text{TypeWithPlace} \times \text{Val} \times \text{Store} \times \text{StoreVal} \times \text{Effect} \\ & \cup & \text{RegEnv} \times (\text{TypeScheme} \times \text{RegVar}) \times \text{Val} \times \text{Store} \times \text{StoreVal} \times \text{Effect} \\ & \cup & \text{RegEnv} \times (\text{TypeScheme} \times \text{RegVar}) \times \text{Val} \times \text{Store} \times \text{TargetVal} \times \text{Effect} \\ & \cup & \text{RegEnv} \times \text{TyEnv} \times \text{Env} \times \text{Store} \times \text{TargetEnv} \times \text{Effect} \end{array}$$

The members of C are referred to as *(consistency) claims*. We use γ to range over claims and Γ to range over sets of claims. For example, a claim of the form $(R, (\sigma, \rho), v, s, sv, \varphi)$ is read: (it is claimed that) storable value sv is consistent with source value v and has type scheme σ and resides at ρ in the store s and region environment R, with respect to effect φ.

Note that $(\mathcal{P}(C), \subseteq)$ is a complete lattice. We now define an operator $\mathcal{F} : \mathcal{P}(C) \to \mathcal{P}(C)$. The definition is expressed using the syntax of inference rules, but it could equally well be expressed as a non-recursive definition by cases; for given $\Gamma \subseteq C$, $\mathcal{F}(\Gamma)$ is defined as the unique set $\{\gamma \in C \mid \gamma \in \mathcal{F}(\Gamma)$ can be inferred by one of the inference rules $\}$. Since the rules a very similar to rules 31–38 we shall not explain them further.

Types and Storable Values $\boxed{(R, \mu, v, s, sv, \varphi) \in \mathcal{F}(\Gamma)}$

$$\frac{i \in \texttt{Int}}{(R, (\texttt{int}, \rho), i, s, i, \varphi) \in \mathcal{F}(\Gamma)} \tag{39}$$

$$\frac{\begin{array}{c}TE \vdash \lambda x.e \Rightarrow \lambda x.e' \texttt{ at } \rho : (\tau, \rho), \{\mathbf{put}(\rho)\} \\ (R', TE, E, s, VE, \varphi) \in \Gamma \\ R' \text{ and } R \text{ agree on } \varphi \quad \text{frv}(e') \subseteq \text{Dom}(R')\end{array}}{(R, (\tau, \rho), \langle x, e, E\rangle, s, \langle x, e', VE, R'\rangle, \varphi) \in \mathcal{F}(\Gamma)} \tag{40}$$

$$\frac{\begin{array}{c}TE \vdash \lambda x.e \Rightarrow \lambda x.e' \texttt{ at } \rho : (\tau, \rho), \{\mathbf{put}(\rho)\} \\ (R', TE, E + \{f \mapsto \langle x, e, E, f\rangle\}, s, VE, \varphi) \in \Gamma \\ R' \text{ and } R \text{ agree on } \varphi \quad \text{frv}(e') \subseteq \text{Dom}(R')\end{array}}{(R, (\tau, \rho), \langle x, e, E, f\rangle, s, \langle x, e', VE, R'\rangle, \varphi) \in \mathcal{F}(\Gamma)} \tag{41}$$

Type Schemes and Storable Values $\boxed{(R,(\sigma,\rho),v,s,sv,\varphi)\in\mathcal{F}(\Gamma)}$

$$\frac{\begin{array}{c}TE+\{f\mapsto(\sigma,\rho)\}\vdash\lambda x.e\Rightarrow\lambda x.e'\ \mathtt{at}\ \rho:(\tau,\rho),\{\mathbf{put}(\rho)\}\\ \sigma=\forall\rho_1\cdots\rho_k\alpha_1\cdots\alpha_n\epsilon_1\cdots\epsilon_m.\tau\qquad \mathrm{bv}(\sigma)\cap\mathrm{fv}(TE,\rho)=\emptyset\\ R'\text{ and }R\text{ agree on }\varphi\quad \mathrm{frv}(e')\subseteq\mathrm{Dom}(R')\cup\{\rho_1,\ldots,\rho_k\}\\ (R',TE+\{f\mapsto(\sigma,\rho)\},E+\{f\mapsto\langle x,e,E,f\rangle\},s,VE,\varphi)\in\Gamma\end{array}}{(R,(\sigma,\rho),\langle x,e,E,f\rangle,s,\langle\rho_1,\ldots,\rho_k,x,e',VE,R'\rangle,\varphi)\in\mathcal{F}(\Gamma)}\tag{42}$$

$$\frac{(R,(\tau,\rho),v,s,sv,\varphi)\in\Gamma}{(R,(\forall().\tau,\rho),v,s,sv,\varphi)\in\mathcal{F}(\Gamma)}\tag{43}$$

Type Schemes and addresses $\boxed{(R,(\sigma,\rho),v,s,v',\varphi)\in\mathcal{F}(\Gamma)}$

$$\frac{v'=(r,o)\quad R(\rho)=r\quad v'\in\mathrm{Pdom}(s)\qquad(R,(\sigma,\rho),v,s,s(v'),\varphi)\in\Gamma}{(R,(\sigma,\rho),v,s,v',\varphi)\in\mathcal{F}(\Gamma)}\tag{44}$$

$$\frac{\mathbf{get}(\rho)\notin\varphi}{(R,(\sigma,\rho),v,s,v',\varphi)\in\mathcal{F}(\Gamma)}\tag{45}$$

Environments $\boxed{(R,TE,E,s,VE,\varphi)\in\mathcal{F}(\Gamma)}$

$$\frac{\begin{array}{c}\mathrm{Dom}\,TE=\mathrm{Dom}\,E=\mathrm{Dom}\,VE\\ \forall x\in\mathrm{Dom}\,TE.(R,TE(x),E(x),s,VE(x),\varphi)\in\Gamma\end{array}}{(R,TE,E,s,VE,\varphi)\in\mathcal{F}(\Gamma)}\tag{46}$$

The operator $\mathcal{F}$ is monotonic: $\Gamma\subseteq\Gamma'$ implies $\mathcal{F}(\Gamma)\subseteq\mathcal{F}(\Gamma')$. Thus, by Tarski's fixed point theorem, there exists a greatest fixed point for $\mathcal{F}$ and this greatest fixed point is also the greatest set Γ satisfying $\Gamma\subseteq\mathcal{F}(\Gamma)$. Let $\Gamma^{\#}$ be this greatest fixed point.

Definition 7.1 *We take $\mathcal{C}$ to be $\Gamma^{\#}$ and we write, for example, $\mathcal{C}(R,\mu,v,s,v')$*w.r.t. *$\varphi$ to mean $(R,\mu,v,s,v',\varphi)\in\mathcal{C}$.*

We use co-induction to prove properties of the consistency relation: to prove that a set Γ of claims is consistent, (ie., that $\Gamma\subseteq\Gamma^{\#}$) it suffices to prove $\Gamma\subseteq\mathcal{F}(\Gamma)$.

8 Properties of Consistency

In this section we prove important lemmas about the consistency relation $\mathcal{C}$. Besides being useful in the proof of the main theorem (Theorem 6.1) they address issues such as why it is safe to re-use a deallocated region even when there are dead pointers into it. The lemmas will be proved using a special style of co-inductive proof, which we call rule-based co-induction.

8.1 Rule-based Co-induction

Rule-based co-inductive proof is a style of proof which makes it possible to present a co-inductive proof in a form which resembles ordinary induction on depth of inference. The scenario is that a set, C, is given, together with an operator $\mathcal{F} : \mathcal{P}(C) \to \mathcal{P}(C)$ which is monotonic with respect to set inclusion. $\mathcal{F}$ is defined by a finite set of inference rules (in our case, Rules 39–46). Let $\Gamma^\#$ be the maximal fixed point of $\mathcal{F}$: $\Gamma^\# = \cup\{\Gamma \subseteq C \mid \Gamma \subseteq \mathcal{F}(\Gamma)\}$. Now consider a lemma which states that, for some given relation $R \subseteq C \times C$:

$$\forall \gamma, \gamma' \in C \text{ if } \gamma \in \Gamma^\# \text{ and } \gamma \; R \; \gamma' \text{ then } \gamma' \in \Gamma^\# \tag{47}$$

Let $\Gamma_R = \{\gamma' \in C \mid \exists \gamma \in \Gamma^\#.\gamma \; R \; \gamma'\}$. We refer formally to the members γ' of Γ_R as the *consequences* of the lemma. Then (47) can be stated $\Gamma_R \subseteq \Gamma^\#$. By the principle of co-induction, it suffices to prove $\Gamma_R \subseteq \mathcal{F}(\Gamma_R)$, i.e., that

$$\forall \gamma' \in C \text{ if there exists } \gamma \in \Gamma^\# \text{ such that } \gamma \; R \; \gamma' \text{ then } \gamma' \in \mathcal{F}(\Gamma_R)$$

Thus the co-inductive proof can be organised as follows: take any $\gamma' \in C$. Let $\gamma \in \Gamma^\#$ be such that $\gamma \; R \; \gamma'$. Show $\gamma' \in \mathcal{F}(\Gamma_R)$, i.e., *show that γ' can be inferred by the inference rules that define $\mathcal{F}$, using only premises which are themselves consequences of the lemma.* Often, this is proved by a case analysis on γ (note: not γ'), since $\gamma \in \Gamma^\#$ implies that γ can be inferred by an application of one of the rules that define $\mathcal{F}$ from premises which are themselves in $\Gamma^\#$. Note that proving $\gamma' \in \mathcal{F}(\Gamma_R)$ is equivalent to inferring $\gamma' \in \Gamma^\#$, using the fixed-point rules for $\mathcal{F}$ (in our case: Rules 31–38) and only using premises γ'_i which are themselves consequences of the lemma (i.e., $\forall i \exists \gamma_i \in \Gamma^\#.\gamma_i \; R \; \gamma'_i$). Thus we can word the co-inductive proof almost as if it were a normal inductive proof on the depth of inference related to mininal fixed points, using the fixed point rules for $\mathcal{F}$ rather than the rules that define $\mathcal{F}$.

We name this style of co-inductive proof *rule-based co-induction.* We emphasise that a rule-based co-inductive proof is *not* a proof on "depth of inference" — for the co-inductive proof establishes claims that are not conclusions of any finite proof tree constructed by the fixed point rules.

8.2 Preservation of consistency

The first lemma states that consistency is preserved under decreasing effect and increasing store. This is to be expected: it is easier to obtain consistency with respect to an observer if the observer observes a little rather than a lot; and the larger the store is, the easier it is for it to contain bits of target values which are consistent with a given source value.

Lemma 8.1 *If* $\mathcal{C}(R, \mu, v, s_1, v')$ w.r.t. φ_1 *and* $\varphi_2 \subseteq \varphi_1$ *and* $s_1 \sqsubseteq s_2$ *then* $\mathcal{C}(R, \mu, v, s_2, v')$ w.r.t. φ_2.

Lemma 8.1 is a special case of the following lemma:

Lemma 8.2 *If* $\mathcal{C}(R_1, \mu, v, s_1, v')$ w.r.t. φ_1 *and* $\varphi_2 \subseteq \varphi_1$ *and* R_2 *and* R_1 *agree on* φ_2 *and* $s_1 \downarrow (\mathrm{Rng}(R_2 \downarrow \mathrm{frv}(\varphi_2))) \sqsubseteq s_2$ *then* $\mathcal{C}(R_2, \mu, v, s_2, v')$ w.r.t. φ_2. *Similarly for the other forms of* $\mathcal{C}$.

Notice that the domain of s_1 need not be a subset of the domain of s_2 for Lemma 8.2 to apply. This is crucial in the proof of the main theorem, in the case for `letregion`. Here s_1 will be the store resulting from a computation which involves local regions; s_2 will be the result of removing the local regions from s_1. The region variables that are free in φ_1, but not in φ_2 will be the variables of the local regions.

Proof We prove Lemma 8.2 and the corresponding statements concerning the other forms of consistency by rule-based co-induction. The cases for the inference rules (31) to (38) are arranged according to judgement forms. In all cases, we assume

$$\varphi_2 \subseteq \varphi_1 \tag{48}$$

$$R_2 \text{ and } R_1 \text{ agree on } \varphi_2 \tag{49}$$

$$s_1 \downarrow (\mathrm{Rng}(R_2 \downarrow \mathrm{frv}(\varphi_2))) \sqsubseteq s_2 \tag{50}$$

Types and Storable Values $\boxed{\mathcal{C}(R, \mu, v, s, sv) \text{ w.r.t. } \varphi}$

Assume

$$\mathcal{C}(R_1, \mu, v, s_1, sv) \text{ w.r.t. } \varphi_1 \tag{51}$$

By the remarks in Section 8 it suffices to prove that $\mathcal{C}(R_2, \mu, v, s_2, sv)$ w.r.t. φ_2 can be inferred using Rules 31–38, from premises which are themselves conclusions of the lemma.

Recall that Rules 31–38 express that $\mathcal{C}$ is a fixed-point of $\mathcal{F}$: one has (51) if and only if either the "premises" (i.e., the formulae above the line) of Rule 31 hold, or the premises of Rule 32 hold, or the premises of Rule 33 hold. We deal with each case in turn:

Rule 31 Here $\mu = (\mathtt{int}, \rho)$, for some ρ, and $v = sv = i$, for some $i \in \mathrm{Int}$. But then $\mathcal{C}(R_2, \mu, v, s_2, sv)$ w.r.t. φ_2, by Rule 31.

Rule 32 Here there exist τ, ρ, TE, x, e, E, e', VE, R' such that (51) is inferred from premises

$$TE \vdash \lambda x.e \Rightarrow \lambda x.e' \ \mathtt{at}\ \rho : (\tau, \rho), \{\mathbf{put}(\rho)\} \tag{52}$$

$$\mathcal{C}(R', TE, E, s_1, VE) \text{ w.r.t. } \varphi_1 \tag{53}$$

$$R' \text{ and } R_1 \text{ agree on } \varphi_1 \quad \mathrm{frv}(e') \subseteq \mathrm{Dom}(R') \tag{54}$$

and $\mu = (\tau, \rho)$, $v = \langle x, e, E\rangle$ and $sv = \langle x, e', VE, R'\rangle$. But then, by (54), (48) and (49) we have

$$R' \text{ and } R_2 \text{ agree on } \varphi_2 \tag{55}$$

Obviously, R' agrees with itself on φ_2 and, by (55) and (50), $s_1 \downarrow (\mathrm{Rng}(R' \downarrow \mathrm{frv}(\varphi_2))) \sqsubseteq s_2$. Thus, using also (48) and (53), we have that the claim

$$\mathcal{C}(R', TE, E, s_2, VE) \text{ w.r.t. } \varphi_2 \tag{56}$$

is a consequence of the lemma.[2] Thus by Rule 32 on (52), (55) and (56) we have $\mathcal{C}(R_2, \mu, v, s_2, sv)$ w.r.t. φ_2, as desired (since (56) is a consequence of the lemma).

Rule 33 Similar to the previous case.

Type Schemes and Storable Values $\boxed{\mathcal{C}(R, (\sigma, \rho), v, s, sv) \text{ w.r.t. } \varphi}$

Assume $\mathcal{C}(R_1, (\sigma, \rho), v, s_1, sv)$ w.r.t. φ_1, which can be inferred by Rule 34 or by Rule 35. The case for Rule 34 is similar to the case for Rule 32. So consider the case for Rule 35. Here σ takes the form $\forall().\tau$ and we have $\mathcal{C}(R_1, (\tau, \rho), v, s_1, sv)$w.r.t. φ_1. Thus the claim $\mathcal{C}(R_2, (\tau, \rho), v, s_2, sv)$ w.r.t. φ_2 is a conseqence of the lemma. But then, by Rule 35 we have $\mathcal{C}(R_2, (\sigma, \rho), v, s_2, sv)$w.r.t.$\varphi_2$, as required (since the premise used, i.e., $\mathcal{C}(R_2, (\tau, \rho), v, s_2, sv)$ w.r.t. φ_2, is a consequence of the lemma).

Type Schemes and addresses $\boxed{\mathcal{C}(R, (\sigma, \rho), v, s, v') \text{ w.r.t. } \varphi}$

Assume

$$\mathcal{C}(R_1, (\sigma, \rho), v, s_1, v') \text{ w.r.t. } \varphi_1 \tag{57}$$

inferred by Rule 36 or Rule 37. Case analysis:

[2] Strictly speaking, we should say "we have that the claim $(R', TE, E, s_2, VE, \varphi_2)$ is a consequence of the lemma", but the chosen formulation seems easier to read, so we adopt it throughout.

$\boxed{\mathbf{get}(\rho) \in \varphi_2}$ Then $\mathbf{get}(\rho) \in \varphi_1$ so by (36) there exist r, o such that $v' = (r, o)$ and

$$R_1(\rho) = r \tag{58}$$

$$v' \in \mathrm{Pdom}(s_1) \tag{59}$$

$$\mathcal{C}(R_1, (\sigma, \rho), v, s_1, s_1(v')) \text{ w.r.t. } \varphi_1 \tag{60}$$

By (49) on (58) we have

$$R_2(\rho) = r \tag{61}$$

Thus (59) and (50) give

$$v' \in \mathrm{Pdom}(s_2) \quad \text{and} \quad s_2(v') = s_1(v') \tag{62}$$

By (60), (48), (49) and (50) we have that the claim $\mathcal{C}(R_2, (\sigma, \rho), v, s_2, s_1(v'))$ w.r.t. φ_2 is a consequence of the lemma, i.e., by (62), that the claim

$$\mathcal{C}(R_2, (\sigma, \rho), v, s_2, s_2(v')) \text{ w.r.t. } \varphi_2 \tag{63}$$

is a consequence of the lemma. Thus Rule 36 on (61), (62) and (63) gives $\mathcal{C}(R_2, (\sigma, \rho), v, s_2, v')$ w.r.t. φ_2, since the premise used is a consequences of the lemma.

$\boxed{\mathbf{get}(\rho) \notin \varphi_2}$ Then $\mathcal{C}(R_2, (\sigma, \rho), v, s_2, v')$ w.r.t. φ_2 by Rule 37.

Environments $\boxed{\mathcal{C}(R, TE, E, s, VE) \text{ w.r.t. } \varphi}$

The case for Rule 38 is straightforward.

8.3 Region renaming

In order to prove that re-use of old regions is safe (Lemma 8.4), we shall want to rename region variables that occur free in some semantic object A but do not occur free in the effect of the rest of the computation, to other region variables that do not occur free in the effect of the rest of the computation. Let S_r be a region substitution. The *yield* of S_r, written $\mathrm{Yield}(S_r)$, is the set $\{S_r(\rho) \mid \rho \in \mathrm{Supp}(S_r)\}$.

Definition 8.1 *Let A be a semantic object, let φ be an effect and let $S = (S_t, S_r, S_e)$ be a substitution. We say that S is* a region renaming of A with respect to φ *if $S \downarrow \mathrm{frv}(A)$ is injective, $(\mathrm{Supp}(S_r) \cup \mathrm{Yield}(S_r)) \cap \mathrm{frv}(\varphi) = \emptyset$ and $\mathrm{Supp}(S_e) = \mathrm{Supp}(S_t) = \emptyset$.*

It is not in general the case that $\mathcal{C}(R, \mu, v, s, v')$w.r.t.$\varphi$ implies $\mathcal{C}(R, S(\mu), v, s, v')$w.r.t. φ, for all substitutions S; the reason is that S might map region variables in the set $\mathrm{frv}(\mu) \setminus \mathrm{frv}(\varphi)$ to variables that are free in φ, thereby making consistency harder to achieve. However, the following special case holds:

Lemma 8.3 *If* $\mathcal{C}(R, \mu, v, s, v')$ w.r.t. φ *and* S *is a region renaming of* μ *with respect to* φ *then* $\mathcal{C}(R, S(\mu), v, s, v')$ w.r.t. φ. *Similarly for the other consistency judgement forms.*

Intuitively: as far as φ is concerned, a region variable $\rho \in \mathrm{frv}(\mu) \setminus \mathrm{frv}(\varphi)$ denotes a garbage region which is no different from any other garbage region!

Proof By rule-based co-induction on $\mathcal{C}(R, \mu, v, s, v')$ w.r.t. φ (and the other consistency judgement forms). The cases are ordered according to judgement forms.

Types and Storable Values $\boxed{\mathcal{C}(R, \mu, v, s, sv) \text{ w.r.t. } \varphi}$

Assume that S is a region renaming of μ with respect to φ and that

$$\mathcal{C}(R, \mu, v, s, sv) \text{ w.r.t. } \varphi \tag{64}$$

Now (64) must be the conclusion of one of the following rules:

Rule 31 By (64) we have $\mu = (\mathtt{int}, \rho)$, for some ρ, and $v = sv \in \mathrm{Int}$. Thus $\mathcal{C}(R, S(\mu), v, s, sv)$ w.r.t. φ.

Rule 32 By (64) there exist TE, x, e, e', R', E, τ, ρ and VE such that

$$TE \vdash \lambda x.e \Rightarrow \lambda x.e' \ \mathtt{at}\ \rho : (\tau, \rho), \{\mathbf{put}(\rho)\} \tag{65}$$

$$\mathcal{C}(R', TE, E', s, VE) \text{ w.r.t. } \varphi \tag{66}$$

$$R' \text{ and } R \text{ agree on } \varphi \quad \mathrm{frv}(e') \subseteq \mathrm{Dom}(R') \tag{67}$$

$$\mu = (\tau, \rho), \quad v = \langle x, e, E \rangle, \quad sv = \langle x, e', VE, R' \rangle \tag{68}$$

where $E' = E$. (The reason for introducing E' will become clear later.) To prove $\mathcal{C}(R, S(\mu), v, s, sv)$ w.r.t. φ we wish to find TE_0, R_0 and e'_0 satisfying

$$TE_0 \vdash \lambda x.e \Rightarrow \lambda x.e'_0 \ \mathtt{at}\ S(\rho) : S(\tau, \rho), \{\mathbf{put}(S(\rho))\} \tag{69}$$

$$\mathcal{C}(R_0, TE_0, E', s, VE) \text{ w.r.t. } \varphi \tag{70}$$

$$R_0 \text{ and } R \text{ agree on } \varphi \quad \mathrm{frv}(e'_0) \subseteq \mathrm{Dom}(R_0) \tag{71}$$

$$sv = \langle x, e'_0, VE, R_0 \rangle \tag{72}$$

and that the claim (70) is itself a consequence of the lemma. Comparing (65) and (69), a tempting idea is simply to apply S throughout (65), taking e'_0 to be $S(e')$. However, S is not necessarily a region renaming on TE, so (70) would not necessarily be a consequence of the lemma.

Therefore, let $\{\rho_1, \ldots, \rho_n\} = \mathrm{frv}(TE) \setminus \mathrm{frv}(\mu, \varphi)$ and let $\{\rho'_1, \ldots, \rho'_n\}$ be distinct new region variables, new in the sense that $\{\rho'_1, \ldots, \rho'_n\} \cap \mathrm{frv}(S(\mu), \varphi) = \emptyset$. Let

$S' = S + \{\rho_i \mapsto \rho'_i \mid 1 \leq i \leq n\}$, let $TE_0 = S'(TE)$ and let $e'_0 = S'(e')$. Then S' is a region renaming of (TE, μ) with respect to φ. Further, R_0 is defined as follows. Let $\mathrm{Dom}(R_0)$ be $\mathrm{frv}(e'_0)$. Since (65) must have been inferred by Rule 23, we have $\mathrm{frv}(e') \subseteq \mathrm{frv}(TE, \tau)$. Thus S' is injective on $\mathrm{frv}(e')$. Then, for every region variable $\rho' \in \mathrm{frv}(e'_0)$ there exists one and only one region variable $\rho \in \mathrm{frv}(e')$ such that $S'(\rho) = \rho'$. Define $R_0(\rho')$ to be $R'(\rho)$. By these definitions, $\langle x, e', VE, R' \rangle$ and $\langle x, e'_0, VE, R_0 \rangle$ are equal. By Lemma 5.3 on (65) and the fact that $S'(\tau, \rho) = S(\tau, \rho)$ we obtain (69), as desired. Notice that R_0 and R' agree on φ, since S' is a region renaming with respect to φ. Thus (71) also holds. Then, by Lemma 8.2 on (66) we have $\mathcal{C}(R_0, TE, E', s, VE)$ w.r.t. φ. But then, since S' is a region renaming of TE with respect to φ we have that the claim (70) is itself a consequence of the lemma, as desired. Finally Rule 32 on (68)–(72) gives $\mathcal{C}(R, S(\mu), v, s, sv)$ w.r.t. φ, as desired.

Rule 33 Almost identical to the previous case: use $E' = E + \{f \mapsto \langle x, e, E, f \rangle\}$ and $v = \langle x, e, E, f \rangle$ instead of $E' = E$ and $v = \langle x, e, E \rangle$. Conclude using Rule 33 instead of using Rule 32.

Type Schemes and Storable Values $\mathcal{C}(R, (\sigma, \rho), v, s, sv)$ w.r.t. φ

Assume that $(\sigma', \rho') = S(\sigma, \rho)$, that S is a region renaming of (σ, ρ) with respect to φ and that

$$\mathcal{C}(R, (\sigma, \rho), v, s, sv) \text{ w.r.t. } \varphi \tag{73}$$

Then (73) is the conclusion of one of the following rules:

Rule 34 Then there exist TE, f, x, e, e', $\rho_1 \ldots \rho_k$, $\alpha_1 \ldots \alpha_n$, $\epsilon_1 \ldots \epsilon_m$, τ, VE and R' such that

$$TE + \{f \mapsto (\sigma, \rho)\} \vdash \lambda x.e \Rightarrow \lambda x.e' \texttt{ at } \rho : (\tau, \rho), \{\mathbf{put}(\rho)\} \tag{74}$$

$$\sigma = \forall \rho_1 \cdots \rho_k \forall \alpha_1 \cdots \alpha_n \forall \epsilon_1 \cdots \epsilon_m . \underline{\tau} \quad \text{and} \quad \mathrm{bv}(\sigma) \cap \mathrm{fv}(TE, \rho) = \emptyset$$

$$R' \text{ and } R \text{ agree on } \varphi \quad \mathrm{frv}(e') \subseteq \mathrm{Dom}(R') \cup \{\rho_1, \ldots, \rho_k\} \tag{75}$$

$$\mathcal{C}(R', TE + \{f \mapsto (\sigma, \rho)\}, E + \{f \mapsto \langle x, e, E, f \rangle\}, s, VE) \text{ w.r.t. } \varphi \tag{76}$$

$$v = \langle x, e, E, f \rangle, \quad sv = \langle \rho_1, \ldots, \rho_k, x, e', VE, R' \rangle \tag{77}$$

As in the previous two cases, S is not necessarily a region renaming of $TE + \{f \mapsto (\sigma, \rho)\}$. Let $\{\rho_1^{old}, \ldots, \rho_l^{old}\} = (\{\rho_1, \ldots, \rho_k\} \cup \mathrm{frv}(TE, \tau)) \setminus \mathrm{frv}((\sigma, \rho), \varphi)$. Let $\{\rho_1^{new}, \ldots, \rho_l^{new}\}$ be distinct new region variables, new in the sense that $\{\rho_1^{new}, \ldots, \rho_l^{new}\} \cap \mathrm{frv}(S(\sigma, \rho), \varphi) = \emptyset$. Let $S' = S + (\{\}, \{\rho_1^{old} \mapsto \rho_1^{new}, \ldots, \rho_l^{old} \mapsto \rho_l^{new}\}, \{\})$. Then

$$S' \text{ is a region renaming on } (\{\rho_1, \ldots, \rho_k\}, TE, \tau, \rho) \text{ with respect to } \varphi \tag{78}$$

Let $TE' = S'(TE)$ and let $e'_0 = S'(e')$. By Lemma 5.3 on (74) we have

$$TE' + \{f \mapsto (S'(\sigma), \rho')\} \vdash \lambda x.e \Rightarrow \lambda x.e'_0 \ \texttt{at} \ \rho' : (S'\tau, \rho'), \{\mathbf{put}(\rho')\} \tag{79}$$

where we have used $S'(\rho) = \rho'$. Since S' is the identity on every type and effect variable, we have

$$S'(\sigma) = \forall S'\rho_1 \cdots S'\rho_k \alpha_1 \cdots \alpha_n \epsilon_1 \cdots \epsilon_m . S'(\tau) \tag{80}$$

Moreover,

$$(\{S'\rho_1, \ldots, S'\rho_k\}, \{\alpha_1, \ldots, \alpha_n\}, \{\epsilon_1, \ldots, \epsilon_m\}) \cap \mathrm{fv}(TE', \rho') = \emptyset \tag{81}$$

since S' is injective on $\mathrm{frv}(\{\rho_1, \ldots, \rho_k\}, TE, \rho)$. Define R_0 as follows. Let $\mathrm{Dom}(R_0) = \mathrm{frv}(e'_0) \setminus \{S'(\rho_1), \ldots, S'(\rho_k)\}$. From (74) and Rule 23 we get $\mathrm{frv}(e') \subseteq \mathrm{frv}(TE + \{f \mapsto (\sigma, \rho)\}, \tau)$. By (78), for every $\rho' \in e'_0$ there exists a unique $\rho \in \mathrm{frv}(e')$ such that $S'(\rho) = \rho'$. Let $R_0(\rho') = R'(\rho)$. The closures $\langle \rho_1, \ldots, \rho_k, x, e', VE, R' \rangle$ and $\langle S'\rho_1, \ldots, S'\rho_k, x, e'_0, VE, R_0 \rangle$ are now equal. Moreover, by (78), R_0 and R' agree on φ. But then, by (75), we have

$$R_0 \text{ and } R \text{ agree on } \varphi \quad \mathrm{frv}(e'_0) \subseteq \mathrm{Dom}(R_0) \cup \{S'\rho_1, \ldots, S'\rho_k\} \tag{82}$$

By Lemma 8.2 on (76), using that R_0 and R' agree on φ, we get

$$\mathcal{C}(R_0, TE + \{f \mapsto (\sigma, \rho)\}, E + \{f \mapsto \langle x, e, E, f \rangle\}, s, VE) \text{ w.r.t. } \varphi \tag{83}$$

Notice that S' is a region renaming of $TE + \{f \mapsto (\sigma, \rho)\}$ with respect to φ. Thus from (83) get get that the claim

$$\mathcal{C}(R_0, TE' + \{f \mapsto (S'(\sigma), \rho')\}, E + \{f \mapsto v\}, s, VE) \text{ w.r.t. } \varphi \tag{84}$$

is a consequence of the lemma. By Rule 34 on (79), (80), (81), (82) and (84) we have

$$\mathcal{C}(R, (S'(\sigma), p'), \langle x, e, E, f \rangle, s, \langle S'\rho_1, \ldots, S'\rho_k, x, e'_0, VE, R_0 \rangle) \text{ w.r.t. } \varphi \tag{85}$$

which is the desired result

Rule 35 By (73) and Rule 35 we have that σ is simple and takes the form $\forall().\tau$ and $\mathcal{C}(R, (\tau, \rho), v, s, sv)$ w.r.t. φ. Thus the claim $\mathcal{C}(R, S'(\tau, \rho), v, s, sv)$ w.r.t. φ is a consequence of the lemma. Thus $\mathcal{C}(R, (S'(\sigma), \rho'), v, s, sv)$ w.r.t. φ, as desired.

The cases for the remaining rules (Rules 36–38) are straightforward. □

8.4 Region allocation

Consistency is not in general preserved under increasing effects or decreasing stores. For example, for all addresses a, we have $\mathcal{C}(\{\rho \mapsto \mathtt{r}\}, (\mathtt{int}, \rho), 3, \{\}, a)$w.r.t. φ if $\varphi = \emptyset$, but not if $\varphi = \{\mathbf{get}(\rho)\}$, since the store is empty. Yet there is one point where we do need to increase effects, namely in the case of the main proof concerning expressions of the form

$$e' \equiv \mathtt{letregion}\ \rho\ \mathtt{in}\ e'_1\ \mathtt{end}$$

Starting from an assumption of the form $\mathcal{C}(R, TE, E, s, VE)$ w.r.t. φ we wish to extend s with a new region, yielding $s' = s + \{r \mapsto \{\}\}$, increase φ to $\varphi \cup \{\mathbf{put}(\rho), \mathbf{get}(\rho)\}$ (the get and put effects representing the effects of e'_1 on the new region) and still be able to claim $\mathcal{C}(R + \{\rho \mapsto r\}, TE, E, s', VE)$ w.r.t. $\varphi \cup \{\mathbf{put}(\rho), \mathbf{get}(\rho)\}$. That this is possible is not trivial, for the region r may have been in use earlier (and there may even be dead pointers into the old region named r). However, if we extend the observing effect with a region variable which is not free in the type environment, then consistency really *is* preserved:

Lemma 8.4 *If* $\mathcal{C}(R, TE, E, s, VE)$ w.r.t. φ *and* $\rho \notin \mathrm{frv}(TE, \varphi)$, $r \notin \mathrm{Dom}(s)$ *and* $\mathrm{frv}(\varphi') \subseteq \{\rho\}$ *then* $\mathcal{C}(R+\{\rho \mapsto r\}, TE, E, s+\{r \mapsto \{\}\}, VE)$w.r.t.$\varphi' \cup \varphi$. *Similarly for the other forms of* $\mathcal{C}$.

Proof The proof is by rule-based co-induction. We assume

$$\mathrm{frv}(\varphi') \subseteq \{\rho\} \tag{86}$$

$$r \notin \mathrm{Dom}(s) \tag{87}$$

For brevity, let $s' = s + \{r \mapsto \{\}\}$. We now have a case analysis with one case for each of Rules 31 to 38.

Types and Storable Values $\boxed{\mathcal{C}(R, \mu, v, s, sv) \text{ w.r.t. } \varphi}$

Assume

$$\mathcal{C}(R, (\tau, \rho_0), v, s, sv) \text{ w.r.t. } \varphi \tag{88}$$

$$\rho \notin \mathrm{frv}((\tau, \rho_0), \varphi) \tag{89}$$

Then (88) is the conclusion of one of the following rules:

Rule 31 Here $v = sv = i$, for some $i \in \mathrm{Int}$ and $\tau = \mathtt{int}$. Hence $\mathcal{C}(R + \{\rho \mapsto r\}, (\tau, \rho_0), v, s', sv)$ w.r.t. $\varphi \cup \varphi'$ by Rule 31 itself.

Rule 32 Here (88) is inferred from premises

$$TE \vdash \lambda x.e \Rightarrow \lambda x.e' \texttt{ at } \rho_0 : (\tau, \rho_0) : \{\mathbf{put}(\rho_0)\} \tag{90}$$

$$\mathcal{C}(R_0, TE, E, s, VE) \text{ w.r.t. } \varphi \tag{91}$$

$$R_0 \text{ and } R \text{ agree on } \varphi \quad \text{frv}(e') \subseteq \text{Dom}(R_0) \tag{92}$$

$$v = \langle x, e, E\rangle \quad \text{and} \quad sv = \langle x, e', VE, R_0\rangle \tag{93}$$

Without loss of generality we can assume

$$\rho \notin \text{frv}(TE) \tag{94}$$

for if $\rho \in \text{frv}(TE)$ we can do the following. Let ρ' be a fresh region variable, fresh in the sense that $\rho' \notin \text{frv}(TE, \varphi, \tau)$. Consider the substitution $S = \{\rho \mapsto \rho'\}$. By (89) and Lemma 5.3 on (90) we have

$$S(TE) \vdash \lambda x.e \Rightarrow \lambda x.S(e') \texttt{ at } \rho_0 : (\tau, \rho_0) : \{\mathbf{put}(\rho_0)\} \tag{95}$$

Moreover, S is a region renaming of TE with respect to φ, so Lemma 8.3 on (91) gives

$$\mathcal{C}(R_0, S(TE), E, s, VE) \text{ w.r.t. } \varphi \tag{96}$$

Let R_0' be the region environment defined as follows. If $\rho \notin \text{Dom}(R_0)$ then let $R_0' = R_0$. Otherwise let R_0' have domain $\text{Dom}(R_0') = \text{Dom}(R_0) \setminus \{\rho\} \cup \{\rho'\}$ and values

$$R_0'(\rho_0') = \begin{cases} R_0(\rho_0') & \text{if } \rho_0' \neq \rho \\ R_0(\rho) & \text{if } \rho_0' = \rho' \end{cases}$$

Let $sv' = \langle x, S(e'), VE, R_0'\rangle$. Since $\text{frv}(e') \subseteq \text{Dom}(R_0)$ we have that sv and sv' are equal and $\text{frv}(S(e')) \subseteq \text{Dom}(R_0')$. Also, R_0' and R_0 agree on φ (since neither ρ' nor ρ is free in φ). Thus by Lemma 8.2 on (96) we have

$$\mathcal{C}(R_0', S(TE), E, s, VE) \text{ w.r.t. } \varphi \tag{97}$$

Thus we can assume that (94) holds.

By (91) and (94) we have that the claim

$$\mathcal{C}(R_0 + \{\rho \mapsto r\}, TE, E, s', VE) \text{ w.r.t. } \varphi \cup \varphi' \tag{98}$$

is itself a conclusion of the lemma. Moreover, from (92) and (86) we have

$$R_0 + \{\rho \mapsto r\} \text{ and } R + \{\rho \mapsto r\} \text{ agree on } \varphi \cup \varphi' \tag{99}$$

By Rule 32 on (90), (98), (99) and the fact that $\text{frv}(e') \subseteq \text{Dom}(R_0 + \{\rho \mapsto r\})$ we get

$$\mathcal{C}(R + \{\rho \mapsto r\}, (\tau, \rho_0), v, s', sv') \text{ w.r.t. } \varphi \cup \varphi' \tag{100}$$

where $sv' = \langle x, e', VE, R_0 + \{\rho \mapsto r\}\rangle$. By (90) and Rule 23 we have $\text{frv}(e') \subseteq \text{frv}(TE, \tau)$ so by (89) and (94) we have $\rho \notin \text{frv}(e')$. Thus sv and sv' are equal; thus (100) is the desired result.

Rule 33 Similar to the previous case.

Type Schemes and Storable Values $\boxed{\mathcal{C}(R,(\sigma,\rho_0),v,s,sv) \text{ w.r.t. } \varphi}$

Assume

$$\mathcal{C}(R,(\sigma,\rho_0),v,s,sv) \text{ w.r.t. } \varphi \tag{101}$$

$$\rho \notin \mathrm{frv}((\sigma,\rho_0),\varphi) \tag{102}$$

where (101) must be the conclusion of one of the following rules:

Rule 34 Here σ is compound and there exist TE, f, x, e, $\rho_1,\ldots,\rho_k$, $\alpha_1,\ldots,\alpha_n$, $\epsilon_1,\ldots,\epsilon_m$, R_0 and VE such that

$$TE + \{f \mapsto (\sigma,\rho_0)\} \vdash \lambda x.e \Rightarrow \lambda x.e' \texttt{ at } \rho_0 : (\tau,\rho_0),\{\mathbf{put}(\rho_0)\} \tag{103}$$

$$\sigma = \forall\rho_1\cdots\rho_k\forall\alpha_1\cdots\alpha_n\forall\epsilon_1\cdots\epsilon_m.\underline{\tau} \qquad \mathrm{bv}(\sigma)\cap\mathrm{fv}(TE,\rho_0) = \emptyset \tag{104}$$

$$R_0 \text{ and } R \text{ agree on } \varphi \quad \mathrm{frv}(e') \subseteq \mathrm{Dom}(R_0)\cup\{\rho_1,\ldots,\rho_k\} \tag{105}$$

$$\mathcal{C}(R_0, TE + \{f \mapsto (\sigma,\rho_0)\}, E + \{f \mapsto \langle x,e,E,f\rangle\}, s, VE) \text{ w.r.t. } \varphi \tag{106}$$

$$v = \langle x,e,E,f\rangle \quad \text{and} \quad sv = \langle \rho_1,\ldots,\rho_k,x,e',VE,R_0\rangle \tag{107}$$

As in the case for Rule 32 we may assume

$$\rho \notin \mathrm{frv}(TE + \{x \mapsto (\sigma,\rho_0)\}) \tag{108}$$

without loss of generality. By (106) and (108) we get that the claim

$$\mathcal{C}(R_0 + \{\rho \mapsto r\}, TE + \{f \mapsto (\sigma,\rho_0)\}, E + \{f \mapsto \langle x,e,E,f\rangle\}, s', VE) \text{ w.r.t. } \varphi\cup\varphi' \tag{109}$$

is a consequence of the lemma. Let $R_0' = R_0 + \{\rho \mapsto r\}$ and let $R' = R + \{\rho \mapsto r\}$. By (105) and (102) we have

$$R_0' \text{ and } R' \text{ agree on } \varphi\cup\varphi' \tag{110}$$

Thus by Rule 34 on (103), (110) and (109) we have

$$\mathcal{C}(R',(\sigma,\rho_0),v,s',\langle\rho_1,\ldots,\rho_k,x,e',VE,R_0'\rangle) \text{ w.r.t. } \varphi\cup\varphi' \tag{111}$$

From (103) and Rule 23 we have $\mathrm{frv}(e') \subseteq \mathrm{frv}(TE + \{f \mapsto (\sigma,\rho_0)\},\tau)$. This with (108) gives that if $\rho \in \mathrm{frv}(e')$ then $\rho \in \{\rho_1,\ldots,\rho_k\}$. Thus sv and $\langle\rho_1,\ldots,\rho_k,x,e',VE,R_0'\rangle$ are equal, so (111) really is the desired result.

Rule 35 Here σ is simple. Write σ in the form $\forall().\tau$. Then $\rho \notin \mathrm{frv}((\tau,\rho_0),\varphi)$, by (102). By (101) and Rule 35 we have $\mathcal{C}(R,(\tau,\rho_0),v,s,sv)$ w.r.t. φ. But then the claim $\mathcal{C}(R + \{\rho \mapsto r\},(\tau,\rho_0),v,s',sv)$ w.r.t. $\varphi\cup\varphi'$ is a consequence of the lemma. Thus $\mathcal{C}(R + \{\rho \mapsto r\},(\sigma,\rho_0),v,s',sv)$ w.r.t. $\varphi\cup\varphi'$, by Rule 35.

Type Schemes and addresses $\boxed{\mathcal{C}(R,(\sigma,\rho_0),v,s,v')\text{ w.r.t. }\varphi}$

Assume

$$\mathcal{C}(R,(\sigma,\rho_0),v,s,v')\text{ w.r.t. }\varphi \tag{112}$$

$$\rho \notin \mathrm{frv}(\sigma,\rho_0,\varphi) \tag{113}$$

Then (112) is the conclusion of one of the following rules:

Rule 36 Here $R(\rho_0) = r$ *of* v', $v' \in \mathrm{Pdom}(s)$ and

$$\mathcal{C}(R,(\sigma,\rho_0),v,s,s(v'))\text{ w.r.t. }\varphi \tag{114}$$

By (113) we have $(R+\{\rho \mapsto r\})(\rho_0) = R(\rho_0) = r$ *of* v'. Since $r \notin \mathrm{Dom}(s)$ we have $v' \in \mathrm{Pdom}(s')$ and $s'(v') = s(v')$. By (114) and (113) we have that the claim $\mathcal{C}(R+\{\rho \mapsto r\},(\sigma,\rho_0),v,s',s'(v'))$ w.r.t. $\varphi \cup \varphi'$ is a consequence of the lemma. Then, by Rule 36, we have $\mathcal{C}(R+\{\rho \mapsto r\},(\sigma,\rho_0),v,s',v')$ w.r.t. $\varphi \cup \varphi'$, as desired.

Rule 37 Since $\mathbf{get}(\rho_0) \notin \varphi$ and (86) and, by (113), $\rho \neq \rho_0$, we have $\mathbf{get}(\rho_0) \notin \varphi \cup \varphi'$. Thus $\mathcal{C}(R+\{\rho \mapsto r\},(\sigma,\rho_0),v,s',v')$ w.r.t. $\varphi \cup \varphi'$, by Rule 37 itself.

Environments $\boxed{\mathcal{C}(R,TE,E,s,VE)\text{ w.r.t. }\varphi}$

The case for Rule 38 is straightforward. □

Lemma 8.5 *If* $\mathcal{C}(R,TE,E,s,VE)$ w.r.t. φ *then* $\mathcal{C}(R,TE,E,s,VE)$ w.r.t. $\varphi \cup \{\epsilon\}$. *Similarly for the other forms of* $\mathcal{C}$.

Proof Straightforward co-inductive proof.

8.5 Recursion

The source and target languages handle recursion differently. The source language "unrolls" a closure each time a recursive function is applied — see Rule 5. In the target language a closure for a recursive function contains a pointer back to itself — see Rule 14. To prove the correctness of our translation, we must show that the two representations are consistent at the point where we create the cycle in the store.

Lemma 8.6 *If* $\mathcal{C}(R,TE,E,s,VE)$ w.r.t. φ *and* σ *is a compound type scheme* $\forall \vec{\rho}\,\vec{\alpha}\,\vec{\epsilon}.\underline{\tau}$, *with* $\mathrm{bv}(\sigma) \cap \mathrm{fv}(TE,\rho) = \emptyset$, *and* $TE + \{f \mapsto (\sigma,\rho)\} \vdash \lambda x.e \Rightarrow \lambda x.e'\,\mathtt{at}\,\rho : (\tau,\rho),\{\mathbf{put}(\rho)\}$ *and* R' *and* R *agree on* φ *and* $\mathrm{frv}(e') \subseteq \mathrm{Dom}(R') \cup \mathrm{frv}(\vec{\rho})$ *and* $R(\rho) = r$ *and* $r \in \mathrm{Dom}(s)$ *and* $o \notin \mathrm{Dom}(s(r))$ *then*

$$\mathcal{C}(\quad R,\ TE + \{f \mapsto (\sigma,\rho)\},\ E + \{f \mapsto \langle x,e,E,f\rangle\},\\ s + \{(r,o) \mapsto \langle \vec{\rho},x,e',VE',R'\rangle\},\ VE')\text{ w.r.t. }\varphi$$

where $VE' = VE + \{f \mapsto (r,o)\}$.

Proof Let $TE' = TE + \{f \mapsto (\sigma, \rho)\}$, $E' = E + \{f \mapsto \langle x, e, E, f\rangle\}$, $VE' = VE + \{f \mapsto (r, o)\}$ and $s' = s + \{(r, o) \mapsto \langle \vec{\rho}, x, e', VE', R'\rangle\}$. By Lemma 8.2 it suffices to prove

$$\mathcal{C}(R', TE', E', s', VE') \text{ w.r.t. } \varphi$$

The proof is by co-induction. Let

$$\begin{aligned} q_1 &= (R', (\sigma, \rho), \langle x, e, E, f\rangle, s', \langle \vec{\rho}, x, e', VE', R'\rangle, \varphi) \\ q_2 &= (R', (\sigma, \rho), \langle x, e, E, f\rangle, s', (r, o), \varphi) \\ q_3 &= (R', TE', E', s', VE', \varphi) \end{aligned}$$

Let $\Gamma' = \Gamma^{\#} \cup \{q_1, q_2, q_3\}$ and show $\Gamma' \subseteq \mathcal{F}(\Gamma')$. We consider q_1, q_2 and q_3 in turn.

$\boxed{q_1}$ Since $q_3 \in \Gamma'$ and $\sigma = \forall \vec{\rho}\, \vec{\alpha}\, \vec{\epsilon}.\underline{\tau}$, with $\text{bv}(\sigma) \cap \text{fv}(TE, \rho) = \emptyset$, and $TE + \{f \mapsto (\sigma, \rho)\} \vdash \lambda x.e \Rightarrow \lambda x.e'$ at $\rho : (\tau, \rho), \{\mathbf{put}(\rho)\}$ and R' agrees with itself on φ and $\text{frv}(e') \subseteq \text{Dom}(R') \cup \text{frv}(\vec{\rho})$ we have $q_1 \in \mathcal{F}(\Gamma')$, by rule 42.

$\boxed{q_2}$ If $\mathbf{get}(\rho) \notin \varphi$ then $q_2 \in \mathcal{F}(\Gamma')$, by Rule 45. Assume $\mathbf{get}(\rho) \in \varphi$. Since R and R' agree on φ we have $R'(\rho) = R(\rho) = r$. Since also $r \in \text{Dom}(s')$ and $q_1 \in \Gamma'$ we have $q_2 \in \mathcal{F}(\Gamma')$, by rule 44.

$\boxed{q_3}$ By Lemma 8.2 on $\mathcal{C}(R, TE, E, s, VE)$w.r.t.$\varphi$ we have $\mathcal{C}(R', TE, E, s', VE)$w.r.t. φ. Thus $\text{Dom}(TE) = \text{Dom}(E) = \text{Dom}(VE)$ and for every $x \in \text{Dom}(TE)$ we have $\mathcal{C}(R', TE(x), E(x), s', VE(x))$w.r.t.$\varphi$, i.e., for $x \neq f$, $\mathcal{C}(R', TE'(x), E'(x), s', VE'(x))$w.r.t. φ. Since also $q_2 \in \Gamma'$ we have $q_3 \in \mathcal{F}(\Gamma')$ by Rule 46.

9 Proof of the Correctness of the Translation

This section is the proof of Theorem 6.1. The proof is by depth of the derivation of $E \vdash e \to v$, each with an inner induction on the depth of inference of $TE \vdash e \Rightarrow e' : \mu, \varphi$. There are 7 cases, one for each rule in the dynamic semantics of the source language. For each of these cases, the inner induction consists of a base case, in which $TE \vdash e \Rightarrow e' : \mu, \varphi$ was inferred by one of the *syntax-directed* rules (i.e., rules 20 – 26) plus an inductive step, where Rule 27 or 28 was applied. It turns out the the inner inductive steps are independent of e, so we start out by doing them once and for all. Then we deal with each of the 7 syntax-directed cases.

In all the cases, we assume

$$TE \vdash e \Rightarrow e' : \mu, \varphi \tag{115}$$

$$\mathcal{C}(R, TE, E, s, VE) \text{ w.r.t. } \varphi \cup \varphi' \tag{116}$$

$$E \vdash e \to v \tag{117}$$

$$R \text{ connects } \varphi \cup \varphi' \text{ to } s \tag{118}$$

$$R' \text{ and } R \text{ agree on } \varphi \cup \varphi' \tag{119}$$

$$\mathrm{frv}(e') \subseteq \mathrm{Dom}\, R' \tag{120}$$

Inner inductive step (a): Rule 27 was applied Assume (115) takes the form

$$TE \vdash e \Rightarrow \texttt{letregion}\ \rho\ \texttt{in}\ e'_1\ \texttt{end} : \mu, \varphi \tag{121}$$

and is inferred by Rule 27 from premises

$$TE \vdash e \Rightarrow e'_1 : \mu, \varphi^+ \tag{122}$$

$$\varphi = \varphi^+ \setminus \{\mathbf{put}(\rho), \mathbf{get}(\rho)\} \tag{123}$$

$$\rho \notin \mathrm{frv}(TE, \mu) \tag{124}$$

By Lemma 5.3 we can choose ρ such that $\rho \notin \mathrm{frv}(\varphi')$ as well as (123)-(124). Thus $\rho \notin \mathrm{frv}(TE, \varphi \cup \varphi')$. Let r be an address satisfying $r \notin \mathrm{Dom}(s)$. Let $R^+ = R + \{\rho \mapsto r\}$ and $s^+ = s + \{r \mapsto \{\}\}$. Then by Lemma 8.4 on (116) we get

$$\mathcal{C}(R^+, TE, E, s^+, VE) \text{ w.r.t. } \varphi^+ \cup \varphi' \tag{125}$$

Let $R'^+ = R' + \{\rho \mapsto r\}$. By (118) we have

$$R^+ \text{ connects } \varphi^+ \cup \varphi' \text{ to } s^+ \tag{126}$$

and by (119)

$$R'^+ \text{ and } R^+ \text{ agree on } \varphi^+ \cup \varphi' \tag{127}$$

By (120) we have

$$\mathrm{frv}(e'_1) \subseteq \mathrm{Dom}\, R'^+ \tag{128}$$

By the inner induction applied to (122), (125), (117), (126), (127) and (128) there exist s'_1 and v' such that

$$s^+, VE, R'^+ \vdash e'_1 \to v', s'_1 \tag{129}$$

$$\mathcal{C}(R'^+, \mu, v, s'_1, v') \text{ w.r.t. } \varphi' \tag{130}$$

Let $s' = s'_1 \setminus\setminus \{r\}$. Rule 15 on (129) gives

$$s, VE, R' \vdash \texttt{letregion}\ \rho\ \texttt{in}\ e'_1\ \texttt{end} \to v', s'$$

Note that R'^+ and R' agree on φ' (as $\rho \notin \mathrm{frv}(\varphi')$). Also, $s'_1 \downarrow (\mathrm{Rng}(R' \downarrow \mathrm{frv}(\varphi'))) \sqsubseteq s'$ by (118) and (119). Then by Lemma 8.2 on (130) we get $\mathcal{C}(R', \mu, v, s', v')$ w.r.t. φ', as required.

Inner inductive step (b): Rule 28 was applied | Assume (115) is inferred by Rule 28 on premises $TE \vdash e \Rightarrow e' : \mu, \varphi^+, \varphi = \varphi^+ \setminus \{\epsilon\}$ and $\epsilon \notin \mathrm{fev}(TE, \mu)$. By Lemma 8.5 on (116) we get $\mathcal{C}(R, TE, E, s, VE)$ w.r.t. $\varphi^+ \cup \varphi'$. Also, R connects $\varphi^+ \cup \varphi'$ to s; R' and R agree on $\varphi^+ \cup \varphi'$ and $\mathrm{frv}(e') \subseteq \mathrm{Dom}(R')$. Thus by the inner induction there exist s' and v' such that $s, VE, R' \vdash e' \to v', s'$ and $\mathcal{C}(R', \mu, v, s', v')$ w.r.t. φ', as desired.

The syntax-directed cases:

Constant, Rule 1 | Since R connects $\{\mathbf{put}(\rho)\} \cup \varphi'$ to s and R' and R agree on $\{\mathbf{put}(\rho)\} \cup \varphi'$ we have that $r = R'(\rho)$ exists and $r \in \mathrm{Dom}(s)$. Take $o \notin \mathrm{Dom}(s(r))$. By Rule 8 we then have $s, VE, R' \vdash c \ \mathtt{at}\ \rho \to (r, o), s + \{(r, o) \mapsto c\}$. Letting $v' = (r, o)$ and $s' = s + \{(r, o) \mapsto c\}$ we furthermore get $\mathcal{C}(R', (\mathtt{int}, \rho), v, s', v')$ w.r.t. φ', by (36), (35) and (31), as desired.

Variable, Rule 2 | There are two cases, depending on whether TE associates a simple or a compound type scheme with the variable. We deal with each of these in turn:

Variable with simple type scheme | Assume (115) was inferred using Rule 21. Then $e = e' = x$, for some variable x. Moreover, $TE(x) = (\sigma, p)$, for some p and simple σ. Let τ be the type for which $\sigma = \forall().\tau$. Then $\mu = (\tau, p)$ and $\varphi = \emptyset$. The evaluation (117) must have been by Rule 2, so we have $v = E(x)$. Let $s' = s$. By (115) and (116) we have $x \in \mathrm{Dom}(VE)$. Thus, letting $v' = VE(x)$, we have $s, VE, R' \vdash x \to v', s'$, as desired. By Rule 38 on (116) we have $\mathcal{C}(R, (\sigma, p), v, s', v')$ w.r.t. φ', i.e., $\mathcal{C}(R, (\tau, p), v, s', v')$ w.r.t. φ', as desired (recall that we identify $\forall().\tau$ and τ).

Variable with compound type scheme | Assume (115) was obtained by Rule 22. Then e is a variable, f, and e' is of the form $f\mathtt{[}S(\rho_1)\mathtt{,}\ldots\mathtt{,}S(\rho_k)\mathtt{]}\ \mathtt{at}\ p$ and $\mu = (\tau, p)$, for some τ, and

$$TE \vdash f \Rightarrow f\mathtt{[}S\rho_1\mathtt{,}\ldots\mathtt{,}S\rho_k\mathtt{]}\ \mathtt{at}\ p : (\tau, p), \varphi \tag{131}$$

was inferred by application of Rule 22 to the premises

$$TE(f) = (\sigma, p') \quad \sigma = \forall\rho_1 \cdots \rho_k \vec{\alpha}\vec{\epsilon}.\underline{\tau_1} \tag{132}$$

$$\sigma \geq \tau \text{ via } S \tag{133}$$

$$\varphi = \{\mathbf{get}(p'), \mathbf{put}(p)\} \tag{134}$$

Then (117) must have been inferred by Rule 2, so we have $v = E(f)$. By (116) and $f \in \mathrm{Dom}(TE)$ we have

$$\mathcal{C}(R, (\sigma, p'), v, s, v_1') \text{ w.r.t. } \varphi \cup \varphi'$$

where $v_1' = VE(f)$. Since $\mathbf{get}(p') \in \varphi$, the definition of $\mathcal{C}$ (rules 36 and 34) gives $v_1' \in \mathrm{Pdom}(s)$ and r *of* $v_1' = R(p')$ and v is a recursive closure

$$v = \langle x_0, e_0, E_0, f_0 \rangle \tag{135}$$

and $s(v_1') = \langle \rho_1, \ldots, \rho_k, x_0, e_0', VE_0, R_0 \rangle$, for some e_0', VE_0 and R_0. Furthermore, there exist TE_0, $\alpha_1, \ldots, \alpha_n$, $\epsilon_1, \ldots, \epsilon_m$ and τ_0 such that

$$\mathcal{C}(R_0, TE_0 + \{f_0 \mapsto (\sigma, p')\}, E_0 + \{f_0 \mapsto v\}, s, VE_0) \text{ w.r.t. } \varphi \cup \varphi' \tag{136}$$

$$TE_0 + \{f_0 \mapsto (\sigma, p')\} \vdash \lambda x_0.e_0 \Rightarrow \lambda x_0.e_0' \texttt{ at } p' : (\tau_0, p'), \{\mathbf{put}(p')\} \tag{137}$$

$$\mathrm{bv}(\sigma) \cap \mathrm{fv}(TE_0, p') = \emptyset \tag{138}$$

$$R_0 \text{ and } R \text{ agree on } \varphi \cup \varphi' \tag{139}$$

$$\mathrm{frv}(e_0') \subseteq \mathrm{Dom}\, R_0 \cup \{\rho_1, \ldots, \rho_k\} \tag{140}$$

Without loss of generality, we can assume that $\rho_1, \ldots, \rho_k$ are chosen so as to satisfy:

$$\{\rho_1, \ldots, \rho_k\} \cap \mathrm{frv}(\varphi') = \emptyset \tag{141}$$

By (134), (118) and (119) we have $R'(p) \in \mathrm{Dom}(s)$. Let $r' = R'(p)$. Let o' be an offset not in $\mathrm{Dom}(s(r'))$. Let $v' = (r', o')$, let $R_0' = R_0 + \{\rho_i \mapsto R'(S(\rho_i)) \,;\, 1 \leq i \leq k\}$ and let $sv = \langle x_0, e_0', VE_0, R_0' \rangle$. Notice that $R'(S(\rho_i))$ exists, by (120). Let $s' = s + \{(r', o') \mapsto sv\}$. It follows from Rule 10 that

$$s, VE, R' \vdash f \texttt{ [} S(\rho_1), \texttt{ ... , } S(\rho_k)\texttt{] at } p \to v', s' \tag{142}$$

as desired. It remains to prove

$$\mathcal{C}(R', (\tau, p), v, s', v') \text{ w.r.t. } \varphi' \tag{143}$$

We now consult Rules 31–38 concerning $\mathcal{C}$. If $\mathbf{get}(p) \notin \varphi'$, we are done. But even if $\mathbf{get}(p) \in \varphi'$ we have $v' \in \mathrm{Pdom}(s')$ and r *of* $v' = r' = R'(p)$ as required by Rule 36. It remains to prove

$$\mathcal{C}(R', (\tau, p), v, s', sv) \text{ w.r.t. } \varphi' \tag{144}$$

Let $TE = TE_0 + \{f_0 \mapsto (\sigma, p')\}$. Since (137) must have been inferred by Rules 23 and 28 we equally have

$$TE \vdash \lambda x_0.e_0 \Rightarrow \lambda x_0.e_0' \texttt{ at } p : (\tau, p), \{\mathbf{put}(p)\} \tag{145}$$

From (119), (139) and $\{\rho_1, \ldots, \rho_k\} \cap \mathrm{frv}(\varphi') = \emptyset$ we get

$$R_0' \text{ and } R' \text{ agree on } \varphi' \tag{146}$$

From Lemma 8.2 on (136) we get

$$\mathcal{C}(R_0', TE, E_0 + \{f_0 \mapsto v\}, s', VE_0) \text{ w.r.t. } \varphi' \tag{147}$$

From (140) we get

$$\mathrm{frv}(e_0') \subseteq \mathrm{Dom}\, R_0' \tag{148}$$

By Rule 33 on (145), (146), (147) and (148) we have $\mathcal{C}(R', (\tau, p), v, s', \langle x_0, e', VE_0, R_0' \rangle)$w.r.t. φ' as desired.

Lambda abstraction, Rule 3 Assume (115) was inferred by Rule 23; then (115) takes the following form:

$$TE \vdash \lambda x.e_1 \Rightarrow \lambda x.e_1' \texttt{ at } p : \mu, \{\mathbf{put}(p)\} \tag{149}$$

Moreover, (117) was inferred by Rule 3 yielding

$$v = \langle x, e_1, E \rangle \tag{150}$$

Since R connects φ to s we have $R(p) \in \mathrm{Dom}(s)$. Let $r = R(p)$ and let o be an offset not in $\mathrm{Dom}(s(r))$. Let $v' = (r, o)$ and $s' = s + \{v' \mapsto \langle x, e_1', VE, R' \rangle\}$. By (119) we have $R'(p) = r$. Thus by rule 11 we have

$$s, VE, R' \vdash \lambda x.e_1' \texttt{ at } p \to v', s' \tag{151}$$

Notice that $\mathcal{C}(R', TE, E, s', VE)$w.r.t.$\varphi'$, by Lemma 8.2 and (119). Also $\mathrm{frv}(e_1') \subseteq \mathrm{Dom}\, R'$, by (120). Thus by Rules 32, 35 and 36 (or by (37)) we have $\mathcal{C}(R, \mu, v, s', v')$w.r.t. φ' as required.

Application of non-recursive closure, Rule 4 Here $e \equiv e_1 e_2$, for some e_1 and e_2, and $e' \equiv e_1' e_2'$, for some e_1' and e_2' and (115) was inferred by Rule 24 on premises

$$TE \vdash e_1 \Rightarrow e_1' : (\mu' \xrightarrow{\epsilon.\varphi_0} \mu, p), \varphi_1 \tag{152}$$

$$TE \vdash e_2 \Rightarrow e_2' : \mu', \varphi_2 \tag{153}$$

$$\varphi = \varphi_1 \cup \varphi_2 \cup \{\epsilon, \mathbf{get}(p)\} \cup \varphi_0 \tag{154}$$

Moreover, (117) was inferred by Rule 4 on premises

$$E \vdash e_1 \to v_1, \quad v_1 = \langle x_0, e_0, E_0 \rangle \tag{155}$$

$$E \vdash e_2 \to v_2 \tag{156}$$

$$E_0 + \{x_0 \mapsto v_2\} \vdash e_0 \to v \tag{157}$$

Let $\varphi_1' = \varphi_2 \cup \{\epsilon, \mathbf{get}(p)\} \cup \varphi_0 \cup \varphi'$, i.e. the effect that remains after the computation of e_1'. Note that $\varphi \cup \varphi' = \varphi_1 \cup \varphi_1'$ so from (116), (118) and (119) we get

$$\mathcal{C}(R, TE, E, s, VE) \text{ w.r.t. } \varphi_1 \cup \varphi_1' \tag{158}$$

$$R \text{ connects } \varphi_1 \cup \varphi_1' \text{ to } s \tag{159}$$

$$R' \text{ and } R \text{ agree on } \varphi_1 \cup \varphi_1' \tag{160}$$

Also, from (120) we get

$$\mathrm{frv}(e_1') \subseteq \mathrm{Dom}\, R' \wedge \mathrm{frv}(e_2') \subseteq \mathrm{Dom}\, R' \tag{161}$$

By induction on (152), (158), (155), (159), (160) and (161) there exist s_1 and v_1' such that

$$s, VE, R' \vdash e_1' \to v_1', s_1 \tag{162}$$

$$\mathcal{C}(R', (\mu' \xrightarrow{\epsilon.\varphi_0} \mu, p), v_1, s_1, v_1') \text{ w.r.t. } \varphi_1' \tag{163}$$

Notice that $\mathbf{get}(p) \in \varphi_1'$. Thus, by the definition of $\mathcal{C}$, (163) tells us that $v_1' \in \mathrm{Pdom}(s_1)$ and r *of* $v_1' = R'(p)$ and there exist e_0', VE_0, TE_0 and R_0 such that

$$s_1(v_1') = \langle x_0, e_0', VE_0, R_0 \rangle \tag{164}$$

$$TE_0 \vdash \lambda x_0.e_0 \Rightarrow \lambda x_0.e_0' \texttt{ at } p : (\mu' \xrightarrow{\epsilon.\varphi_0} \mu, p), \{\mathbf{put}(p)\} \tag{165}$$

$$\mathcal{C}(R_0, TE_0, E_0, s_1, VE_0) \text{ w.r.t. } \varphi_1' \tag{166}$$

$$R_0 \text{ and } R' \text{ agree on } \varphi_1' \tag{167}$$

$$\mathrm{frv}(e_0') \subseteq \mathrm{Dom}\, R_0 \tag{168}$$

Let $\varphi_2' = \{\epsilon, \mathbf{get}(p)\} \cup \varphi_0 \cup \varphi'$, i.e. the effect that remains after the computation of e_2'. By Lemma 4.1 on (162) we have $s \sqsubseteq s_1$. Furthermore, we have $\varphi_2 \cup \varphi_2' \subseteq \varphi \cup \varphi'$, so by Lemma 8.1 on (116) we have

$$\mathcal{C}(R, TE, E, s_1, VE) \text{ w.r.t. } \varphi_2 \cup \varphi_2' \tag{169}$$

Also, from (118) and (119) we get

$$R \text{ connects } \varphi_2 \cup \varphi_2' \text{ to } s_1 \tag{170}$$

$$R' \text{ and } R \text{ agree on } \varphi_2 \cup \varphi_2' \tag{171}$$

By induction on (153), (169), (156), (170), (171) and (161) there exist s_2 and v_2' such that

$$s_1, VE, R' \vdash e_2' \to v_2', s_2 \tag{172}$$

$$\mathcal{C}(R', \mu', v_2, s_2, v_2') \text{ w.r.t. } \varphi_2' \tag{173}$$

Let $TE_0^+ = TE_0 + \{x_0 \mapsto \mu'\}$. Now (165) must have been inferred by Rules 23 and 28. Thus there exists a φ_0' such that $\varphi_0' \subseteq \varphi_0$ and

$$TE_0^+ \vdash e_0 \Rightarrow e_0' : \mu, \varphi_0' \tag{174}$$

We have $s_1 \sqsubseteq s_2$, by Lemma 4.1 on (172). By Lemma 8.2 on (166), (167) and $\varphi_0' \subseteq \varphi_0$ we then have

$$\mathcal{C}(R', TE_0, E_0, s_2, VE_0) \text{ w.r.t. } \varphi_0' \cup \varphi' \tag{175}$$

and by Lemma 8.1 on (173) and $\varphi_0' \subseteq \varphi_0$ we get

$$\mathcal{C}(R', \mu', v_2, s_2, v_2') \text{ w.r.t. } \varphi_0' \cup \varphi' \tag{176}$$

Let $E_0^+ = E_0 + \{x_0 \mapsto v_2\}$ and let $VE_0^+ = VE_0 + \{x_0 \mapsto v_2'\}$. Combining (175) and (176) we get

$$\mathcal{C}(R', TE_0^+, E_0^+, s_2, VE_0^+) \text{ w.r.t. } \varphi_0' \cup \varphi' \tag{177}$$

Also, by (118), (119) and $s \sqsubseteq s_2$ we get

$$R' \text{ connects } \varphi_0' \cup \varphi' \text{ to } s_2 \tag{178}$$

and by (167)

$$R_0 \text{ and } R' \text{ agree on } \varphi_0' \cup \varphi' \tag{179}$$

Then by induction on (174), (177), (157), (178), (179) and (168) there exist s' and v' such that

$$s_2, VE_0^+, R_0 \vdash e_0' \to v', s' \tag{180}$$

$$\mathcal{C}(R_0, \mu, v, s', v') \text{ w.r.t. } \varphi' \tag{181}$$

From (162), (164), (172) and (180) we get $s, VE, R' \vdash e_1'\ e_2' \to v', s'$, as desired. Moreover, by Lemma 8.2 on (181) and (167) we have $\mathcal{C}(R', \mu, v, s', v')$ w.r.t. φ', as desired.

Application of recursive closure, Rule 5 This case is similar to the previous case, but we include it for the sake of completeness. We have $e \equiv e_1 e_2$, for some e_1 and e_2, and $e' \equiv e_1' e_2'$, for some e_1' and e_2' and, by Rule 24, there exist μ', p, ϵ, φ_0, φ_1 and φ_2 such that

$$TE \vdash e_1 \Rightarrow e_1' : (\mu' \xrightarrow{\epsilon.\varphi_0} \mu, p), \varphi_1 \tag{182}$$

$$TE \vdash e_2 \Rightarrow e_2' : \mu', \varphi_2 \tag{183}$$

$$\varphi = \varphi_1 \cup \varphi_2 \cup \varphi_0 \cup \{\mathbf{get}(p), \epsilon\} \tag{184}$$

Also, assume that (117) was inferred by application of Rule 5 on premises

$$E \vdash e_1 \to v_1 \qquad v_1 = \langle x_0, e_0, E_0, f \rangle \tag{185}$$

$$E \vdash e_2 \to v_2 \tag{186}$$

$$E_0 + \{f \mapsto v_1\} + \{x_0 \mapsto v_2\} \vdash e_0 \to v \tag{187}$$

To use induction first time, we split the effect $\varphi \cup \varphi'$ into $\varphi_1 \cup \varphi_1'$, where $\varphi_1' = \varphi_2 \cup \varphi_0 \cup \{\mathbf{get}(p), \epsilon\} \cup \varphi'$. By (116), (118) and (119) we have

$$\mathcal{C}(R, TE, E, s, VE) \text{ w.r.t. } \varphi_1 \cup \varphi_1' \tag{188}$$

$$R \text{ connects } \varphi_1 \cup \varphi_1' \text{ to } s \tag{189}$$

$$R' \text{ and } R \text{ agree on } \varphi_1 \cup \varphi_1' \tag{190}$$

Also, by (120) we have

$$\mathrm{frv}(e_1') \subseteq \mathrm{Dom}\, R' \wedge \mathrm{frv}(e_2') \subseteq \mathrm{Dom}\, R' \tag{191}$$

By induction on (182), (188), (185), (189), (190) and (191), there exist v_1' and s_1 such that

$$s, VE, R' \vdash e_1' \to v_1', s_1 \tag{192}$$

$$\mathcal{C}(R', (\mu' \xrightarrow{\epsilon.\varphi_0} \mu, p), v_1, s_1, v_1') \text{ w.r.t. } \varphi_1' \tag{193}$$

Notice that $\mathbf{get}(p) \in \varphi_1'$. Thus by (193) and the rules for $\mathcal{C}$ (Rules 33, 35 and 36) we have $v_1' \in \mathrm{Pdom}(s_1)$ and $r\ of\ v_1' = R'(p)$ and there exist e_0', VE_0, TE_0 and R_0 such that

$$s_1(v_1') = \langle x_0, e_0', VE_0, R_0 \rangle \tag{194}$$

$$TE_0 \vdash \lambda x_0.e_0 \Rightarrow \lambda x_0.e_0' \texttt{ at } p : (\mu' \xrightarrow{\epsilon.\varphi_0} \mu, p), \{\mathbf{put}(p)\} \tag{195}$$

$$\mathcal{C}(R_0, TE_0, E_0 + \{f \mapsto v_1\}, s_1, VE_0) \text{ w.r.t. } \varphi_1' \tag{196}$$

$$R_0 \text{ and } R' \text{ agree on } \varphi_1' \tag{197}$$

$$\mathrm{frv}(e_0') \subseteq \mathrm{Dom}\, R_0 \tag{198}$$

To use induction a second time, we split the remaining effect φ_1' into $\varphi_2 \cup \varphi_2'$, where $\varphi_2' = \varphi_0 \cup \{\mathbf{get}(p), \epsilon\} \cup \varphi'$. We have $s \sqsubseteq s_1$, by Lemma 4.1. Then, by Lemma 8.1 on (116) we have

$$\mathcal{C}(R, TE, E, s_1, VE) \text{ w.r.t. } \varphi_2 \cup \varphi_2' \tag{199}$$

Moreover, (118) and (119) imply

$$R \text{ connects } \varphi_2 \cup \varphi_2' \text{ to } s_1 \tag{200}$$

$$R' \text{ and } R \text{ agree on } \varphi_2 \cup \varphi_2' \tag{201}$$

By induction on (183), (199), (186), (200), (201) and (191) there exist s_2 and v_2' such that

$$s_1, VE, R' \vdash e_2' \to v_2', s_2 \tag{202}$$

$$\mathcal{C}(R', \mu', v_2, s_2, v_2') \text{ w.r.t. } \varphi_2' \tag{203}$$

Let $TE_0^+ = TE_0 + \{x_0 \mapsto \mu'\}$. Now (195) must have been inferred by Rules 23 and 28. Thus there exists an effect φ_0' with $\varphi_0' \subseteq \varphi_0$ and

$$TE_0^+ \vdash e_0 \Rightarrow e_0' : \mu, \varphi_0' \tag{204}$$

By Lemma 8.2 on (196) and (197) we have

$$\mathcal{C}(R', TE_0, E_0 + \{f \mapsto v_1\}, s_2, VE_0) \text{ w.r.t. } \varphi_2' \tag{205}$$

Let $E_0^+ = E_0 + \{f \mapsto v_1\} + \{x_0 \mapsto v_2\}$ and let $VE_0^+ = VE_0 + \{x_0 \mapsto v_2'\}$ From (205) and (203) and $\varphi_0' \subseteq \varphi_0$ we have

$$\mathcal{C}(R', TE_0^+, E_0^+, s_2, VE_0^+) \text{ w.r.t. } \varphi_0' \cup \varphi' \tag{206}$$

From (197) we get

$$R_0 \text{ and } R' \text{ agree on } \varphi_0' \cup \varphi' \tag{207}$$

By (118), (119) and $s \sqsubseteq s_2$ we get

$$R' \text{ connects } \varphi_0' \cup \varphi' \text{ to } s_2 \tag{208}$$

By induction on (204), (206), (187), (208), (207) and (198) there exist s' and v' such that

$$s_2, VE_0^+, R_0 \vdash e_0' \to v', s' \tag{209}$$

$$\mathcal{C}(R_0, \mu, v, s', v') \text{ w.r.t. } \varphi' \tag{210}$$

Rule 12 on (192), (202), (194) and (209) gives $s, VE, R' \vdash e_1'\ e_2' \to v', s'$, as desired. Moreover, Lemma 8.2 on (210) and (207) gives the desired $\mathcal{C}(R', \mu, v, s', v')$ w.r.t. φ'.

`let` expressions, Rule 6 Assume (115) was inferred by Rule 25; then (115) takes the form

$$TE \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end} \Rightarrow \texttt{let}\ x\ \texttt{=}\ e_1'\ \texttt{in}\ e_2'\ \texttt{end} : \mu, \varphi \tag{211}$$

Moreover, (115) and (117) must be inferred by Rules 25 and 6 from the premises

$$TE \vdash e_1 \Rightarrow e_1' : (\tau_1, p_1), \varphi_1 \tag{212}$$

$$TE + \{x \mapsto (\tau_1, p_1)\} \vdash e_2 \Rightarrow e_2' : \mu, \varphi_2 \tag{213}$$

$$\varphi = \varphi_1 \cup \varphi_2 \tag{214}$$

$$E \vdash e_1 \to v_1 \tag{215}$$

$$E + \{x \mapsto v_1\} \vdash e_2 \to v \tag{216}$$

Let φ_1' be the effect that remains after the evaluation of e_1', i.e. let $\varphi_1' = \varphi_2 \cup \varphi'$. Note that $\varphi \cup \varphi' = \varphi_1 \cup \varphi_1'$, so by (116), (118) and (119) we have

$$\mathcal{C}(R, TE, E, s, VE) \text{ w.r.t. } \varphi_1 \cup \varphi_1' \tag{217}$$

$$R \text{ connects } \varphi_1 \cup \varphi_1' \text{ to } s \tag{218}$$

$$R' \text{ and } R \text{ agree on } \varphi_1 \cup \varphi_1' \tag{219}$$

By (120) we have

$$\text{frv}(e_1') \subseteq \text{Dom}\, R' \wedge \text{frv}(e_2') \subseteq \text{Dom}\, R' \tag{220}$$

By induction on (212), (217), (215), (218), (219) and (220) there exist s_1 and v_1' such that

$$s, VE, R' \vdash e_1' \to v_1', s_1 \tag{221}$$

$$\mathcal{C}(R', (\tau_1, p_1), v_1, s_1, v_1') \text{ w.r.t. } \varphi_1' \tag{222}$$

By Lemma 8.2 on (222) we get

$$\mathcal{C}(R, (\tau_1, p_1), v_1, s_1, v_1') \text{ w.r.t. } \varphi_1' \tag{223}$$

By Lemma 8.1 on (116) we get

$$\mathcal{C}(R, TE, E, s_1, VE) \text{ w.r.t. } \varphi_1' \tag{224}$$

Combining these two, we get

$$\mathcal{C}(R, TE + \{x \mapsto (\tau_1, p_1)\}, E + \{x \mapsto v_1\}, s_1, VE + \{x \mapsto v_1'\}) \text{ w.r.t. } \varphi_2 \cup \varphi' \tag{225}$$

By (118) and (119) and $s \sqsubseteq s_1$ we have

$$R \text{ connects } \varphi_2 \cup \varphi' \text{ to } s_1 \tag{226}$$

$$R' \text{ and } R \text{ agree on } \varphi_2 \cup \varphi' \tag{227}$$

By induction on (213), (225), (216), (226), (227) and (220) there exist s' and v' such that

$$s_1, VE + \{x \mapsto v_1'\}, R' \vdash e_2' \to v', s' \tag{228}$$

$$\mathcal{C}(R', \mu, v, s', v') \text{ w.r.t. } \varphi' \tag{229}$$

Here (229) is one of the desired results. Moreover, by Rule 13 on (221) and (228) we get the desired $s, VE, R' \vdash \texttt{let } x \texttt{ = } e_1' \texttt{ in } e_2' \texttt{ end} \to v, s'$.

`letrec`, Rule 7 In this case (115) takes the form

$$\begin{array}{l} TE \vdash \texttt{letrec}\ f(x)\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end} \Rightarrow \\ \quad \texttt{letrec}\ f\texttt{[}\rho_1\texttt{,}\ldots\texttt{,}\rho_k\texttt{]}(x)\ \texttt{at}\ p\ \texttt{=}\ e_1'\ \texttt{in}\ e_2'\ \texttt{end} : \mu, \varphi \end{array} \tag{230}$$

and is inferred by application of Rule 26 to the premises

$$TE + \{f \mapsto (\forall \rho_1 \cdots \rho_k \vec{\epsilon}.\underline{\tau}, p)\} \vdash \lambda x.e_1 \Rightarrow \lambda x.e_1'\ \texttt{at}\ p : (\tau, p), \varphi_1 \tag{231}$$

$$\mathrm{fv}(\vec{\alpha}, \vec{\rho}, \vec{\epsilon}) \cap \mathrm{fv}(TE, \varphi_1) = \emptyset \tag{232}$$

$$TE + \{f \mapsto (\sigma', p)\} \vdash e_2 \Rightarrow e_2' : \mu, \varphi_2 \tag{233}$$

$$\varphi = \varphi_1 \cup \varphi_2 \tag{234}$$

where $\vec{\rho} = \rho_1 \cdots \rho_k$ and $\sigma' = \forall \vec{\alpha}\, \vec{\rho}\, \vec{\epsilon}.\tau$. Moreover, (117) was inferred by Rule 7 on the premise

$$E + \{f \mapsto \langle x, e_1, E, f \rangle\} \vdash e_2 \to v \tag{235}$$

Since (231) must have been inferred by Rules 23 and 28, we have $\varphi_1 = \{\mathbf{put}(p)\}$. By (118) and (119) we have $R'(p) = R(p) \in \mathrm{Dom}(s)$. Let $r_1 = R(p)$. Let o_1 be an offset with $o_1 \notin \mathrm{Dom}(s(r_1))$. Let $v_1 = (r_1, o_1)$. Let $VE' = VE + \{f \mapsto v_1\}$ and let $s^+ = s + \{v_1 \mapsto \langle \rho_1, \ldots, \rho_k, x, e_1', VE', R' \rangle\}$. By Lemma 5.4 on (231) we have that

$$TE + \{f \mapsto (\sigma', p)\} \vdash \lambda x.e_1 \Rightarrow \lambda x.e_1'\ \texttt{at}\ p : (\tau, p), \varphi_1 \tag{236}$$

Let $TE^+ = TE + \{f \mapsto (\sigma', p)\}$ and let $E^+ = E + \{f \mapsto \langle x, e_1, E, f \rangle\}$. By (120) we have

$$\mathrm{frv}(e_1') \subseteq \mathrm{Dom}\, R' \cup \{\rho_1, \ldots, \rho_k\} \wedge \mathrm{frv}(e_2') \subseteq \mathrm{Dom}\, R' \tag{237}$$

By Lemma 8.6 on (116), (232), (236), (119) and (237) we have $\mathcal{C}(R, TE^+, E^+, s^+, VE')$ w.r.t. $\varphi \cup \varphi'$. Then by Lemma 8.1 we get

$$\mathcal{C}(R, TE^+, E^+, s^+, VE') \text{ w.r.t. } \varphi_2 \cup \varphi' \tag{238}$$

Also, by (118) and (119) we get

$$R \text{ connects } \varphi_2 \cup \varphi' \text{ to } s^+ \tag{239}$$

$$R' \text{ and } R \text{ agree on } \varphi_2 \cup \varphi' \tag{240}$$

By induction on (233), (238), (235), (239), (240) and (237) there exist s' and v' such that

$$s^+, VE', R' \vdash e_2' \to v', s' \tag{241}$$

$$\mathcal{C}(R', \mu, v, s', v') \text{ w.r.t. } \varphi' \tag{242}$$

From (241) and Rule 14 we get

$$s, VE, R' \vdash \texttt{letrec}\ f\texttt{[}\rho_1\texttt{,}\ldots\texttt{,}\rho_k\texttt{]}(x)\ \texttt{at}\ p\ \texttt{=}\ e_1'\ \texttt{in}\ e_2'\ \texttt{end} \to v', s' \tag{243}$$

Now (242) and (243) are the desired results.

This concludes the proof of Theorem 6.1.

10 Algorithms

The algorithms used for implementing the region inference rules in the ML Kit will not be described here. We shall give a brief overview, however. First, ordinary ML type inference is performed using Milner's algorithm W, extended to all of Core ML. The output of this phase is an explicitly typed lambda term, e_0, say. Then region inference is done in two phases. First e_0 is decorated with fresh region and effect variables everywhere a region and effect variable will be required in an explicitly typed version the fully region annotated target expression. This phase is called *spreading*. During spreading, every recursive function f of type scheme σ^{ML}, say, is given the most general type scheme σ_0 which has σ^{ML} as its projection (in the sense of Section 5.3). For example, a `letrec`-bound $\texttt{int} \to \texttt{int}$ function will be given type scheme $\forall\rho_1\rho_2\epsilon.(\texttt{int},\rho_1) \xrightarrow{\epsilon.\emptyset} (\texttt{int},\rho_2)$. The spreading phase performs the unifications suggested by the inference rules. For example, the two occurrences of μ' in Rule 24 suggest a unification of the types and places of operator and operand. Spreading employs rules 27 and 28 as aggressively as possible (i.e., after every application of rules 22, 24, 25 and 26). The resulting program, call it e_1, is well-annotated with regions, except for the fact that the type schemes assumed for recursive functions may be too general, compared to the type schemes that were inferred for the lambda expressions which define the functions.

The second phase is called *fixed-point resolution* and takes e_1 as input. For each recursive function in e_1, the region inference steps (unification, introduction of letregions etc) are iterated, using less and less general type schemes for the recursive functions, till a fixed point is reached. This is similar in spirit to Mycroft's algorithms for full polymorphic recursion (Mycroft 1984).

It is possible to extend the notion of principal unifiers for types to a notion of principal unifier for region-annotated types, even though region-annotated types contain effects. This relies on invariants about arrow effects that were outlined in Section 5.1. One can prove that every two types τ_1 and τ_2 that have the same underlying ML type have a most general unifier, provided all the arrow effects in τ_1 and τ_2 satisfy the invariants.

The reason for the separation of spreading and fixed-point resolution is that, unless one takes care, the iteration used to handle the polymorphic region recursion does not terminate. In particular, there is a danger of arrow effects that grow ever larger, as more fresh region and effect variables are generated. The division into spreading and fixed-point resolution solves this problem by only generating fresh variables during the spreading phase. It can then be shown that the second phase always terminates. This approach does not always give principal types, for there are cases where that function in the fixed-point resolution which is responsible for forming type schemes is refused the opportunity to quantify region and effect variables even though it is permitted by the inference

rules. When this happens, the implementation simply prints a warning about the possible loss of principal types and continues with a less-than-principal type scheme. Fortunately, this happens rather infrequently in practice, and since the soundness result of the present paper shows the correctness for *all* derivations $TE \vdash e \Rightarrow e' : \mu, \varphi$, safety is not violated.

11 Language Extensions

In this section we outline some of the extensions that have been made to the region inference rules in order to handle references, exceptions and recursive datatypes in the ML Kit.

11.1 References

Assume primitives `ref`, `!` and `:=` for creating a reference, dereferencing and assignment, respectively. For the purpose of region inference, these can be treated as variables with the following type schemes:

$$\begin{array}{rcl}
\texttt{ref} & : & \forall\alpha\rho_1\rho_2\epsilon.(\alpha,\rho_1) \xrightarrow{\epsilon.\{\mathbf{put}(\rho_2)\}} ((\alpha,\rho_1)\texttt{ref},\rho_2) \\
\texttt{!} & : & \forall\alpha\rho_1\rho_2\epsilon.((\alpha,\rho_1)\texttt{ref},\rho_2) \xrightarrow{\epsilon.\{\mathbf{get}(\rho_2)\}} (\alpha,\rho_1) \\
\texttt{:=} & : & \forall\alpha\rho_1\rho_2\rho_3\rho_4\epsilon.(((\alpha,\rho_1)\texttt{ref},\rho_2) * (\alpha,\rho_1),\rho_3) \xrightarrow{\epsilon.\{\mathbf{put}(\rho_2),\mathbf{put}(\rho_4)\}} (\texttt{unit},\rho_4)
\end{array}$$

The most interesting of these is assignment. The new contents of the reference is represented by a pointer (or by a word, if the value is in unboxed representation). The assignment updates the reference with this pointer (or word). Thus there is a **put** effect on the region where the reference resides. The assignment does not make a copy the stored value. Thus assignment is a constant time operation, but the downside is that the old and the new contents must be in the same regions (see the two occurrences of ρ_1 in the type for `:=`). Thus, for values with boxed representation, all the different contents of the reference will be kept alive for as long as the reference is live. In "mostly functional" programs this does not seem to be a serious problem and even if there are many side-effects, one can still expect reasonable memory usage as long as the references are relatively short-lived. Long-lived references that contain boxed values and are assigned freshly created contents often are hostile to region inference.

11.2 Exceptions

Our approach here is simple-minded: exception values are put into global regions. Every evaluation of an exception declaration gives rise to an allocation in some global region. Application of a unary exception constructor to an argument forces

the argument to bein global regions as well. Thus if one constructs may exception values using unary exception constructors, one gets a space leak (indeed, the space leaking region ρ_{122} in Figure 5 contains constructed exception values). If one uses nullary constructors only, there is only going to be one allocation for each evaluation of each exception declaration.

11.3 Recursive Datatypes

So far, every type constructor has been paired with one region variable. For values of recursive datatypes, additional region variables, the so-called *auxiliary* region variables, are associated with type constructors. For example, consider the declaration of the `list` datatype:

```
datatype 'a list = nil | :: of 'a * 'a list
```

The region-annotated version of the type α `list` takes the form $(\alpha, \rho_1)(\texttt{list}_{[\rho_2]}, \rho_3)$, where ρ_1 stands for a region which contains the list elements, ρ_3 contains the spine of the list (i.e., the constructors `nil` and `::`) and ρ_2 is an auxiliary region which contains the pairs, to which `::` is applied. Thus lists are kept "very boxed": in region ρ_3 every cons cell takes up two words, the first is a tag (saying "I am cons") and the second is a pointer to the pair to which `::` is applied. The region ρ_2 is called auxiliary because it holds values which are internal to the datatype declaration; there will be one auxiliary region for each type constructor or product type formation in each constructor in the datatype. However, all occurrences of the type constructor being declared are put in the same region. Hence `::` receives type

$$\forall\rho_1\rho_2\rho_3\alpha.((\alpha,\rho_1) * ((\alpha,\rho_1)\texttt{list}_{[\rho_2]}, \rho_3), \rho_2) \xrightarrow{\varepsilon.\{\mathbf{put}(\rho_3)\}} ((\alpha,\rho_1)\texttt{list}_{[\rho_2]}, \rho_3)$$

Sequential datatype declarations pose an interesting design problem:

```
datatype t1 = C of int
datatype t2 = C of t1 * t1
datatype t3 = C of t2 * t2
...
datatype ti = C of ti-1 * ti-1
...
```

In the declaration of t_i, should one give the two occurrences of t_{i-1} on the right-hand side the same or different regions? If one gives them the same regions, one introduces unnecessary sharing; if one gives them different regions, the number of auxiliary region variables grows exponentially in i, potentially leading to slow region inference. A third possibility is to put a limit on the number of auxiliary region variables one will allow. We have chosen the second solution (spreading regions as much as possible), but a systematic empirical study of different solutions has not been conducted.

12 Strengths and Weaknesses

The region inference rules were first implemented in a prototype system (Tofte and Talpin 1994) and then in the ML Kit (Birkedal *et al.* 1996). Neither of these systems use garbage collection. This section records some of the experience gained from these systems, with special emphasis on how details of the region inference rules influence memory management. We first illustrate consequences of the region inference rules by a series of small, but complete, examples. Then we report a few results from larger benchmarks run on the ML Kit. Throughout, we use Standard ML syntax (Milner *et at.* 1990); roughly, `fun` is translated into `letrec` and `val` into `let`.

12.1 Small examples

The examples are grouped according to the general point they are intended to make.

12.1.1 Polymorphic Recursion

Generally speaking, polymorphic region recursion favours recursive functions that have a balanced call tree (as opposed to an iterative computation, where the call tree is a list). We illustrate this with two examples. The first is the exponential version of the Fibonacci function:

```
fun fib n = if n<=1 then 1 else fib(n-2) + fib(n-1)
val fib15 = fib 15;
```

Due to region polymorphism, the two recursive calls of `fib` use different regions, local to the body (see Figure 2). The memory usage appears in Figure 4.

The next example, called `reynolds2` (Birkedal *et al.* 1996) is a depth-first search in a tree, using a predicate to record the path from the root to the present node.

```
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
```

	max inf reg *(i)*	max fin reg *(ii)*	max stack *(iii)*	final *(iv)*
`fib15`	1,600	0	188	0
`appel1`	484,000	2,024	12,240	0
`appel2`	9,600	820	3,508	0
`reynolds2`	8,800	500	1,856	44
`reynolds3`	75,508,000	720	2,188	44
`string1`	35,200	4,052	11,188	6,500
`string2`	156	2,028	7,280	156

Figure 4: Memory usage when running sample programs on the ML Kit with Regions, Version 29a3: *(i)* Maximal space (in bytes) used for variable size regions (one region page is 800 bytes); *(ii)* Maximal space (in bytes) used for fixed size regions; *(iii)* Maximal stack size during execution (in bytes); *(iv)* Number of bytes holding values at the end of the computation (regions on stack + data in variable sized regions)

```
            orelse
            search (fn y => y=x orelse p y) t2
val reynolds2 = search (fn _ => false) (mk_tree 20)
```

Due to the polymorphic recursion, the recursive call of `search` does not put the closures for `(fn y => y=x orelse p y)` in the same region as `p`, so the space usage will be proportional to the depth of the tree. This leads to good memory utilisation (Figure 4).

By contrast, consider the first-order variant, called `reynolds3`, which uses a list to represent the path. It 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
val reynolds3 = search [] (mk_tree 20)
```

As we saw in Section 11, region inference does not distinguish between a list and its tail, so all cons cells (one for each node in the tree) are put in the same region. This gives poor memory utilisation, the difference from `reynolds2` being

exponential in the depth of the tree (Figure 4). More generally, in connection with a recursive datatype, one should not count on polymorphic recursion to separate the life-times of a value v of that type and other values of the same type contained in v.

12.1.2 Tail recursion

Another common pattern of computation is iteration. This is best implemented using a recursive function whose type scheme takes the form $\forall\vec{\alpha}\vec{\rho}\vec{\epsilon}.(\mu \xrightarrow{\epsilon.\varphi} \mu)$ (note that the argument and result types are the same, even after region annotation). Such a function is called a *region endomorphism*. Here is how to write a simple loop to sum the numbers 1 to 100:

```
fun sum(p as (acc,0)) = p
  | sum(acc,n) = sum(n+acc,n-1)
val sumit = #1(sum(0,100));
```

In ML, all functions in principle takes one argument, in his case a tuple, and that is how it is implemented in the ML Kit. One might think that 100 pairs would pile up in one region; however, an analysis called the *storage mode analysis* (Birkedal *et al.* 1996) discovers that the region can be reset just before each pair is written, so that in fact the region will only ever contain one pair. Memory usage is independent of the number of iterations, in this example. By contrast, the non-tail-recursive version

```
fun sum' 0 = 0
  | sum' n = n + sum'(n-1)
val sum'it = sum' 100
```

uses stack space proportional to the number of iterations.

The next program, `appel1`, is a variant of a program in (Appel 1992).:

```
fun s(0) = nil
  | s(i) = 0 :: s(i-1)
fun length [] = 0
  | length(x::xs) = 1 + length xs
val N = 100
fun f(n,x)=
    let val z = length x
    in if n=0 then 0 else f(n-1, s N)
    end
val appel1 = f(N,nil)
```

Here $\texttt{f}(n, \texttt{nil})$ uses space $\Theta(N^2)$, although $\Theta(N)$ should be enough. The problem is that at each iteration a list of length N is created, put in a fresh region, and

then passed to the recursive call, which only uses the list to compute `z`. The list, however, stays live till the end of the recursive call: Rule 23 and 27 tell us that the λ-bound `x` will be allocated throughout the evaluation of the body of `f`. The cure in this case is not to use the polymorphic recursion:

```
fun f(p as (n,x))=
    let val z = length x
    in if n=0 then 0 else f(if true then (n-1, s N) else p)
    end
val appel2 = f(N,nil)
```

Now the storage mode analysis will discover that the region containing the entire list can be reset at each iteration; this is tail call optimisation for recursive datatypes! The above transformation is a rather indirect way of instructing the region inference algorithm that one does not want polymorphic recursion and if the optimiser eliminated the conditional, it would not even have the desired effect. It would probably be better to allow programmers to state their intentions directly. Memory consumption is in Figure 7.

12.1.3 Higher-order functions

If a function f is lambda-bound, it is not region-polymorphic (Rule 23). For example, consider

```
fun foldl f acc [] = acc
  | foldl f acc (x::xs) = foldl f (f(acc,x)) xs
fun concat l = foldl (op ^) "" l
fun blanks 0  = []
  | blanks n = " " :: blanks(n-1)
val N = 100
val string1 = concat(blanks N)
```

Despite the fact that `foldl` is region-polymorphic, the lambda-bound `f` is not, so all applications of the concatenation operator `^` in `concat` will put their results in the same region, leading to $\Theta(N^2)$ space usage. To obtain $\Theta(N)$ space usage, one specializes `foldl` to `^`, uncurries the resulting function and turns it into a region endomorphism:

```
fun concat'(p as (acc,[])) = p
  | concat'(acc,(x::xs)) = concat'(acc ^ x, xs)
fun concat(l) = #1(concat'("", l))
fun blanks 0  = []
  | blanks n = " " :: blanks(n-1)
val string2 = concat(blanks 100)
```

12.2 Larger Benchmarks

A number of benchmarks from the New Jersey Standard ML benchmark suite have been ported to the Kit and compared (space and time usage) against execution as stand-alone programs under Standard ML of New Jersey, version 93. The largest benchmark is Simple (1148 lines), a program which originally used arrays of floating point numbers extensively. To make it run on the Kit (which does not support arrays) arrays were translated into lists of references, so the ported program is probably not indicative of how one would write the program without arrays to start with. Life (252 lines) uses lists very extensively; Mandelbrot (170 lines) uses floating points extensively; Knuth-Bendix (752 lines) does extensive dynamic allocation of data structures that represent terms.

Initially, programs often use more space when running on the Kit; for example, Figure 5 shows a region profile for the original version of the Knuth-Bendix benchmark, produced using Hallenberg's region profiler (Hallenberg 1996). The region profiler can also pinpoint the program points which are responsible for space leaks. The source program is then changed, to make it more region friendly. Interestingly, transformations that are good for region inference often are good for SML/NJ too (see Knuth-Bendix in Figure 7 for an example). This is not very surprising: when the static analysis is able to infer shorter lifetimes, it may well be because the values actually need to be live for a shorter time, and this is good for garbage collection too. The region profile of the improved Knuth-Bendix completion is shown in Figure 6; see Figure 7 for a comparison with SML of New Jersey, version 93.

12.3 Automatic Program Transformation

Apart from functions that are deliberately written as region endomorphisms, the general rule is that the more regions are separated, the better (since it makes more aggressive re-cycling of memory possible). The Kit performs optimisations which separate regions. These include replacing `let` x `=` e_1 `in` e_2 `end` by $e_2[e_1/x]$ in cases where e_1 is a syntactic value and either x occurs at most once in e_2 or the value denoted by e_1 is not larger than some given constant. Another optimisation, which is implemented, is specialisation of curried functions, as in the `string2` example above; however, the Kit does not attempt to turn functions into region endomorphisms (which was the last thing we did in `string2`). As a matter of principle, the Kit avoids optimisations which can lead to increased memory usage.

Also useful is the ability of the region inference to suggest where space leaks may be expected. If a function has compound type scheme

$$\forall \vec{\rho}\vec{\alpha}\vec{\epsilon}.\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2$$

and φ contains an atomic effect of the form $\mathbf{put}(\rho)$, where ρ is not amongst the bound region variables $\vec{\rho}$, then one quite possibly has a space leak: every call of

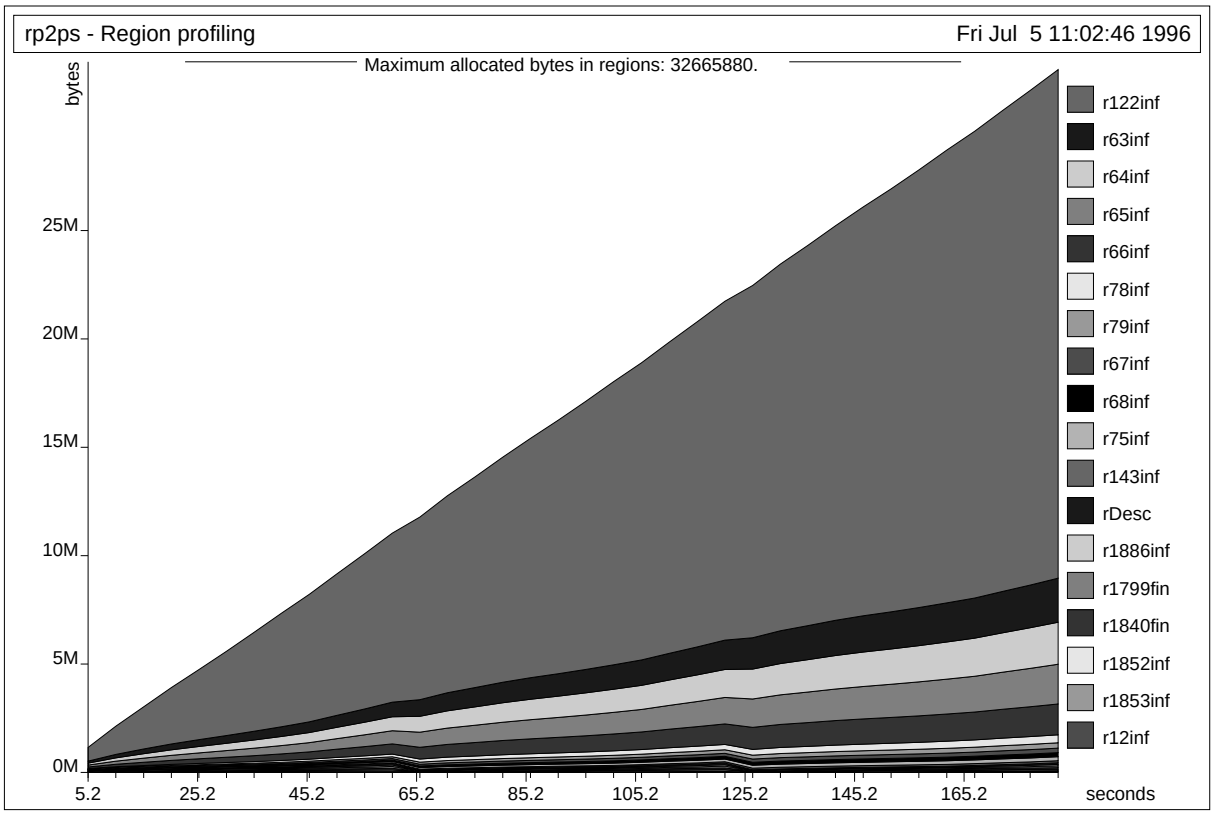


Figure 5: Region profile for Knuth-Bendix before optimisations. One region (ρ_{122}) of unbounded size, indicated as `r122inf` in the picture, is responsible for most of the space leak. Additional profiling reveals that a single program point (the application of an exception constructor to a constant string) is responsible for all values in that region.

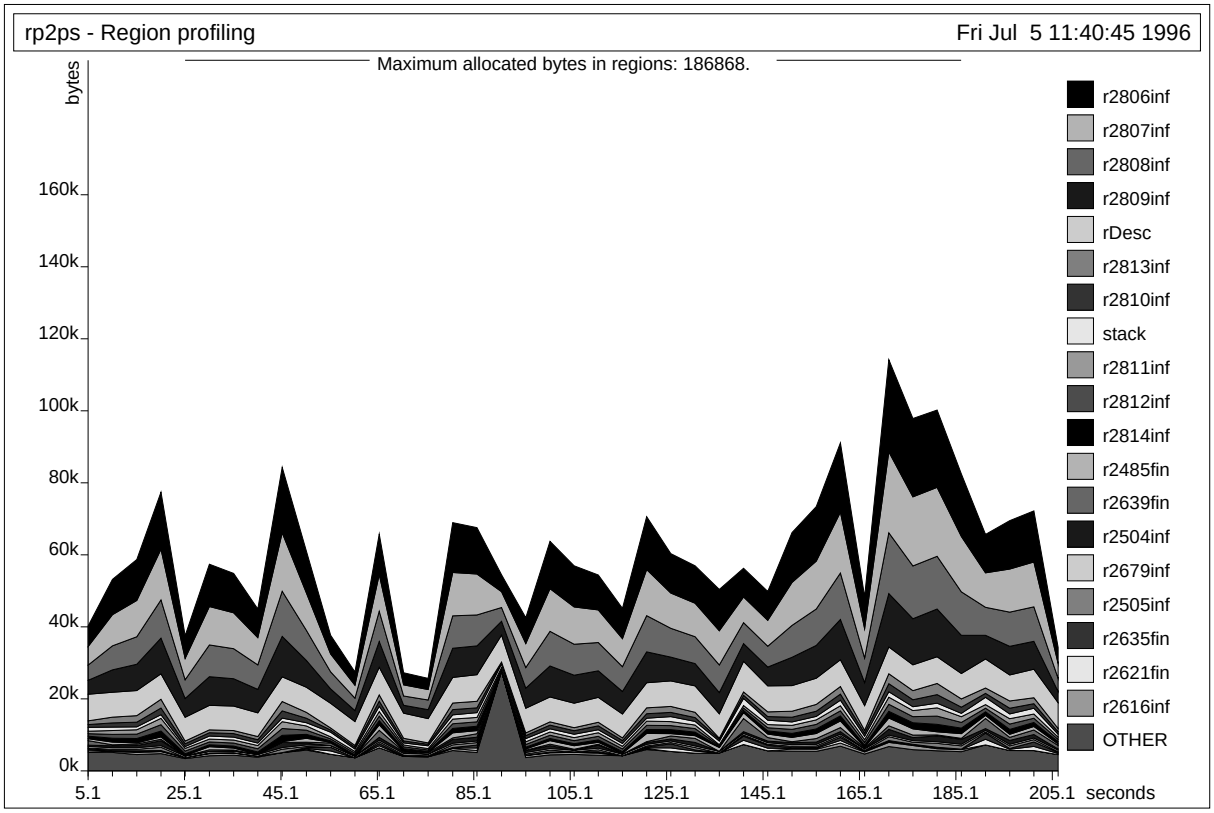


Figure 6: Region profile for Knuth-Bendix after optimisations

	Time Kit	Time SML/NJ	Space Kit	Space SML/NJ
Mandelbrot (orig.)	32.6	22.10	360	937
Life (orig.)	7.20	8.76	4,304	1,793
Life (impr.)	7.55	6.72	228	1,775
Knuth-Bendix (orig.)	30.5	20.92	33,853	6,761
Knuth-Bendix (impr.)	29.41	22.34	680	2,253
Simple (orig.)	57.58	14,33	1,692	2,136

Figure 7: Comparison between stand-alone programs created with the ML Kit (using the HP PA-RISC code generator) and SML of New Jersey, respectively. Here "orig" means original program, while "impr" means improved for region inference. All times are user time in seconds on an HP 9000 s700, measured using the unix `time` command. Space is maximal resident memory in kilobytes, measured with `top`, and includes code and runtime system. All values are average over three runs.

the function might put a value into some region which is external to the function. If in addition ρ does not occur free in μ_2, that is all the more reason for concern, for the value will not even be part of the result of the function. In other words, the function has a side-effect at the implementation level. This can easily happen even when there are no side-effects in the source program.

In such cases, the implementation simply issues a short warning. This turns out to be very useful in practice.

Another usage of the inferred information is the ability to detect dead code. Consider the rule for `letregion` (Rule 27). If $\mathbf{put}(\rho) \in \varphi$ and $\mathbf{get}(\rho) \notin \varphi$ then whatever value that was put into ρ was never used. For example, this can detect that the functions `f` and `g` below are never used:

```
let
  fun f(x) = x+1
  fun g(x) = f(f(x))
in
   (fn x => 3)(fn() => g 5)
end
```

12.4 Conclusion

As has been shown with the previous examples, it is not the case that every ML program automatically runs well on a stack of regions. Often, one has to program in a region friendly style, aided by profiling tools to find space leaks. Thus, programming with regions is different from usual ML programming, where one

relies on a garbage collector for memory management. On the other hand, the region discipline offers what we feel is an attractive combination of the convenience of an expressive programming language and the ability to reason about the time and space performance of programs. The relationship between the abstract model of the regions presented in this paper and the concrete implementation is close enough that one can use the abstract model — combined with the profiling tools mentioned earlier — to tune programs, often resulting in very space efficient programs that are executed as written, with no added costs of unbounded size.

Acknowledgments

It would have been impossible to assess the practical use of the region inference rules without the software developed by the ML Kit with Regions development team. Lars Birkedal wrote the compiler from region-annotated lambda-terms to C, together with a runtime system in C. Martin Elsman and Niels Hallenberg extended this work to HP PA-RISC code generation, including register allocation and instruction scheduling. Magnus Vejlstrup developed the multiplicity inference for inferring region sizes. Niels Hallenberg implemented the region profiler. Peter Sestoft and Peter Bertelsen conducted thorough tests of the system and improved the storage mode analysis. The first author wishes to thank Mikkel Thorup and Bob Paige for generously providing algorithmic expertice, specifically on graph algorithms; their input was very important for the detailed design and implementation of the region inference algorithms in the Kit. The depth-first search algorithms in Section 12.1 were suggested by John Reynolds. Finally, we wish to thank the referees for many constructive suggestions and comments.

References

[1] A. Aiken, M. Fähndrich, and R. 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] A. W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[3] H. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[4] H. G. Baker. Unify and conquer (garbage collection, updating, aliasing, ...) in functional languages. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 218–226, June 1990.

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

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

[7] 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.

[8] E. W. Dijkstra. Recursive programming. *Numerische Math*, 2:312–318, 1960. Also in Rosen: "Programming Systems and Languages", McGraw-Hill, 1967.

[9] M. Elsman and N. 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.

[10] M. Georgeff. Transformations and reduction strategies for typed lambda expressions. *ACM Transactions on Programming Languages and Systems*, 6(4):603–631, Oct 1984.

[11] P. Hudak. A semantic model of reference counting and its abstraction. In *ACM Symposium on List and Functional Programming*, pages 351–363, 1986.

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

[13] H. S. Katsuro Inoue and H. Yagi. Analysis of functional programs to detect run-time garbage cells. *ACM Transactions on Programming Languages and Systems*, 10(4):555–578, 1988.

[14] D. E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, 1972.

[15] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

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

[17] 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.

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

[19] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[20] A. Mycroft. Polymorphic type schemes and recursive definitions. In *Proc. 6th Int. Conf. on Programming, LNCS 167*, 1984.

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

[22] C. Ruggieri and T. P. Murtagh. Lifetime analysis of dynamically allocated objects. In *Proceedings of the 15th Annual ACM Symposium on Principles of Programming Languages*, pages 285–293, January 1988.

[23] J.-P. Talpin. Theoretical and practical aspects of type and effect inference. Doctoral Dissertation, May 1993. Also available as Research Report EMP/CRI/A-236, Ecole des Mines de Paris.

[24] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3), 1992.

[25] M. Tofte and J.-P. Talpin. A theory of stack allocation in polymorphically typed languages. Technical Report DIKU-report 93/15, Department of Computer Science, University of Copenhagen, 1993.

[26] M. Tofte and J.-P. 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.

Index

The index refers to sections where the concepts are introduced. For example, the entry "region name ($r \in$ RegName) 2, Figure 1, 4.1" means that the notion of region name is introduced in Sections 2 and 4.1, appears in Figure 1 and that meta-variable r ranges over region names throughout the paper.

A Appendix: Example Three-Address Code

The three-address code which the ML Kit produces on the way to HP PA-RISC code for the example given in Section 1 is shown below. Temporary variables start with `V`. Fixed registers are used for the stack pointer (`SP`), and for function call and return (`stdArg`, `stdClos`, `stdRes`). In this example, the compiler discovers that all regions can be represented on the stack; in other cases, `letregion` and `end` translate into calls of runtime system procedures that resemble lightweight `malloc` and `free` operations.

```
LABEL 1: (*  main *)
   ...
   AllocRegion(V43); (* allocate global region  rho1 *)
   ...
                            (* begin LETREGION [rho4 ,rho5 ] *)
   Move(SP,V46);            (* V46:= SP, i.e. rho4 *)
   Offset(SP,12,SP);
   Move(SP,V47);            (* rho5 *)
   Offset(SP,12,SP);
                            (* begin APP --- non tail call *)
                            (* begin operator *) ;
                            (* begin LETREGION (rho6 eliminated) *)
                            (* begin LET *)
                            (* begin RECORD *)
   Move(V47,V54);           (* allocate storage for record *)
   Move(5,V55);             (* 5 represents 2 *)
   StoreIndexL(V55,V54,1);  (* store component of record *)
   Move(7,V55);             (* 7 represents 3 *)
   StoreIndexL(V55,V54,2);  (* store component of record *)
   StoreIndexL(20,V54,0);   (* tag *)
   Move(V54,V51);           (* save address of record as result *)
                            (* end of RECORD *)
                            (* LET scope: *)
   Move(V46,V52); (* allocate storage for closure for FN y => ...*)
   StoreIndexL(Lab5,V52,0); (* store code pointer in closure *)
   Move(V51,V53);
   StoreIndexL(V53,V52,1);  (* save free variable x in closure *)
   FetchVars(V43);
   Move(V43,V53);
   StoreIndexL(V53,V52,2);  (* save free variable rho1 in closure*)
   Move(V52,V48);           (* save address of closure as result *)
                            (* end LET *)
                            (* end LETREGION (rho6 eliminated)*)
                            (* end operator, begin operand *)
   Move(11,V49);            (* 11 represents 5 *)
                            (* end operand *)
```

```
   Push(Lab4);              (* push return address *)
   Move(V48,stdClos);
   Move(V49,stdArg);
   FetchIndexL(stdClos,0,V50); (* fetch code address from closure *)
   Jmp(V50)
LABEL 4:                    (* return address *)
   Move(stdRes,V45);
                            (* end APP *) ;
   Offset(SP,~12,SP);       (* end LETREGION rho5*)
   Offset(SP,~12,SP);       (* end LETREGION rho4*)
   HALT

LABEL 5:                    (* code for function FN y => ... *)
                            (* begin RECORD *)
   FetchVars(V43);
   Move(V43,V57);
   AllocMemL(V57,3,V57);    (* allocate storage for record at rho1 *)
   FetchIndexL(stdClos,1,V59); (* access variable: x *)
   FetchIndexL(V59,1,V58);  (* extract component 0 from record. *)
   StoreIndexL(V58,V57,1);  (* store component of record *)
   Move(stdArg,V58);        (* access variable: y *)
   StoreIndexL(V58,V57,2);  (* store component of record *)
   StoreIndexL(20,V57,0);   (* tag *)
   Move(V57,stdRes);        (* save address of record as result *)
                            (* end of RECORD *)
                            (* return: *)
   Pop(V56);
   Jmp(V56)
```