A Theory of Stack Allocation in Polymorphically Typed Languages*

Mads Tofte, Dept. of Computer Science, Copenhagen University
Jean-Pierre Talpin, European Computer-Industry Research Center†

July 9, 1993

Abstract

We present a stack-allocation scheme for the call-by-value lambda calculus typed according to Milner's polymorphic type discipline. All the values a program produces, including function closures, are put into regions at runtime. Regions are allocated and deallocated in a stack-like manner. Region inference and effect inference is used to infer where regions can be allocated and deallocated. By allowing a limited form of polymorphic recursion in region types, the region inference is able to distinguish between the life-times of different invocations of a function, even when the function is recursive.

The region analysis is eager in its attempt to reclaim memory as quickly as possible. The main result of this report is that region inference is safe, a result which entails that regions really can be deallocated, when region inference says they can. We give detailed proofs of this claim, which (in various forms) has been advanced several times in the literature.

An algorithm for implementing the region inference rules is presented.

Finally, we present simulation results obtained with a prototype implementation. This implementation had only regions (i.e. no heap and no garbage collection). Programs with interesting forms of storage behaviour were tested; most of them turned out to use modest amounts of memory. In particular, 5000 numbers were sorted using a list-processing version of Quicksort in less than five times the memory needed to represent the list.

*Technical Report 93/15, Department of Computer Science, Copenhagen University.
†Work done while at Ecole des Mines de Paris.

Contents

1 Introduction

The stack allocation scheme for block-structured languages[10,9] often gives economical use of memory resources. Part of the reason for this is that the stack discipline is eager to reuse dead memory locations (i.e. locations, whose contents is of no consequence to the rest of the computation). Every point of allocation is matched by a point of deallocation and these points can easily be identified in the source program.

In heap-based storage management schemes[4,18,17], allocation is separate from deallocation, the latter being handled by garbage collection. This separation is useful when the lifetime of values is not apparent from the source program. However, compared to stack allocation schemes, heap-based schemes are often lazy in reusing memory, with the result that memory is being used for holding dead values.

All automatic allocation schemes, regardless of whether they involve a stack or a heap, rest on conservative assumptions about when data can be considered dead. In the stack allocation scheme, for example, an actual parameter associated with an activation record for some procedure, P, will remain allocated for as long as the activation record for P exists, even in cases where P only uses the actual parameter for a short time, compared to the duration of the call of P. Heap allocation also rests on conservative assumptions about liveness, for instance the assumption that every memory cell which can be reached via pointers from some set of root pointers may potentially be used by the rest of the computation.

One can construct examples where stack allocation gives better storage utilisation than heap allocation, and vice versa. Therefore, one scheme is not always better than the other. This report is about when stack allocation is *possible in principle*; it is not about when it is *desirable in practice*. In particular, we are interested in the connection between stack allocation and certain language features of present-day programming languages, notably recursive datatypes and higher-order functions. The language we have studied is essentially Milner's language Exp, i.e. the call-by-value λ-calculus with recursive functions and polymorphic `let`[21]. We present a stack discipline which can be used for evaluating all well-typed Exp programs. We prove that our stack allocation scheme is *safe*, a result which entails that references to memory cells that have been deallocated never are referenced by the rest of the computation.

Our use of the term "stack allocation" is somewhat non-standard. Our model of the runtime system involves a stack of *regions*, see Figure 1. We do not expect always to be able to determine the size of a region when we allocate it. Part of the reason for this is that we consider recursive datatypes, such as lists, a must; the size of a region which is supposed to hold an entire list, say, cannot in general be determined when the region is allocated. Our stack of regions can therefore normally not be represented by a hardware runtime stack.

Our stack allocation scheme differs from the classical stack allocation scheme in that it admits functions as first-class values. In particular, functions can return functions as results. The points of allocation and deallocation are still coupled and can be nested, but they are not obvious from the source program. Instead, they are inferred mechanically

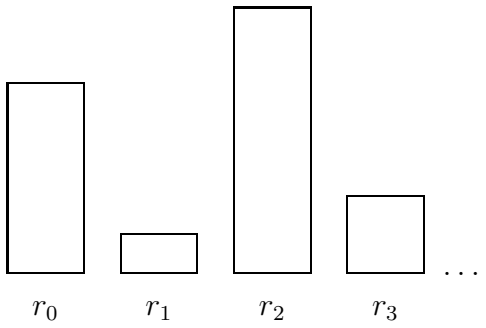


Figure 1: The store is regarded as having two dimensions; a region is a box in the picture. Regions are allocated and deallocated in a stack-like manner, with the stack growing to the right. In the mathematical model of the runtime system, we allow an infinite number of regions; the question of horizontal and vertical overflow in finite memory is discussed in Section 6.

using a translation scheme called *region inference*. Region introduction and elimination follow inference rules that are presented below.

Our stack of regions resembles the scheme of Ruggieri and Murtagh[27]. They suggest having a stack of subheaps. Each subheap 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 subheaps to put values in. They consider updates, which we do not. However, we deal with polymorphism and higher-order functions, which they do not.

Region inference is based on polymorphic type inference. The net effect of region inference is to translate every source language expression into a target language expression. Roughly speaking, target expressions are annotated source expressions; 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 a constant expression, a tuple-building expression, a λ-abstraction or some other form of expression which directly constructs a value. Here 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. `letregion` binds ρ in e_2 and the usual definitions about free and bound variables apply. At runtime, `letregion` ρ `in` e_2 `end` is evaluated as follows. First an unused region, r, is allocated and bound to ρ. Then e_2 is evaluated, a process which can involve using r for storing values. Finally, r is deallocated. Clearly, the `letregion` construct is similar to the block construct in block-structured languages, only blocks are inserted automatically in

our scheme. The `letregion` expression is the only way of introducing and eliminating regions. It is clear, therefore, that regions are allocated and deallocated in a stack-like manner.

By basing our translation scheme on type inference, it becomes possible to separate the definition of what it means for a target program to be well annotated, from the problem of finding an algorithm, which performs the annotation. Our translation rules resemble Milner's inference rules[21,7]; our region inference algorithm takes as input an expression which is assumed to have been type-checked (and type-annotated) using Milner's algorithm W. To determine the lifetime of objects, we use effect inference[19,20,14].

Our region inference rules are similar to those of Talpin and Jouvelot[31], who do prove a soundness result, albeit for a dynamic semantics, in which there are no regions and hence no notion allocation and deallocation of regions at runtime.

The idea that unification of region variables during type inference performs a kind of value-flow analysis is nicely presented by Baker[3]. Baker's location variables correspond to our region variables; Baker does not prove safety, however.

Good separation of lifetimes hinges on the use of (a limited form) of polymorphic recursion. This approach to the separation of lifetimes is novel, as far as we know.

The rest of this report is organised as follows. The source and the target languages are presented in Section 2. The region inference rules are presented in Section 3. The safety of our translation scheme is proved in Section 4. The region inference algorithm is presented in Section 5. In Section 6 we discuss strengths and weaknesses of region inference and give results obtained with a prototype implementation.

We wish to point out that although the motivation for this work is practical, this is basically a theory paper. Our concern is for safety and correctness. The examples and experiments are not intended as final proof that our allocation scheme works well in practice (this we cannot asses fully at this point in time), although we hope that the discussion and experimental results in Section 6 will convince the reader that the scheme is not obviously flawed from a practical point of view.

2 Source and target languages

Before defining the source and target languages formally, let us look at two examples which illustrate the translation.

Example 1. Function values Consider the expression

$$e \equiv \texttt{let } x \texttt{ = (2,3) in } \lambda\, y\, .\texttt{(fst } x\texttt{, } y\texttt{) end 5}$$

This is an application expression; the operator evaluates to the function $\lambda\, y\, .\texttt{(2,}y\texttt{)}$, so the whole expression evaluates to the pair `(2,5)`. Using region inference, e is translated into

```
e′  ≡  letregion ρ4, ρ5
       in letregion ρ6
          in let x = (2 at ρ2, 3 at ρ6) at ρ4
             in (λy.(fst x, y) at ρ1) at ρ5
             end
          end
          5 at ρ3
       end
```

Here we see that the lifetime of the region, which contains 3, namely ρ_6, is shorter than the lifetime of the function, which is put in ρ_5. Of course, this is all in order, for when the function is applied, only the first component of the pair is needed. The region variables ρ_1, ρ_2 and ρ_3 are free in the target expression. That is because they occur in the type of the target expression; they must be allocated at runtime before evaluation of the target expression begins. The details of the evaluation of e' appear at the end of this section (see page 9). □

Example 2. Region polymorphism The target language permits special functions, *region functions*, which take regions as parameters and produces ordinary functions as results. Region functions are said to be *region-polymorphic*. When defining a recursive function, different invocations of the function can use different regions. For an example, 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
```

This 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. The region function *fib* is located in ρ_2. When called with actual regions r_1 and r_2 as arguments, it produces a function, f, say, which can then be applied to an integer in region r_1. This happens twice in the example; the first function closure is put in ρ_7, the second in ρ_{10}. Looking at the body of *fib*, we see that region inference essentially has discovered the allocations and deallocations that would happen in a traditional stack-based implementation. When executed, the target program performs a total of 15030 value allocations within just 75 memory cells. The resulting memory consists of a single region, ρ_1, which contains just one number, namely the 15th Fibonacci number. □

```
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 ρ10 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.

2.1 The source language

The grammar for the source language is

$$\begin{array}{rcl} e & ::= & x \mid \lambda x.e \mid e_1 e_2 \mid \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \\ & \mid & \texttt{letrec } f\texttt{(}x\texttt{)} \texttt{ = } e_1 \texttt{ in } e_2 \mid \texttt{(}e_1\texttt{,}e_2\texttt{)} \mid \texttt{fst } e \mid \texttt{snd } e \end{array}$$

It defines the syntactic class SExp of *(source) expressions*, e. In the grammar, x and f range over the set Var of *(value) variables*. For brevity, we omit constants, conditionals and arithmetic operators from the grammar; Also, we often drop the `end` terminating `let`, `letrec` and `letregion`. We keep pairs and projections, however, since they have interesting properties in connection of the safety of the translation scheme.

The dynamic semantics of the source language is entirely standard. The semantic objects appear in Figure 3. An *environment* is a finite map from variables to values.[1] The

[1]Some terminology concerning finite maps: 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.

$$
\begin{array}{rll}
v \in \mathrm{Val} = \mathrm{Int} \cup \mathrm{Clos} \cup \mathrm{RecClos} \cup \mathrm{Pair} & & \text{value} \\
\langle x, e, E \rangle \in \mathrm{Clos} = \mathrm{Var} \times \mathrm{SourceExp} \times \mathrm{Env} & & \text{closure} \\
\langle x, e, E, f \rangle \in \mathrm{RecClos} = \mathrm{Var} \times \mathrm{SourceExp} \times \mathrm{Env} \times \mathrm{Var} & & \text{recursive closure} \\
(v_1, v_2) \in \mathrm{Pair} = \mathrm{Val} \times \mathrm{Val} & & \text{pair} \\
E \in \mathrm{Env} = \mathrm{Var} \xrightarrow{\text{fin}} \mathrm{Val} & & \text{environment}
\end{array}
$$

Figure 3: Semantic objects of the dynamic semantics for the source language

rules occur below. Note that there are two classes of closures; the first, Clos, represents non-recursive functions, whereas the second, RecClos, represents recursive functions. The fourth component of a recursive closure is the name of the recursive function. This is used when the recursive function is applied, to "unroll" the closure once — see rule (4).

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

$$\frac{E(x) = v}{E \vdash x \to v} \tag{1}$$

$$\frac{}{E \vdash \lambda x.e \to \langle x, e, E \rangle} \tag{2}$$

$$\frac{E \vdash e_1 \to \langle x_0, e_0, E_0 \rangle \qquad E \vdash e_2 \to v_2 \qquad E_0 + \{x_0 \mapsto v_2\} \vdash e_0 \to v}{E \vdash e_1\ e_2 \to v} \tag{3}$$

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

$$\frac{E \vdash e_1 \to v_1 \qquad E + \{x \mapsto v_1\} \vdash e_2 \to v}{E \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \to v} \tag{5}$$

$$\frac{E + \{f \mapsto \langle x, e_1, E, f \rangle\} \vdash e_2 \to v}{E \vdash \texttt{letrec}\ f\texttt{(}x\texttt{)}\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \to v} \tag{6}$$

$$\frac{E \vdash e_1 \to v_1 \qquad E \vdash e_2 \to v_2}{E \vdash \texttt{(}e_1\texttt{,}e_2\texttt{)} \to (v_1, v_2)} \tag{7}$$

$$\frac{E \vdash e \to (v_1, v_2)}{E \vdash \texttt{fst}\ e \to v_1} \tag{8}$$

$$\frac{E \vdash e \Rightarrow (v_1, v_2)}{E \vdash \texttt{snd}\ e \to v_2} \tag{9}$$

2.2 The target language

We assume a denumerably infinite set RegVar = $\{\rho_1, \rho_2, \ldots\}$ of *region variables*. We use ρ to range over region variables. We also assume a denumerably infinite set RegName = $\{\texttt{r1}, \texttt{r2}, \ldots\}$ of *region names*. We use r to range over region names. Region names serve to identify regions uniquely at runtime, i.e. the store is formally a map from region names to regions. In order to be able to give the semantics of the target language using substitution of region names for region variables, we allow region names in target expressions. Hence the syntactic class, Place, ranged over by p.

The grammar for the target language, TExp, is

$$
\begin{array}{rcl}
p & ::= & \rho \mid r \\
e & ::= & x \mid f\ \texttt{[}p_1\texttt{, } \cdots\texttt{, } p_n\texttt{]}\ \texttt{at}\ p \mid \lambda x.e\ \texttt{at}\ p \\
 & \mid & e_1 e_2 \mid \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \\
 & \mid & \texttt{letrec f[}\rho_1, \cdots, \rho_k\texttt{](}x\texttt{)}\ \texttt{at}\ p\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \\
 & \mid & \texttt{(}e_1\texttt{,}e_2\texttt{)}\ \texttt{at}\ p \mid \texttt{fst}\ e \mid \texttt{snd}\ e \\
 & \mid & \texttt{letregion}\ \rho\ \texttt{in}\ e
\end{array}
$$

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

We now present a store semantics for the target language. This semantics formalises the notion of region. The semantic objects of the target language appear in Figure 4. Closures (Clos) represent ordinary functions, whereas region closures (RegionClos) represent region functions. To make matters as simple as possible, all values are "boxed", i.e. all values are addresses.

A store is formally a finite map from region names to regions. An environment VE in the target language is called an *variable environment*; it maps value variables to target language values.

The notation Vec(N), where N is a set, means the set of finite sequences of members of N. In the inference rules we are brief about indirect addressing. Thus, whenever a is an address (r, o) (r is a region name and o an offset — see Figure 4), we write $s(a)$ to mean $s(r)(o)$ and we write $a \in \mathrm{Dom}(s)$ as a shorthand for $r \in \mathrm{Dom}(s)$ and $o \in \mathrm{Dom}(s(r))$. Similarly, when s is a store and sv is a storable value, we write $s + \{(r, o) \mapsto sv\}$ as a shorthand for $s + \{r \mapsto (s(r) + \{o \mapsto sv\})\}$.

Rule 11 is the evaluation rule for application of a region function. The notation $e[r_1/\rho_1, \cdots, r_k/\rho_k]$ means the simultaneous substitution of $r_1, \ldots, r_k$ for $\rho_1, \ldots, \rho_k$, respectively. The $\texttt{at}\ r$ indicates where the resulting function closure is to be put.

Another place where region names are substituted for region variables is in rule 19. We express the popping of a region r from a store s by writing "$s \setminus\setminus \{r\}$", which formally means the restriction of s to $\mathrm{Dom}(s) \setminus \{r\}$, i.e. s with the entire region r removed.

Rule 15 concerns region-polymorphic and (perhaps) 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

$o \in \text{OffSet}$	offset
$r \in \text{RegionName}$	region name
$s \in \text{Store} = \text{RegionName} \xrightarrow{\text{fin}} \text{Region}$	store
$reg \in \text{Region} = \text{OffSet} \xrightarrow{\text{fin}} \text{StoreVal}$	region
$v \in \text{TargetVal} = \text{Addr}$	value
$sv \in \text{StoreVal} = \text{Int} \cup \text{Clos} \cup \text{Pair} \cup \text{RegionClos}$	storable value
$\langle x, e, VE\rangle \in \text{Clos} = \text{Var} \times \text{TargetExp} \times \text{VarEnv}$	closure
$\langle \vec{\rho}, x, e, VE\rangle \in \text{RegionClos} = \text{Vec}\,(\text{RegVar}) \times \text{Var} \times \text{TargetExp} \times \text{VarEnv}$	region closure
$(v_1, v_2) \in \text{Pair} = \text{TargetVal} \times \text{TargetVal}$	pair
a or $(r, o) \in \text{Addr} = \text{RegionName} \times \text{OffSet}$	address
$VE \in \text{VarEnv} = \text{Var} \xrightarrow{\text{fin}} \text{TargetVal}$	variable environment

Figure 4: Semantic objects of the dynamic semantics for the target language

with `letrec` need not be recursive, so one can also use the `letrec` construct to define region functions, that produce non-recursive functions. Rule 15 creates a region closure in the store and handles recursion by creating a cycle: the binding for f points back to the region closure itself. With this handling of recursion, only one rule for function application, namely rule 13, is needed.

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

$$\frac{VE(x) = v}{s, VE \vdash x \to v, s} \tag{10}$$

$$\frac{\begin{array}{c} VE(f) = a, \quad s(a) = \langle \rho_1, \ldots, \rho_k, x, e, VE_0\rangle \\ o \notin \text{Dom}(s(r)) \qquad sv = \langle x, e[p_1/\rho_1, \cdots, p_k/\rho_k], VE_0\rangle \end{array}}{s, VE \vdash f \ \texttt{[}p_1\texttt{, } \cdots\texttt{, } p_k\texttt{]} \ \texttt{at} \ r \to (r, o), s + \{(r, o) \mapsto sv\}} \tag{11}$$

$$\frac{o \notin \text{Dom}(s(r))}{s, VE \vdash \lambda x.e \ \texttt{at} \ r \to (r, o), s + \{(r, o) \mapsto \langle x, e, VE\rangle\}} \tag{12}$$

$$\frac{\begin{array}{c} s, VE \vdash e_1 \to a_1, s_1 \qquad s_1(a_1) = \langle x_0, e_0, VE_0\rangle \\ s_1, VE \vdash e_2 \to v_2, s_2 \qquad s_2, VE_0 + \{x_0 \mapsto v_2\} \vdash e_0 \to v, s' \end{array}}{s, VE \vdash e_1 \ e_2 \to v, s'} \tag{13}$$

$$\frac{s, VE \vdash e_1 \to v_1, s_1 \qquad s_1, VE + \{x \mapsto v_1\} \vdash e_2 \to v, s'}{s, VE \vdash \texttt{let} \ x \ \texttt{=} \ e_1 \ \texttt{in} \ e_2 \to v, s'} \tag{14}$$

$$\frac{\begin{array}{c} o \notin \text{Dom}(s(r)) \qquad VE' = VE + \{f \mapsto (r, o)\} \\ s + \{(r, o) \mapsto \langle \rho_1, \cdots, \rho_k, x, e_1, VE'\rangle\}, VE' \vdash e_2 \to v, s' \end{array}}{s, VE \vdash \texttt{letrec f[}\rho_1, \ldots, \rho_k\texttt{](}x\texttt{) at } r \texttt{ = } e_1 \texttt{ in } e_2 \to v, s'} \tag{15}$$

$$\frac{s, VE \vdash e_1 \to v_1, s_1 \qquad s_1, VE \vdash e_2 \to v_2, s_2 \qquad o \notin \mathrm{Dom}(s_2(r))}{s, VE \vdash \texttt{(}e_1\texttt{,}e_2\texttt{)}\ \texttt{at}\ r \to (r, o), s_2 + \{(r, o) \mapsto (v_1, v_2)\}} \tag{16}$$

$$\frac{s, VE \vdash e \to a, s' \qquad s'(a) = (v_1, v_2)}{s, VE \vdash \texttt{fst}\ e \to v_1, s'} \tag{17}$$

$$\frac{s, VE \vdash e \to a, s' \qquad s'(a) = (v_1, v_2)}{s, VE \vdash \texttt{snd}\ e \to v_2, s'} \tag{18}$$

$$\frac{r \notin \mathrm{Dom}(s) \qquad s + \{r \mapsto \{\}\}, VE \vdash e[r/\rho] \to v, s_1}{s, VE \vdash \texttt{letregion}\ \rho\ \texttt{in}\ e \to v, s_1 \setminus\!\setminus \{r\}} \tag{19}$$

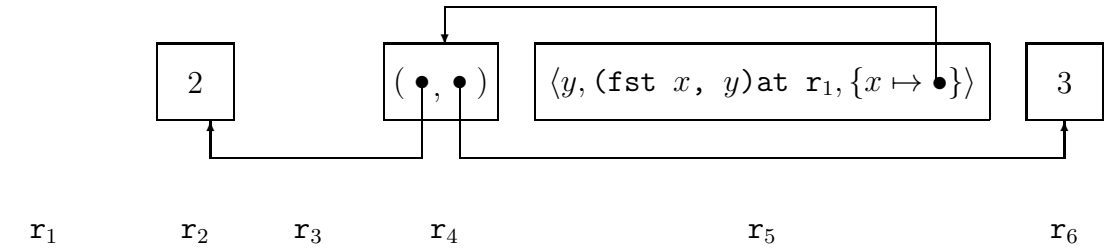
We can now state formally that the complete evaluation of an expression does not decrease the store. For arbitrary finite maps f_1 and f_2, we say that f_2 *extends* f_1, written $f_1 \subseteq f_2$, if $\mathrm{Dom}(f_1) \subseteq \mathrm{Dom}(f_2)$ and for all $x \in \mathrm{Dom}(f_1)$, $f_1(x) = f_2(x)$. 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 2.1 *If* $s, VE \vdash e \to v, s'$ *then* $\mathrm{Dom}(s) = \mathrm{Dom}(s')$ *and* $s \sqsubseteq s'$.

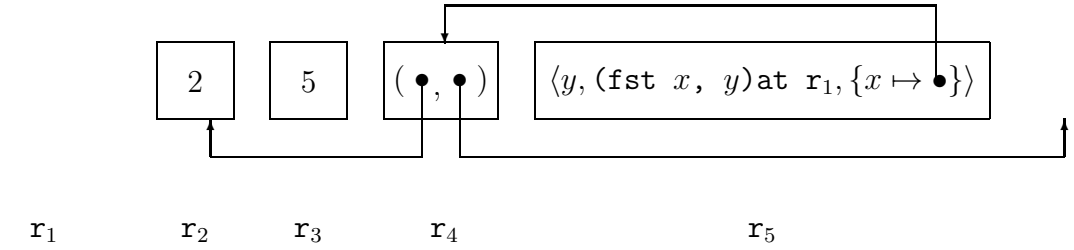
The proof is a straightforward induction on the depth of inference of $s, VE \vdash e \to v, s'$. The formula $\mathrm{Dom}(s) = \mathrm{Dom}(s')$ in Lemma 2.1 expresses that the store resulting from the elaboration has neither more nor less regions than the store in which the evaluation begins, although 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.

The above semantic model is a simplification of what happens in our prototype implementation. To avoid that a region, r, grows without limits, it is convenient to allow another method of storing a value v in r: first delete all the present contents of r and then store v at the beginning of r. This is similar to treating r as an updatable variable, something which is very useful in certain situations (see Section 6).

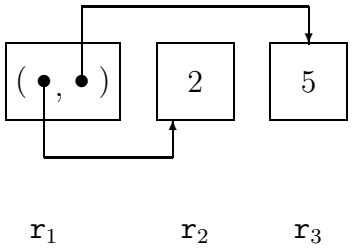
We end this section by completing Example 1. We show three snapshots from the evaluation of e', namely (a) just after the closure has been allocated; (b) just before the closure is applied and (c) at the end; we assume six regions with names $\mathtt{r}_1, \ldots \mathtt{r}_6$, which become bound to $\rho_1, \ldots, \rho_6$, respectively. Notice the dangling, but harmless, pointer at (b).



(a)



(b)



(c)

3 Region inference rules

In Section 3.1 we present the region inference rules. In Section 3.2 we state some formal properties of the region inference system.

3.1 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. In Example 1, μ is $((\texttt{int}, \rho_2) * (\texttt{int}, \rho_3), \rho_1)$, φ is $\{\mathbf{put}(\rho_1), \mathbf{put}(\rho_2), \mathbf{put}(\rho_3)\}$ and TE is empty.

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

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

Figure 5: Semantic objects of the Translation Semantics

of places. Indeed, the region inference rules allow one to put all values in a single region, even though this would normally not be a good idea.

The inference rules relies in an essential way on the use of type schemes (a la Milner[21]). In some accounts of polymorphism, type schemes are avoided with the aid of `let`-expansion.[2] In our system, `let`-expansion is not sufficient, for type schemes have operational importance. The fact that some function, f, takes a region ρ as parameter is represented formally by a quantification $\forall\rho$ in the type scheme for f. Moreover, the application of f to some actual region, ρ_a, results from an instantiation of ρ to ρ_a. In other words, since region names are passed around at runtime, region quantification in type schemes is a characteristic feature of our approach.

Region functions are not values, although ordinary functions are. That is why the rules allow sentences of the form $TE \vdash e \Rightarrow e' : \mu, \varphi$, but not sentences of the form $TE \vdash e \Rightarrow e' : \forall\vec{\rho}.\mu, \varphi$. The special status of region functions is represented in the semantics by a distinction between *compound type schemes*, which are used for region functions, and *simple type schemes*, which are used for (possibly polymorphic) values. Even a region function which must be applied to zero region arguments to produce a function, is not a value. Thus a compound type scheme of the special form $\forall\rho_1\cdots\rho_k.\sigma$ where σ is simple and k is 0, is not the same as the simple type scheme σ.

The semantic objects are defined in Figure 5.

[2] `let`-expansion consists of replacing every occurrence of an expression of the form `let` x `=` e_1 `in` e_2 by $e_2[e_1/x]$.

A *substitution* S is a triple (S_r, S_t, S_e), where S_r is a finite map from region variables to places, S_t is a finite map from type variables to types and S_e is a finite map from effect variables to arrow effects.[3] Its effect is to carry out the three substitutions simultaneously on the three kinds of variables. Substitutions can be applied to larger semantic objects containing type, region and effect variables. Substitutions can be composed.

For any compound type scheme $\sigma = \forall\rho_1 \cdots \rho_k \forall\alpha_1 \cdots \alpha_n \forall\epsilon_1 \cdots \epsilon_m.\tau$, and type τ', we say that τ' is *an instance of* σ *(via* S*)*, written $\sigma \geq \tau'$, if there exists a substitution $S = (\{\rho_1 \mapsto p_1, \ldots, \rho_k \mapsto p_k\}, \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}, \{\epsilon_1 \mapsto \epsilon'_1.\varphi_1, \ldots, \epsilon_m \mapsto \epsilon'_m.\varphi_m\})$ such that $S(\tau) = \tau'$. Instantiation $\sigma \geq \tau'$, in the case where σ is simple, is defined similarly.

$$\frac{TE(x) = (\sigma, p) \quad \sigma \text{ simple} \quad \sigma \geq \tau}{TE \vdash x \Rightarrow x : (\tau, p), \emptyset} \tag{20}$$

$$\frac{\begin{array}{c} TE(f) = (\sigma, p') \quad \sigma \text{ compound, i.e. } \sigma = \forall\rho_1 \cdots \rho_k.\sigma_1, \text{ where } \sigma_1 \text{ is simple} \\ \sigma \geq \tau \text{ via } S \qquad \varphi = \{\mathbf{get}(p'), \mathbf{put}(p)\} \end{array}}{TE \vdash f \Rightarrow f\,\texttt{[}S(\rho_1)\texttt{,}\ldots\texttt{,}S(\rho_k)\texttt{]}\ \texttt{at}\ p : (\tau, p), \varphi} \tag{21}$$

$$\frac{TE + \{x \mapsto \mu_1\} \vdash e \Rightarrow e' : \mu_2, \varphi \qquad \varphi \subseteq \varphi'}{TE \vdash \lambda x.e \Rightarrow \lambda x.e'\ \texttt{at}\ p : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, p), \{\mathbf{put}(p)\}} \tag{22}$$

$$\frac{TE \vdash e_1 \Rightarrow e'_1 : (\mu' \xrightarrow{\epsilon.\varphi} \mu, p), \varphi_1 \qquad TE \vdash e_2 \Rightarrow e'_2 : \mu', \varphi_2}{TE \vdash e_1\ e_2 \Rightarrow e'_1\ e'_2 : \mu, \varphi \cup \varphi_1 \cup \varphi_2 \cup \{\epsilon, \mathbf{get}(p)\}} \tag{23}$$

$$\frac{\begin{array}{c} TE \vdash e_1 \Rightarrow e'_1 : (\tau_1, p_1), \varphi_1 \\ TE + \{x \mapsto (\sigma, p_1)\} \vdash e_2 \Rightarrow e'_2 : \mu, \varphi_2 \qquad \sigma = \mathrm{TyEffGen}(TE, \varphi_1)(\tau_1) \end{array}}{TE \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \Rightarrow \texttt{let}\ x\ \texttt{=}\ e'_1\ \texttt{in}\ e'_2 : \mu, \varphi_1 \cup \varphi_2} \tag{24}$$

$$\frac{\begin{array}{c} TE + \{f \mapsto (\forall\vec{\rho}\forall\vec{\epsilon}.\tau, p)\} \vdash \lambda\, x.e_1 \Rightarrow \lambda\, x.e'_1\ \texttt{at}\ p : (\tau, p), \varphi_1 \\ \forall\vec{\rho}\forall\vec{\epsilon}.\tau = \mathrm{RegEffGen}(TE, \varphi_1)(\tau) \\ \sigma' = \mathrm{RegTyEffGen}(TE, \varphi_1)(\tau) \\ TE + \{f \mapsto (\sigma', p)\} \vdash e_2 \Rightarrow e'_2 : \mu, \varphi_2 \end{array}}{\begin{array}{l} TE \vdash\ \texttt{letrec}\ f\texttt{(}x\texttt{)}\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \Rightarrow \\ \qquad \texttt{letrec}\ f\,\texttt{[}\vec{\rho}\texttt{](}x\texttt{)}\ \texttt{at}\ p\ \texttt{=}\ e'_1\ \texttt{in}\ e'_2 : \mu, \varphi_1 \cup \varphi_2 \end{array}} \tag{25}$$

$$\frac{TE \vdash e_1 \Rightarrow e'_1 : \mu_1, \varphi_1 \qquad TE \vdash e_2 \Rightarrow e'_2 : \mu_2, \varphi_2}{TE \vdash \texttt{(}e_1\texttt{,}e_2\texttt{)} \Rightarrow \texttt{(}e'_1\texttt{,}e'_2\texttt{)}\ \texttt{at}\ p : (\mu_1 * \mu_2, p), \varphi_1 \cup \varphi_2 \cup \{\mathbf{put}(p)\}} \tag{26}$$

[3]The reason that S_e maps effect variables to arrow effects, rather than effects, will become clear shortly.

$$\frac{TE \vdash e \Rightarrow e' : (\mu_1 * \mu_2, p), \varphi}{TE \vdash \texttt{fst}\ e \Rightarrow \texttt{fst}\ e' : \mu_1, \varphi \cup \{\mathbf{get}(p)\}} \tag{27}$$

$$\frac{TE \vdash e \Rightarrow e' : (\mu_1 * \mu_2, p), \varphi}{TE \vdash \texttt{snd}\ e \Rightarrow \texttt{snd}\ e' : \mu_2, \varphi \cup \{\mathbf{get}(p)\}} \tag{28}$$

$$\frac{TE \vdash e \Rightarrow e' : \mu, \varphi \qquad \varphi' = \mathrm{Observe}(TE, \mu)(\varphi) \quad \{\rho_1, \ldots, \rho_k\} = \mathrm{frv}(\varphi \setminus \varphi')}{TE \vdash e \Rightarrow \texttt{letregion}\ \rho_1 \cdots \rho_k\ \texttt{in}\ e' : \mu, \varphi'} \tag{29}$$

Rule 20 is essentially the usual instantiation rule for polymorphic variables in ML-like type systems[7]. Note that the effect of referring to x is empty; it is only if we use the value of x (as in the expression x`+1` or `fst` x) that we need to access x in the store.

Rule 21 concerns variables with a compound type scheme, i.e. a region polymorphic function f. Again, we can take an arbitrary instance of the type scheme, but this time the instantiation of the bound region variables is made explicit in the target expression. The resulting effect includes $\mathbf{get}(p')$ and $\mathbf{put}(p)$, for we must access the region closure in p' and create an ordinary function closure in p. Note that the rule does not say which p to choose.

In rule 22, μ_1 is regarded as a type scheme. Whenever a type and place is regarded as a type scheme, it is always considered to be simple. The effect of *creating* the function closure at place p is simply $\{\mathbf{put}(p)\}$. The effect φ is the effect of *calling* the function. It is therefore sometimes called the *latent* effect. 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.

Note that the latent effect moves onto the arrow in the function type. Also, it is "tagged" by an effect variable, to form an *arrow effect*, $\epsilon.\varphi$. This tagging is useful for two purposes. First, effect variables can be quantified, when type schemes are formed; hence different uses of a polymorphic function can be given different effects[31]. Second, the tagging is useful for unification purposes. (The use of effect variables in arrow effects is similar to the use of "row variables" or "extension variables" in work on typing of flexible records[26,32].) Notice that because substitutions map effect variables to arrow effects, applying a substitution to an arrow effect yields an arrow effect: $S(\epsilon.\varphi) = \epsilon'.(\varphi' \cup S(\varphi))$, where $\epsilon'.\varphi' = S(\epsilon)$.

In rule 23 we see that the latent effect is brought out when the function is applied. The $\mathbf{get}(p)$ in the resulting effect is due to the fact that we must access the closure (which is at place p) to do the function application.

We now introduce notation needed for the rules concerning introduction of type schemes. For any semantic object A, $\mathrm{frv}(A)$ denotes the set of region variables that occur free in A, $\mathrm{frn}(A)$ denotes the set of region names that occur free in A, $\mathrm{ftv}(A)$ denotes the set of type variables that occur free in A, and $\mathrm{fev}(A)$ denotes the set of effect variables that occur free in A. Further, let A be a semantic object or assembly of semantic objects, let τ be at type and let $\{\alpha_1, \ldots, \alpha_n\} = \mathrm{ftv}(\tau) \setminus \mathrm{ftv}(A)$, let $\{\rho_1, \ldots, \rho_k\} = \mathrm{frv}(\tau) \setminus \mathrm{frv}(A)$

and let $\{\epsilon_1, \ldots, \epsilon_m\} = \mathrm{fev}(\tau) \setminus \mathrm{fev}(A)$. Then we define the following three operations for forming type schemes:

$$\begin{array}{rcll} \mathrm{RegTyEffGen}(A)(\tau) & = & \forall\rho_1 \cdots \rho_k \forall\alpha_1 \cdots \alpha_n \forall\epsilon_1 \cdots \epsilon_m.\tau & \text{(compound)} \\ \mathrm{RegEffGen}(A)(\tau) & = & \forall\rho_1 \cdots \rho_k \forall\epsilon_1 \cdots \epsilon_m.\tau & \text{(compound)} \\ \mathrm{TyEffGen}(A)(\tau) & = & \forall\alpha_1 \cdots \alpha_n \forall\epsilon_1 \cdots \epsilon_m.\tau & \text{(simple)} \end{array}$$

Rule 24 for `let`-expressions is similar to the usual rule for ML-polymorphism[7]. Notice that one is not allowed to quantify any region variables in connection with `let`. The reason for this is that introducing region quantification blocks the evaluation of e_1. If the language is extended with side-effects, blocking evaluation would change the semantics of programs. We only allow region quantification in connection with `letrec`, where evaluation would be blocked anyway, see rule 25. Here RegEffGen allows generalisation on region and effect variables, but not on type variables. Thus a recursive function can be used region-polymorphically, but not type-polymorphically, inside its own definition. In e_2, however, f is polymorphic in types, regions and effects.

Rule 29 concerns the introduction of `letregion` expressions. The basic idea, which goes back to early work on effect systems[6], is this. Suppose $TE \vdash e \Rightarrow e' : \mu, \varphi$ and assume that ρ is a region variable which occurs free in φ, but does not occur free in TE or in μ. *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.* The correctness of this idea requires a proof, we believe, and we shall provide one in Section 4. One of the reasons why the idea works is that types are rich enough to contain all the region variables that values of that type need in order to exist. In particular, it is crucial that the type of a function contains the region variables for the regions that may be accessed if the function is called.

Since ρ is no longer needed, we can introduce a "`letregion` ρ `in` $\cdots$ `end`" around e' and discharge any effect containing ρ from φ.

Thus, following [31], for every effect φ and semantic object A, we define *the observable part of φ with respect to A*, written $\mathrm{Observe}(A)(\varphi)$, to be the following subset of φ:

$$\begin{array}{rcl} \mathrm{Observe}(A)(\varphi) & = & \{\mathbf{put}(p) \in \varphi \mid p \in \mathrm{frv}(A) \cup \mathrm{frn}(A)\} \\ & & \cup \{\mathbf{get}(p) \in \varphi \mid p \in \mathrm{frv}(A) \cup \mathrm{frn}(A)\} \\ & & \cup \{\epsilon \in \varphi \mid \epsilon \in \mathrm{fev}(A)\} \end{array}$$

3.2 Lemmas about the region inference rules

Every expression e which is well-typed according to Milner's type discipline[21,7] can be translated using the region inference rules. We shall not prove this in detail, but explain informally why this is the case. Let us write $\vdash_{ML} e : \tau$ to mean that e has type τ according to Milner's type discipline. (For ease of comparison, we assume that all quantification of type variables is done in connection with `let` and `letrec` and that instantiation of type schemes is only done at variables, as for example in the Definition of Standard ML[22].) Let *erase* be the function which takes one of our semantic objects and produces an object in the ML type system by removing all effects and regions. We claim that if $\vdash_{ML} e : \tau$ then there exist e', μ and φ such that $\vdash e \Rightarrow e' : \mu, \varphi$ and $\mathit{erase}(\mu) = \tau$. To obtain these from a

proof of $\vdash_{ML} e : \tau$, choose an arbitrary region variable, ρ, and an arbitrary effect variable, ϵ, put "`at` ρ" on all value-constructing subexpressions of e and put $\epsilon.\{\mathbf{get}(\rho), \mathbf{put}(\rho), \epsilon\}$ on every function arrow; φ we be a subset of $\{\mathbf{get}(\rho), \mathbf{put}(\rho)\}$. (This, incidentally, is probably the worst possible region annotation, from a practical point of view.)

Our first lemma states that region inference is preserved under substitution:

Lemma 3.1 *If $TE \vdash e \Rightarrow e' : \mu, \varphi$ then $S(TE) \vdash e \Rightarrow S(e') : S(\mu), S(\varphi)$, for all substitutions S.*

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

We end this subsection with a lemma which states that translation is preserved under the operation of making assumed type schemes more type-polymorphic:

Lemma 3.2 *Let σ be a simple type scheme $\forall\vec{\epsilon}.\tau$ and let σ' be $\forall\vec{\alpha}\forall\vec{\epsilon}.\tau$, for some $\vec{\alpha}$, or let σ be a compound type scheme $\forall\vec{\rho}\forall\vec{\epsilon}.\tau$ and let σ' be $\forall\vec{\rho}\forall\vec{\alpha}\forall\vec{\epsilon}.\tau$, for some $\vec{\alpha}$. If $TE + \{f \mapsto (\sigma, p)\} \vdash e \Rightarrow e' : \mu, \varphi$ then $TE + \{f \mapsto (\sigma', p)\} \vdash e \Rightarrow e' : \mu, \varphi$.*

We omit the proof, which is an uncomplicated induction on the depth of inference of $TE + \{f \mapsto (\sigma, p)\} \vdash e \Rightarrow e' : \mu, \varphi$.[4]

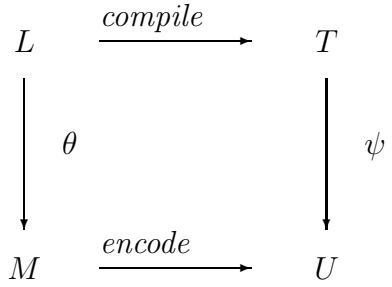
4 Region Inference is Safe

On the surface, our target language resembles the source language and its semantics appears to be a relatively straightforward store semantics, so one might reasonably ask whether it is necessary to prove the translation correct.

We believe that it is in fact crucial to prove the translation correct, for we know of no previous proof that it is always possible to stack-allocate all values, including functions. Some implementors of functional languages have to distinguish between "known" and "unknown" functions based on syntactic criteria, reserving stack allocation for the former; in our scheme, all functions are treated in a uniform manner. It is far from obvious that this is safe. In particular, the scheme naturally raises questions such as: "Is it really safe to deallocate a region when the region inference rules say that it can be done?" and "Is it really safe to have dangling references out of closures?" According to our proof, the answer to both questions is yes.

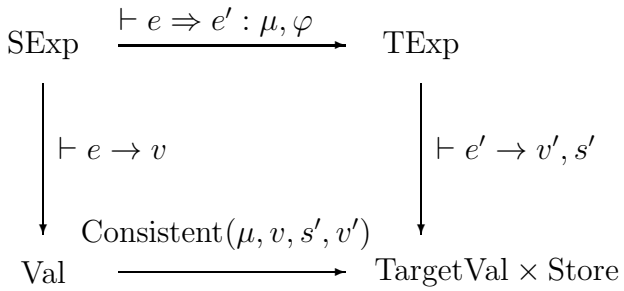
Going back to Morris[23], compiler correctness has been stated as the commutativity of a diagram of the form:

[4]For the inductive proof to go through, one has to prove something slightly stronger, than what we have stated, namely that typing is preserved even if more than one variable is made more type-polymorphic.



Here L is a set of source programs, T is a set of target programs, M is a set of source language meanings, U is a set of target language meanigs, θ and ψ are semantic functions for the source and target languages, *compile* is a function from source language programs to target language programs and *encode* is a map from source language meanings to target language meanings. The goal is to prove that the diagram commutes, i.e. that for all source programs l, $\psi(\mathit{compile}(l)) = \mathit{encode}(\theta(l))$.

The same idea applies when one uses operational rather than denotational semantics[8]. In our case, the "semantic functions" are the evaluation functions induced by the inference rules in Sections 2.1 and 2.2. Corresponding to the *compile* function, we have a relation, namely the relation defined by the region inference rules in Section 3. Corresponding to the *encode* function, we shall define a relation, called Consistent, between, among other things, values of the source language and values of the target language. If we restrict attention to closed expressions only, we hence consider the following diagram (where we have omitted empty stores and environments):



Following [8], we distinguish between *soundness* and *completeness* of a translation. Soundness says, roughly speaking, that every computation performed by a source program is simulated by the corresponding target program. In terms of the above diagrams, given the left and top arrow one can find arrow to put at the right and at the bottom to make the diagram commute. *Completeness*, on the other hand, says that whenever one has the top and the right arrow, one can find a left and a bottom arrow to make the diagram commute, i.e., every result computed by the target program represents a result computed by the source program.

In our case, soundness is the most critical property to prove. The fear is that, for example, soundness might not hold because the target program attempts to access a value in a region which has been deallocated and later filled with other values. Completeness is less problematic, in our view, for the following reason. It is easy to see that the source language semantics is monogenic, i.e. that for all E, e, v_1 and v_2, if $E \vdash e \to v_1$ and $E \vdash e \to v_2$ then $v_1 = v_2$. Similarly, the target language semantics is monogenic, up to the choice of fresh region names and offsets. Provided that soundness holds, the only way completeness could fail to hold would be that the source program diverged whereas the target program converged. We conjecture, without having proved it, that this is impossible. The evaluation order in the target language is exactly as in the source language; the handling of recursion is different in the two languages, but the representation of unfolding by cyclic closures is a standard technique. Henceforth, we shall focus on soundness only.

4.1 On the loss of consistency

In order to be able to formulate a soundness statement which also applies to expressions with free variables, let us assume that we also have a relation Consistent(TE, E, s, VE). Our first soundness conjecture is:

Conjecture 4.1 *If* $TE \vdash e \Rightarrow e' : \mu, \varphi$ *and* Consistent(TE, E, s, VE) *and* $E \vdash e \to v$ *then there exist* v' *and* s' *such that* $s, VE \vdash e' \to v', s'$ *and* Consistent(μ, v, s', v').

From a technical point of view, something is wrong with the above conjecture: φ is only mentioned in the premise $TE \vdash e \Rightarrow e' : \mu, \varphi$, so the theorem says nothing about what it means for an expression to have an effect. From an intuitive point of view, however, it seems that there is a much more basic and interesting difficulty with the above statement. One way of reading Conjecture 4.1 is that consistency is preserved under evaluation. It appears, however, that no matter how we try to define consistency, consistency is *not* preserved under evaluation; rather is appears to be *monotonically decreasing* during the evaluation. The saving factor is that there is always enough consistency left for the rest of the program to complete its computation without detecting the lost consistency.

Example 1 (page 9) illustrates this point. Recall that the closure indirectly contains a "dangling reference", at (b). Of course, this is all in order, since the reference will never be dereferenced. But we observe that the consistency that exists between the pair `(2,3)` and the value in $\mathtt{r}_4$ at (a), where the closure is *created*, no longer exists at the point (b), where the closure is *applied*.

This loss of consistency appears to be at variance with another syntactic approach to proving soundness, namely proving soundness by proving a subject reduction property[33]. Subject reduction is useful for proving that typing is *preserved* under reduction.

To express the idea that there is "enough" consistency left, we make the Consistent relation depend on an effect φ, which can be thought of as the effect of the rest of the program.

Conjecture 4.2 *If* $TE \vdash e \Rightarrow e' : \mu, \varphi$ *and* Consistent(TE, E, s, VE) w.r.t. $\varphi \cup \varphi'$ *and* $E \vdash e \to v$ *then there exist* v' *and* s' *such that* $s, VE \vdash e' \to v', s'$ *and* Consistent(μ, v, s', v')w.r.t. φ'.

Here $\varphi \cup \varphi'$ is the effect of the rest of the computation before the evaluation of e' is begun; after e' has produced v', the effect φ' is left. Notice that φ is the effect inferred for e'. One can say that soundness is expressed in "continuation style".

The statement that φ' stands for the effect of the rest of the program needs to be clarified. The rest of the program can allocate regions which are used for a while and then deallocated. References to these temporary regions are not included in φ'; in other words, φ' only refers to regions that are already allocated. One could say that φ' is the "net" effect of the rest of the program, but for brevity we shall just refer to φ' as the effect of the rest of the program.

4.2 Region Environments

A translation $TE \vdash e \Rightarrow e' : \mu, \varphi$ can be such that TE, e', μ and φ contain free region variables. Unless at least the region variables that occur free in φ are interpreted as region names that are in the domain of s, one cannot expect the evaluation $s, E \vdash e' \to \cdots$ to succeed.

A *region environment* to be a map $R : \text{RegVar} \to \text{Place}$. A region environment R *connects* φ *to* s, if $\text{frv}(R(\varphi)) = \emptyset$ and $\text{frn}(R(\varphi)) \subseteq \text{Dom}(s)$. Consistency must now be relative to a region environment, so the consistency relations now take the form Consistent(R, TE, E, s, VE) w.r.t. φ and Consistent(R, μ, v, s, v') w.r.t. φ. This leads us to the next soundness conjecture.

Conjecture 4.3 *If* $TE \vdash e \Rightarrow e' : \mu, \varphi$ *and* Consistent(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 *then there exist* v' *and* s' *such that* $s, VE \vdash R(e') \to v', s'$ *and* Consistent(R, μ, v, s', v') w.r.t. φ'.

4.3 On region environments and closures

Conjecture 4.3 is true, with a suitable definition of Consistent. However, if one tries to prove it directly, for example by induction on the depth of the proof of $E \vdash e \to v$, then one encounters an interesting problem in the case for function application. Without going into details, the problem is that the region environment, which exists at the point when a function is created need not be identical to the one that exists, when the function is applied. We therefore strengthen the statement of the soundness theorem as follows. Let us say that two region environments R_1 and R_2 *agree on effect* φ, if $R_1(\rho) = R_2(\rho)$, for all $\rho \in \text{frv}(\varphi)$.

Theorem 4.1 *If* $TE \vdash e \Rightarrow e' : \mu, \varphi$ *and* Consistent(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* $\varphi \cup \varphi'$ *then there exist* s' *and* v' *such that* $s, VE \vdash R'(e') \to v', s'$ *and* Consistent(R, μ, v, s', v') w.r.t. φ'.

Incidentally, because of the assumption " R' and R agree on $\varphi \cup \varphi'$ ", the second part of the conclusion is equivalent to " Consistent(R', μ, v, s', v') w.r.t. φ' " — this is a consequence of Lemma 4.2 below.

4.4 The definition of consistency

We now define three relations

$$\text{Consistent}(R, \mu, v, s, v') \text{ w.r.t. } \varphi$$

$$\text{Consistent}(R, (\sigma, p), v, s, v') \text{ w.r.t. } \varphi$$

$$\text{Consistent}(R, TE, E, s, VE) \text{ w.r.t. } \varphi$$

The first of these defines what it is for a source language value v to be consistent with a target language value v' with type and region μ in a store s and region environment R with respect to an effect φ. Intuitively, φ represent the effect of the rest of the program, i.e. of the computation that v' is consumed by. The second relation is a generalisation of the first, to cover type schemes. The third relation is the pointwise extension of the second relation. These relations are the maximal fixed point of a certain monotonic operator, F, which is defined in Appendix A.

For the purpose of the definition, let RegEnv denote the set of region environments and let

$$\begin{aligned} U_1 &= \text{RegEnv} \times \text{TypeAndPlace} \times \text{Val} \times \text{Store} \times \text{TargetVal} \times \text{Effect} \\ U_2 &= \text{RegEnv} \times (\text{TypeScheme} \times \text{Place}) \times \text{Val} \times \text{Store} \times \text{TargetVal} \times \text{Effect} \\ U_3 &= \text{RegEnv} \times \text{TyEnv} \times \text{Env} \times \text{Store} \times \text{VarEnv} \times \text{Effect} \\ U &= U_1 \times U_2 \times U_3 \end{aligned}$$

Let $v = (r, o)$ be a target language value; we then write "r of v" to mean the first component of v.

Definition 4.1 Consistent *is the largest relation on U satisfying the following:*

$$\boxed{\text{Consistent}(R, \mu, v, s, v') \text{ w.r.t. } \varphi}$$

if and only if, writing μ in the form (τ, p), if $\mathbf{get}(p) \in \varphi$ *then* $v' \in \text{Dom}(s)$ *and* r of $v' = R(p)$ *and*

1. *If v is an integer i then $\tau =$* `int` *and $s(v') = i$.*
2. *If v is a pair (v_1, v_2) then $\tau = \mu_1 * \mu_2$ and $s(v')$ is a pair (v'_1, v'_2) and* Consistent(R, μ_1, v_1, s, v'_1) w.r.t. φ *and* Consistent(R, μ_2, v_2, s, v'_2) w.r.t. φ*;*
3. *If v is a closure $\langle x, e, E\rangle$ then $s(v')$ is a closure $\langle x, e', VE\rangle$, for some e' and VE, and there exist TE, R' and e'' such that $TE \vdash \lambda x.e \Rightarrow \lambda x.e''$* `at` *$p : \mu, \{\mathbf{put}(p)\}$ and* Consistent(R, TE, E, s, VE) w.r.t. φ *and R' and R agree on φ and $R'(e'') = e'$*

4. *If* v *is a closure* $\langle x, e, E, f\rangle$ *then* $s(v')$ *is a closure* $\langle x, e', VE\rangle$*, for some* e' *and* VE*, and there exist* TE*,* σ*,* p'*,* R' *and* e'' *such that* $TE + \{f \mapsto (\sigma, p')\} \vdash \lambda x.e \Rightarrow \lambda x.e''$ `at` $p : \mu, \{\mathbf{put}(p)\}$ *and* $\mathrm{Consistent}(R, TE + \{f \mapsto (\sigma, p')\}, E + \{f \mapsto v\}, s, VE)$ w.r.t. φ *and* R' *and* R *agree on* φ *and* $R'(e'') = e'$

$$\boxed{\mathrm{Consistent}(R, (\sigma, p), v, s, v') \text{ w.r.t. } \varphi}$$

if and only if: if $\mathbf{get}(p) \in \varphi$ *then* $v' \in \mathrm{Dom}(s)$ *and* $(r \text{ of } v') = R(p)$ *and*

1. *If* σ *is simple then for all* τ *if* $\sigma \geq \tau$ *then* $\mathrm{Consistent}(R, (\tau, p), v, s, v')$ w.r.t. φ*;*
2. *If* σ *is compound, then* v *is a recursive closure* $\langle x, e, E, f\rangle$ *and* $s(v') = \langle \rho'_1, \cdots, \rho'_k, x, e', VE\rangle$*, for some* $\rho_1, \ldots, \rho_k$*,* e' *and* VE*, and there exist* TE*,* R' *and* e'' *such that* $\mathrm{Consistent}(R, TE + \{f \mapsto (\sigma, p)\}, E + \{f \mapsto v\}, s, VE)$ w.r.t. φ *and* σ *can be written in the form* $\forall\rho_1 \cdots \rho_k \forall\alpha_1 \cdots \alpha_n \forall\epsilon_1 \cdots \epsilon_m.\tau$*, where none of the bound variables occur free in* (TE, p)*, and* $TE + \{f \mapsto (\sigma, p)\} \vdash \lambda x.e \Rightarrow \lambda x.e''$ `at` $p : (\tau, p), \{\mathbf{put}(p)\}$ *and* R' *and* R *agree on* φ *and* $R'\langle \rho_1, \cdots, \rho_k, x, e'', VE\rangle = \langle \rho'_1, \cdots, \rho'_k, x, e', VE\rangle$

$$\boxed{\mathrm{Consistent}(R, TE, E, s, VE) \text{ w.r.t. } \varphi}$$

if and only if $\mathrm{Dom}\, TE = \mathrm{Dom}\, E = \mathrm{Dom}\, VE$ *and for all* $x \in \mathrm{Dom}\, TE$*,* $\mathrm{Consistent}(R, TE(x), E(x), s, VE(x))$ w.r.t. φ

A couple of comments on the above definition may be in order. The only way the effect φ plays a rôle is in the condition "if $\mathbf{get}(p) \in \varphi$ then $\cdots$", which qualifies the definitions of $\mathrm{Consistent}(R, (\tau, p), v, s, v')$ w.r.t. φ and $\mathrm{Consistent}(R, (\sigma, p), v, s, v')$ w.r.t. φ. In other words, if $\mathbf{get}(p) \notin \varphi$ then consistency holds vacuously, the idea being that the rest of the program will never access any value at place p.

Item 3, especially the need for R', was motivated briefly in Section 4.3. Item 4 is the natural extension of item 3 to cover recursion. The fact that we introduce σ and p' and require

$$\mathrm{Consistent}(R, TE + \{f \mapsto (\sigma, p)\}, E + \{f \mapsto v\}, s, VE) \text{ w.r.t. } \varphi \tag{30}$$

as opposed to

$$\mathrm{Consistent}(R, TE, E, s, VE) \text{ w.r.t. } \varphi \tag{31}$$

makes it important that consistency is the largest relation satisfying the conditions enumerated. If we only required (31), it would not be clear what type scheme to ascribe to f in the proof of Theorem 4.1 in the case for application of a recursive closure.[5]

Proceding to the definition of $\mathrm{Consistent}(R, (\sigma, p), v, s, v')$ w.r.t. φ, item 2 is the natural extension of the two previous cases to region closures. In any region closure

$$\langle \rho_1, \ldots, \rho_k, x, e', VE\rangle$$

[5]The point is that μ (or any type scheme obtained by quantifying some of the region variables free in μ) is not necessarily as general an assumption about f, as one needs, in order to infer a type for the function body.

the region variables $\rho_1, \ldots, \rho_k$ are regarded as bound variables with scope e'. Thus, in the formula

$$R'\langle \rho_1, \cdots, \rho_k, x, e'', VE\rangle = \langle \rho'_1, \cdots, \rho'_k, x, e', VE\rangle$$

the bound names in the region closure on the left may have to be renamed before R' is applied, in order to avoid capture, and equality is only postulated up to renaming of bound variables.

4.5 Lemmas about consistency

We shall now state certain lemmas concerning the interaction between consistency and evaluation. Once these are proved, the proof of Theorem 4.1 proceeds by an uncomplicated inductive argument. The lemmas deal with three issues, namely (in order of increasing complexity):

1. Conditions under which consistency is preserved;
2. The handling of recursion;
3. The handling of dangling references;

Preservation of consistency

The first lemma states that consistency is preserved under decreasing effect and increasing store.

Lemma 4.1 *If* Consistent(R, μ, v, s_1, v') w.r.t. φ_1 *and* $\varphi_2 \subseteq \varphi_1$ *and* $s_1 \sqsubseteq s_2$ *then* Consistent(R, μ, v, s_2, v') w.r.t. φ_2.

Recall that the effect in the consistency relation represents the needs of the rest of the program. It is not in general true that consistency is preserved under increasing effects. For example, for all addresses a, we have

$$\text{Consistent}(\{\}, (\texttt{int}, \rho), 3, \{\}, a) \text{ w.r.t. } \emptyset$$

but not

$$\text{Consistent}(\{\}, (\texttt{int}, \rho), 3, \{\}, a) \text{ w.r.t. } \{\mathbf{get}(\rho)\}$$

since the store is empty. Lemma 4.1 is a special case of the following lemma:[6]

Lemma 4.2 *If* Consistent(R_1, μ, 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* $s_1 \downarrow \text{frn}(R_2\varphi_2) \sqsubseteq s_2$ *then* Consistent(R_2, μ, v, s_2, v') w.r.t. φ_2.

Notice that the domain of s_1 need not be a subset of the domain of s_2 for Lemma 4.2 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_2, but not in φ_1 will be the variables of the local regions.

Lemma 4.2 is proved in Appendix B.

[6]For any finite map M and set A, we write $M \downarrow A$ to mean the restriction of M to A.

Recursion

The representation of recursive functions in the source language semantics differs from the representation of recursive functions in the target language semantics. In the source language we use closures which are "unrolled" each time the function is applied — see rule 4. In the target language a closure for a recursive function contains a pointer back to itself — see rule 15. To prove the correctness of our translation, we must show that the two representations are consistent at that crucial point where we "tie the knot" (i.e. create the cycle in the store). The following lemma states that under certain conditions, which are met when we create recursive closures, consistency of the two representations holds:

Lemma 4.3 *If* Consistent(R, TE, E, s, VE) w.r.t. φ *and* σ *is a compound type scheme* $\forall\vec{\rho}\forall\vec{\alpha}\forall\vec{\epsilon}.\tau$ *and no bound variable of* σ *is free in* (TE, p) *and* $TE + \{f \mapsto (\sigma, p)\} \vdash \lambda x.e_1 \Rightarrow \lambda x.e_1' \texttt{ at } p : (\tau, p), \{\mathbf{put}(p)\}$ *and* R' *and* R *agree on* φ *and* $R(p) = r$ *and* $r \in \mathrm{Dom}(s)$ *and* $o \notin \mathrm{Dom}(s(r))$ *and* $VE^+ = VE + \{f \mapsto (r, o)\}$ *and* $s^+ = s + \{(r, o) \mapsto R'\langle\vec{\rho}, x, e_1', VE^+\rangle\}$ *and* $TE^+ = TE + \{f \mapsto (\sigma, p)\}$ *and* $E^+ = E + \{f \mapsto \langle x, e_1, E, f\rangle\}$ *then* Consistent$(R, TE^+, E^+, s^+, VE^+)$ w.r.t. φ.

The proof of the above lemma relies in an essential way on Consistent being a maximal fixed point. The proof is found in Appendix C.

Dangling references

As we saw in Example 1 (page 3), region inference is sometimes so eager to collect regions, that dangling references arise. It is not surprising that dangling references complicate the soundness proof, but it is very instructive to see precisely where the difficulty appears. It turns out to be in the case concerning `letregion`, and only there. Assume (as in Theorem 4.1) that

$$TE \vdash e \Rightarrow \texttt{letregion } \rho \texttt{ in } e' : \mu, \varphi \tag{32}$$

$$\text{Consistent}(R, TE, E, s, VE) \text{ w.r.t. } \varphi \cup \varphi' \tag{33}$$

$$E \vdash e \to v \tag{34}$$

$$R \text{ and } R' \text{ agree on } \varphi \cup \varphi' \tag{35}$$

$$R \text{ connects } \varphi \cup \varphi' \text{ to } s \tag{36}$$

Now (32) could have been inferred by rule 29 on premises

$$TE \vdash e \Rightarrow e' : \mu, \varphi^+ \tag{37}$$

$$\varphi = \varphi^+ \setminus \{\mathbf{get}(\rho), \mathbf{put}(\rho)\} \tag{38}$$

We now have to allocate a region which is not in the domain of s, call it $\mathtt{r}$. Let $R^+ = R + \{\rho \mapsto \mathtt{r}\}$ and let $s^+ = s + \{\mathtt{r} \mapsto \{\}\}$. We would like to use induction on (37) together with, among other things,

$$\text{Consistent}(R^+, TE, E, s^+, VE) \text{ w.r.t. } \varphi^+ \cup \varphi \tag{39}$$

But, as we saw in Section 4.5, one cannot in general increase an effect and expect consistency to be preserved. Indeed, in the case at hand, `r` may have been in use previously, and s (or VE) may contain dangling references into that region. One might fear that when we add ρ to the effect, we suddenly have to account for consistency of values whose types in TE contain ρ. Fortunately, the side-condition on rule 29 demands that ρ not be free in TE! The following lemma states that, in these circumstances, (39) does hold.

Lemma 4.4 *If* Consistent(R, TE, E, s, VE) w.r.t. φ *and* $\rho \notin \mathrm{frv}(TE, \varphi)$, $r \notin \mathrm{Dom}(s)$ *and* $\varphi' \subseteq \{\mathbf{put}(\rho), \mathbf{get}(\rho)\} \cup \{\epsilon_1, \ldots, \epsilon_k\}$, *where* $\epsilon_1, \ldots, \epsilon_k$ *are effect variables* ($k \geq 0$) *then* Consistent$(R + \{\rho \mapsto r\}, TE, E, s + \{r \mapsto \{\}\}, VE)$ w.r.t. $\varphi' \cup \varphi$.

The proof of Lemma 4.4 appears in Appendix E. The proof is complicated by the fact that in the case where v is a closure, the type environment, which the definition of the consistency relation guarantees the existence of, could contain ρ free. It is then necessary to rename ρ in that type environment to a suitably new region variable. The next lemma states that this is possible.

The *support* of a substitution S, written $\mathrm{Supp}(S)$, is the set of variables β such that $S(\beta) \neq \beta$.

Definition 4.2 *For any semantic object A, let us say that a substitution S is a region renaming of A with respect to φ if $S \downarrow \mathrm{frv}(A)$ is injective and for all $\rho \in \mathrm{Supp}(S)$, $\rho \notin \mathrm{frv}(\varphi)$ and $S(\rho)$ is a region variable and $S(\rho) \notin \mathrm{frv}(\varphi)$.*

Lemma 4.5 *If* Consistent(R, μ, v, s, v') w.r.t. φ *and S is a region renaming of μ with respect to φ then* Consistent$(R, S(\mu), v, s, v')$ w.r.t. φ.

A proof of this lemma is found in Appendix D.

4.6 Proof of the correctness of the translation

We are now able to prove Theorem 4.1. The proof is fairly long, because there are many cases to consider, but thanks to the lemmas in the previous section, the individual cases are not too long.

The proof is by depth of the derivation of $E \vdash e \to v$. There are 9 cases, one for each rule in the dynamic semantics of the source language. In each case the translation statement $TE \vdash e \Rightarrow e' : \mu, \varphi$ must have been inferred by the application of a rule, which is uniquely determined by e and TE, followed by 0 or more applications of the `letregion` rule (i.e. rule 29). Applying rule 29 more than once is equivalent to applying it exactly once. For each of the 9 cases, there is therefore an inner case analysis with two cases; the first, let us call it the `letregion` case, is the one where the `letregion` rule has been applied; the second, let us call it the *syntax-directed* case, is where the `letregion` rule was not applied. It turns out that the `letregion` case follows from the syntax-directed case in the same way no matter what e is. Thus we need only do this case once. Below we first present the `letregion` case; then we deal with each of the 9 syntax-directed cases.

Readers, who would like to get a general feel for how the inductive argument goes (in particular for how the effect of subexpressions are propagated "backwards") may like to read the case concerning pairing (page 34) first.

In all the cases, we assume

$$TE \vdash e \Rightarrow e' : \mu, \varphi \tag{40}$$

$$\text{Consistent}(R, TE, E, s, VE) \text{ w.r.t. } \varphi \cup \varphi' \tag{41}$$

$$E \vdash e \to v \tag{42}$$

$$R \text{ connects } \varphi \cup \varphi' \text{ to } s \tag{43}$$

$$R' \text{ and } R \text{ agree on } \varphi \cup \varphi' \tag{44}$$

Inner proof case: the `letregion` rule was applied | Assume (40) takes the form

$$TE \vdash e \Rightarrow \texttt{letregion}\ \rho_1 \cdots \rho_k\ \texttt{in}\ e_1' : \mu, \varphi \tag{45}$$

Then (45) is inferred by rule 29 on the premises

$$TE \vdash e \Rightarrow e_1' : \mu, \varphi^+ \tag{46}$$

$$\varphi = \text{Observe}(TE, \mu)(\varphi^+) \tag{47}$$

$$\{\rho_1, \ldots, \rho_k\} = \text{frv}(\varphi^+ \setminus \varphi) \tag{48}$$

Without loss of generality we can choose $\rho_1, \ldots, \rho_k$ such that $\{\rho_1, \ldots, \rho_k\} \cap \text{frv}(\varphi') = \emptyset$ as well as (47)-(48). Thus $\{\rho_1, \ldots, \rho_k\} \cap \text{frv}(TE, \varphi \cup \varphi') = \emptyset$. Let $r_1, \ldots, r_k$ be distinct addresses none of which are in $\text{Dom}(s)$. Then by repeated application of Lemma 4.4 starting from (41) we get

$$\text{Consistent}(R^+, TE, E, s^+, VE) \text{ w.r.t. } \varphi^+ \cup \varphi' \tag{49}$$

where $R^+ = R + \{\rho_1 \mapsto r_1, \ldots, \rho_k \mapsto r_k\}$ and $s^+ = s + \{r_1 \mapsto \{\}, \ldots, r_k \mapsto \{\}\}$. Also by (43)

$$R^+ \text{ connects } \varphi^+ \cup \varphi' \text{ to } s^+ \tag{50}$$

and by (44)

$$R'^+ \text{ and } R^+ \text{ agree on } \varphi^+ \cup \varphi' \tag{51}$$

where $R'^+ = R' + \{\rho_1 \mapsto r_1, \ldots, \rho_k \mapsto r_k\}$. By the relevant syntax-directed proof case applied to (46), (49), (42), (50) and (51) there exist s_1' and v' such that

$$s^+, VE \vdash R'^+(e_1') \to v', s_1' \tag{52}$$

$$\text{Consistent}(R^+, \mu, v, s_1', v') \text{ w.r.t. } \varphi' \tag{53}$$

Now if $R'(\texttt{letregion}\ \rho_1 \cdots \rho_k\ \texttt{in}\ e_1') = \texttt{letregion}\ \rho_1' \cdots \rho_k'\ \texttt{in}\ e_1''$ then

$$R'^+ e_1' = e_1''[r_1/\rho_1', \ldots, r_k/\rho_k']$$

Thus, repeated application of rule 19 starting from (52) gives

$$s, VE \vdash R'(\texttt{letregion}\ \rho_1 \cdots \rho_k\ \texttt{in}\ e_1') \to v', s'$$

where $s' = s_1' \setminus\setminus \{r_1, \ldots, r_k\}$. Note that R^+ and R agree on φ' (as $\{\rho_1, \ldots, \rho_k\} \cap \mathrm{frv}(\varphi') = \emptyset$). Also, $s_1' \downarrow \mathrm{frn}(R\varphi') \sqsubseteq s'$ by (43). Then by Lemma 4.2 on (53) we get $\mathrm{Consistent}(R, \mu, v, s', v')$ w.r.t. φ', as required.

Variable, rule 1 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 (40) was inferred using rule 20. Then $e = e' = x$, for some variable x. Moreover, $TE(x) = (\sigma, p)$, for some p and simple σ. Also, $\mu = (\tau, p)$, for some τ with $\sigma \geq \tau$, and $\varphi = \emptyset$. The evaluation (42) must have been by rule 1, so we have $v = E(x)$. Let $s' = s$. By (40) and (41) we have $x \in \mathrm{Dom}\, VE$. Thus, letting $v' = VE(x)$, we have $s, VE \vdash R'(x) \to v', s'$, as desired. Finally

$$\mathrm{Consistent}(R, (\tau, p), v, s', v') \text{ w.r.t. } \varphi' \tag{54}$$

follows from (41) and the definition of Consistent.

Variable with compound type scheme Assume (40) was obtained by rule 21. Then e is of the form f, and e' is of the form $f\,\texttt{[}S\rho_1\texttt{,}\ldots\texttt{,}S\rho_k\texttt{]}\ \texttt{at}\ p$ and

$$TE \vdash f \Rightarrow f\,\texttt{[}S\rho_1\texttt{,}\ldots\texttt{,}S\rho_k\texttt{]}\ \texttt{at}\ p : (\tau, p), \varphi \tag{55}$$

was inferred by application of rule 21 to the premises

$$TE(f) = (\sigma, p') \quad \sigma = \forall \rho_1 \cdots \rho_k . \sigma_1 \quad \sigma_1 \text{ simple} \tag{56}$$

$$\sigma \geq \tau \text{ via } S \tag{57}$$

$$\varphi = \{\mathbf{get}(p'), \mathbf{put}(p)\} \tag{58}$$

Then (42) must have been inferred by rule (1), so we have $v = E(f)$. By (40) and $f \in \mathrm{Dom}(TE)$ we have

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

where $v_1' = VE(f)$. Since $\mathbf{get}(p') \in \varphi$, the definition of Consistent gives $v_1' \in \mathrm{Dom}(s)$ and r of $v_1' = R(p')$ and v is a recursive closure

$$v = \langle x_0, e_0, E_0, f_0 \rangle \tag{59}$$

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

$$\mathrm{Consistent}(R, TE_0 + \{f_0 \mapsto (\sigma, p')\}, E_0 + \{f_0 \mapsto v\}, s, VE_0) \text{ w.r.t. } \varphi \cup \varphi' \tag{60}$$

$$\sigma = \forall\rho_1 \cdots \rho_k \forall\alpha_1 \cdots \alpha_n \forall\epsilon_1 \cdots \epsilon_m . \tau_0 \tag{61}$$

$$TE_0 + \{f_0 \mapsto (\sigma, p')\} \vdash \lambda x_0.e_0 \Rightarrow \lambda x_0.e_0'' \texttt{ at } p' : (\tau, p'), \{\mathbf{put}(p')\} \tag{62}$$

$$\text{No bound variable of } \sigma \text{ is free in } (TE_0, p') \tag{63}$$

$$R_0 \text{ and } R \text{ agree on } \varphi \cup \varphi' \tag{64}$$

$$R_0(\langle \rho_1, \ldots, \rho_k, x_0, e_0'', VE_0 \rangle) = \langle \rho_1', \ldots, \rho_k', x_0, e_0', VE_0 \rangle \tag{65}$$

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

$$\{\rho_1, \ldots, \rho_k\} \cap \mathrm{frv}(\varphi') = \emptyset \tag{66}$$

Also without loss of generality, we may assume that e_0'' is chosen such that the set $A = \mathrm{frv}(e_0'') \setminus \mathrm{frv}(TE, \tau_0, p')$ satisfies $A \cap \mathrm{frv}(\mathrm{Rng}(S)) = \emptyset$. By (58), (43) and (44) we have $R'(p) \in \mathrm{Dom}(s)$. Let $r' = R'(p)$. Let o' be an offset not in $\mathrm{Dom}(s(r'))$. Let $v' = (r', o')$ and let $sv = \langle x_0, S'e_0', VE_0 \rangle$, where $S' = \{\rho_1' \mapsto R'(S\rho_1), \ldots, \rho_k' \mapsto R'(S\rho_k)\}$. Let $s' = s + \{(r', o') \mapsto sv\}$. It follows from rule 11 that

$$s, VE \vdash R'(f \texttt{ [} S\rho_1 \texttt{, } \ldots \texttt{, } S\rho_k \texttt{] at } p) \to v', s' \tag{67}$$

as desired. It remains to prove

$$\mathrm{Consistent}(R, (\tau, p), v, s', v') \text{ w.r.t. } \varphi' \tag{68}$$

We now consult the definition of Consistent. If $\mathbf{get}(p) \notin \varphi$, we are done. But even if $\mathbf{get}(p) \in \varphi$ we have $v' \in \mathrm{Dom}(s')$ and $r\mathrm{of}v' = r' = R(p)$ and v is the recursive closure (59), so we go to item 4 of the definition of Consistent. Certainly, $s'(v')$ is a closure, namely the above sv.

Let TE, σ and p' be be TE_0, σ and p', respectively. Further, let $e'' = S(e_0'')$ and let R_ν' be the map defined by:

$$R_\nu'(\rho) = \begin{cases} R'(\rho) & \text{if } \rho = S(\rho_i), \text{ for some } \rho_i \in \{\rho_1, \ldots, \rho_k\} \\ R_0(\rho) & \text{if } \rho \in \mathrm{frv}(\varphi') \cup A \end{cases} \tag{69}$$

We need to check that the two cases in the definition of R_ν' are not conflicting. Recall that $A \cap \mathrm{Rng}(S) = \emptyset$. Moreover, R' and R_0 agree on $\mathrm{Rng}(S) \cap \mathrm{frv}(\varphi')$, by (44) and (64). Thus (69) makes sense. By (63) and Lemma 3.1 on (62) we get

$$TE_0 + \{f_0 \mapsto (\sigma, p')\} \vdash \lambda x_0.e_0 \Rightarrow \lambda x_0.e'' \texttt{ at } p' : (\tau, p'), \{\mathbf{put}(p')\} \tag{70}$$

where we have used $S(p') = p'$ and that $\sigma \geq \tau$ via S. Thus also

$$TE_0 + \{f_0 \mapsto (\sigma, p')\} \vdash \lambda x_0.e_0 \Rightarrow \lambda x_0.e'' \texttt{ at } p : (\tau, p), \{\mathbf{put}(p)\} \tag{71}$$

From Lemma 4.1 on (60) we get

$$\mathrm{Consistent}(R, TE_0 + \{f_0 \mapsto (\sigma, p')\}, E_0 + \{f_0 \mapsto v\}, s', VE_0) \text{ w.r.t. } \varphi' \tag{72}$$

From (69) we have

$$R'_\nu \text{ and } R \text{ agree on } \varphi' \tag{73}$$

Following the definition of Consistent, it remains to prove $R'_\nu(e'') = S'(e'_0)$. In other words, we must prove

$$R'_\nu(S(e''_0)) = \{\rho'_1 \mapsto R'(S(\rho_1)), \ldots, \rho'_k \mapsto R'(S(\rho_k))\}(e'_0) \tag{74}$$

By (65), e''_0 and e'_0 are identical, except perhaps for their choice of free region variables. Thus in order to establish (74), it suffices to prove $R'_\nu(S\rho) = S'(\tilde{\rho})$, whenever ρ is a region variable occurring free in e''_0 and $\tilde{\rho}$ is the region variable that occurs at the corresponding position of e'_0. But that follows by case analysis. If $\rho = \rho_i$, for some $\rho_i \in \{\rho_1, \ldots, \rho_k\}$, then $\tilde{\rho} = \rho'_i$ and $R'_\nu(S(\rho)) = R'(S(\rho_i)) = S'(\rho'_i) = S(\tilde{\rho})$; otherwise, $R'_\nu(\rho) = R_0(\rho) = \tilde{\rho} = S'(\tilde{\rho})$. Thus

$$R'_\nu(e'') = S'(e'_0) \tag{75}$$

From (71), (72), (73) and (75) we finally have (68), the desired conclusion.

Lambda abstraction, rule 2 Assume (40) was inferred by rule 22; then (40) takes the following form:

$$TE \vdash \lambda x.e_1 \Rightarrow \lambda x.e'_1 \texttt{ at } p : \mu, \{\mathbf{put}(p)\} \tag{76}$$

Moreover, (42) was inferred by rule 2 yielding

$$v = \langle x, e_1, E \rangle \tag{77}$$

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 $s' = s + \{v' \mapsto \langle x, R'e'_1, VE \rangle\}$. By (44) we have $R'(p) = r$. Thus by rule 12 we have

$$s, VE \vdash R'(\lambda x.e'_1 \texttt{ at } p) \to v', s' \tag{78}$$

Notice that Consistent(R, TE, E, s', VE) w.r.t. φ', by Lemma 4.1. Thus, using R' for R_0 and e'_1 for e''_0 in the definition of Consistent, we have

$$\text{Consistent}(R, \mu, v, s', v') \text{ w.r.t. } \varphi' \tag{79}$$

as required.

Application of non-recursive closure, rule 3 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 (40) was inferred by rule 23 on premises

$$TE \vdash e_1 \Rightarrow e'_1 : (\mu' \xrightarrow{\epsilon.\varphi_0} \mu, p), \varphi_1 \tag{80}$$

$$TE \vdash e_2 \Rightarrow e'_2 : \mu', \varphi_2 \tag{81}$$

$$\varphi = \varphi_1 \cup \varphi_2 \cup \{\epsilon, \mathbf{get}(p)\} \cup \varphi_0 \tag{82}$$

Moreover, (42) was inferred by rule 3 on premises

$$E \vdash e_1 \to v_1, \quad v_1 = \langle x_0, e_0, E_0 \rangle \tag{83}$$

$$E \vdash e_2 \to v_2 \tag{84}$$

$$E_0 + \{x_0 \mapsto v_2\} \vdash e_0 \to v \tag{85}$$

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 (41), (43) and (44) we get

$$\text{Consistent}(R, TE, E, s, VE) \text{ w.r.t. } \varphi_1 \cup \varphi'_1 \tag{86}$$

$$R \text{ connects } \varphi_1 \cup \varphi'_1 \text{ to } s \tag{87}$$

$$R' \text{ and } R \text{ agree on } \varphi_1 \cup \varphi'_1 \tag{88}$$

By induction on (80), (86), (83), (87) and (88) there exist s_1 and v'_1 such that

$$s, VE \vdash R'(e'_1) \to v'_1, s_1 \tag{89}$$

$$\text{Consistent}(R, (\mu' \xrightarrow{\epsilon.\varphi_0} \mu, p), v_1, s_1, v'_1) \text{ w.r.t. } \varphi'_1 \tag{90}$$

Notice that $\mathbf{get}(p) \in \varphi'_1$. Thus, by the definition of Consistent, (90) tells us that $v'_1 \in \text{Dom}(s_1)$ and r of $v'_1 = R(p)$ and there exist e'_0, VE'_0, TE_0, e''_0 and R_0 such that

$$s_1(v'_1) = \langle x_0, e'_0, VE'_0 \rangle \tag{91}$$

$$TE_0 \vdash \lambda x_0.e_0 \Rightarrow \lambda x_0.e''_0 \texttt{ at } p : (\mu' \xrightarrow{\epsilon.\varphi_0} \mu, p), \{\mathbf{put}(p)\} \tag{92}$$

$$\text{Consistent}(R, TE_0, E_0, s_1, VE_0) \text{ w.r.t. } \varphi'_1 \tag{93}$$

$$R_0 \text{ and } R \text{ agree on } \varphi'_1 \tag{94}$$

$$R_0(e''_0) = e'_0 \tag{95}$$

Let $\varphi'_2 = \{\epsilon, \mathbf{get}(p)\} \cup \varphi_0 \cup \varphi'$, i.e. the effect that remains after the computation of e'_2. Note that $\varphi_2 \cup \varphi'_2 \subseteq \varphi \cup \varphi'$ and $s \sqsubseteq s_1$, so by Lemma 4.1 on (41) we have

$$\text{Consistent}(R, TE, E, s_1, VE) \text{ w.r.t. } \varphi_2 \cup \varphi'_2 \tag{96}$$

Also, from (43) and (44) we get

$$R \text{ connects } \varphi_2 \cup \varphi'_2 \text{ to } s_1 \tag{97}$$

$$R' \text{ and } R \text{ agree on } \varphi_2 \cup \varphi_2' \tag{98}$$

By induction on (81), (96), (84), (97) and (98) there exist s_2 and v_2' such that

$$s_1, VE \vdash R'(e_2') \to v_2', s_2 \tag{99}$$

$$\text{Consistent}(R, \mu', v_2, s_2, v_2') \text{ w.r.t. } \varphi_2' \tag{100}$$

Let $TE_0^+ = TE_0 + \{x_0 \mapsto \mu'\}$. By (92) there exists a φ_0' such that $\varphi_0' \subseteq \varphi_0$ and

$$TE_0^+ \vdash e_0 \Rightarrow e_0'' : \mu, \varphi_0' \tag{101}$$

By Lemma 4.1 on (93) and $\varphi_0' \subseteq \varphi_0$ we have

$$\text{Consistent}(R, TE_0, E_0, s_2, VE_0) \text{ w.r.t. } \varphi_0' \cup \varphi' \tag{102}$$

and by Lemma 4.1 on (100) and $\varphi_0' \subseteq \varphi_0$ we get

$$\text{Consistent}(R, \mu', v_2, s_2, v_2') \text{ w.r.t. } \varphi_0' \cup \varphi' \tag{103}$$

Let $E_0^+ = E_0 + \{x_0 \mapsto v_2\}$ and let $VE_0^+ = VE_0 + \{x_0 \mapsto v_2'\}$. Combining (102) and (103) we get

$$\text{Consistent}(R, TE_0^+, E_0^+, s_2, VE_0^+) \text{ w.r.t. } \varphi_0' \cup \varphi' \tag{104}$$

Also, by (43) and $s \sqsubseteq s_2$ we get

$$R \text{ connects } \varphi_0' \cup \varphi' \text{ to } s_2 \tag{105}$$

and by (94)

$$R_0 \text{ and } R \text{ agree on } \varphi_0' \cup \varphi' \tag{106}$$

Then by induction on (101), (104), (85), (105) and (106) there exist s' and v' such that

$$s_2, VE_0^+ \vdash R_0(e_0'') \to v', s' \tag{107}$$

$$\text{Consistent}(R, \mu, v, s', v') \text{ w.r.t. } \varphi' \tag{108}$$

But by (95) we have $R_0(e_0'') = e_0'$ so (107) reads

$$s_2, VE_0 + \{x_0 \mapsto v_2\} \vdash e_0' \to v', s' \tag{109}$$

From (89), (91), (99) and (109) we get

$$s, VE \vdash R'(e_1'\ e_2') \to v', s' \tag{110}$$

which together with (108) is the desired result.

Application of recursive closure, rule 4 | As in the previous case, $e \equiv e_1 e_2$, for some e_1 and e_2, and $e' \equiv e'_1 e'_2$, for some e'_1 and e'_2 and by rule 23 we have

$$TE \vdash e_1 \Rightarrow e'_1 : (\mu' \xrightarrow{\epsilon.\varphi_0} \mu, p), \varphi_1 \tag{111}$$

$$TE \vdash e_2 \Rightarrow e'_2 : \mu', \varphi_2 \tag{112}$$

$$\varphi = \varphi_1 \cup \varphi_2 \cup \varphi_0 \cup \{\mathbf{get}(p), \epsilon\} \tag{113}$$

Also, assume that (42) was inferred by application of rule 4 on premises

$$E \vdash e_1 \to v_1 \qquad v_1 = \langle x_0, e_0, E_0, f \rangle \tag{114}$$

$$E \vdash e_2 \to v_2 \tag{115}$$

$$E_0 + \{f \mapsto v_1\} + \{x_0 \mapsto v_2\} \vdash e_0 \to v \tag{116}$$

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 (41), (43) and (44) we have

$$\text{Consistent}(R, TE, E, s, VE) \text{ w.r.t. } \varphi_1 \cup \varphi'_1 \tag{117}$$

$$R \text{ connects } \varphi_1 \cup \varphi'_1 \text{ to } s \tag{118}$$

$$R' \text{ and } R \text{ agree on } \varphi_1 \cup \varphi'_1 \tag{119}$$

By induction on (111), (117), (114), (118) and (119), there exist v'_1 and s_1 such that

$$s, VE \vdash R'(e'_1) \to v'_1, s'_1 \tag{120}$$

$$\text{Consistent}(R, (\mu' \xrightarrow{\epsilon.\varphi_0} \mu, p), v_1, s_1, v'_1) \text{ w.r.t. } \varphi'_1 \tag{121}$$

Notice that $\mathbf{get}(p) \in \varphi'_1$. Thus by (121) and the definition of Consistent we have $v'_1 \in \text{Dom}(s_1)$ and r of $v'_1 = R(p)$ and there exist e'_0, VE_0, TE_0, σ_0, p_0, R_0 and e''_0 such that

$$s_1(v'_1) = \langle x_0, e'_0, VE_0 \rangle \tag{122}$$

$$TE_0 + \{f \mapsto (\sigma_0, p_0)\} \vdash \lambda x_0.e_0 \Rightarrow \lambda x_0.e''_0 \,\mathtt{at}\, p : (\mu' \xrightarrow{\epsilon.\varphi_0} \mu, p), \{\mathbf{put}(p)\} \tag{123}$$

$$\text{Consistent}(R, TE_0 + \{f \mapsto (\sigma_0, p_0)\}, E_0 + \{f \mapsto v_1\}, s_1, VE_0) \text{ w.r.t. } \varphi'_1 \tag{124}$$

$$R_0 \text{ and } R \text{ agree on } \varphi'_1 \tag{125}$$

$$R_0(e''_0) = e'_0 \tag{126}$$

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 2.1. Then, by Lemma 4.1 on (41) we have

$$\text{Consistent}(R, TE, E, s_1, VE) \text{ w.r.t. } \varphi_2 \cup \varphi'_2 \tag{127}$$

Moreover, (43) and (44) imply

$$R \text{ connects } \varphi_2 \cup \varphi'_2 \text{ to } s_1 \tag{128}$$

$$R' \text{ and } R \text{ agree on } \varphi_2 \cup \varphi_2' \tag{129}$$

By induction on (112), (127), (115), (128) and (129) there exist s_2 and v_2' such that

$$s_1, VE \vdash R'(e_2') \to v_2', s_2 \tag{130}$$

$$\text{Consistent}(R, \mu', v_2, s_2, v_2') \text{ w.r.t. } \varphi_2' \tag{131}$$

Let $TE_0^+ = TE_0 + \{f \mapsto (\sigma_0, p_0)\} + \{x_0 \mapsto \mu'\}$. From (123) we have the existence of an effect φ_0' with $\varphi_0' \subseteq \varphi_0$ and

$$TE_0^+ \vdash e_0 \Rightarrow e_0'' : \mu, \varphi_0' \tag{132}$$

By Lemma 4.1 on (124) we have

$$\text{Consistent}(R, TE_0 + \{f \mapsto (\sigma_0, p_0)\}, E_0 + \{f \mapsto v_1\}, s_2, VE_0) \text{ w.r.t. } \varphi_2' \tag{133}$$

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 (133) and (131) and $\varphi_0' \subseteq \varphi_0$ we have

$$\text{Consistent}(R, TE_0^+, E_0^+, s_2, VE_0^+) \text{ w.r.t. } \varphi_0' \cup \varphi' \tag{134}$$

From (125) we get

$$R_0 \text{ and } R \text{ agree on } \varphi_0' \cup \varphi' \tag{135}$$

By (43) and $s \sqsubseteq s_2$ we get

$$R \text{ connects } \varphi_0' \cup \varphi' \text{ to } s_2 \tag{136}$$

By induction on (132), (134), (116), (136) and (135) there exist s' and v' such that

$$s_2, VE_0^+ \vdash R_0(e_0'') \to v', s' \tag{137}$$

$$\text{Consistent}(R, \mu, v, s', v') \text{ w.r.t. } \varphi' \tag{138}$$

By (126), we have that (137) reads

$$s_2, VE_0^+ \vdash e_0' \to v', s' \tag{139}$$

Rule 13 on (120), (130), (122) and (137) gives

$$s, VE \vdash R'(e_1'\ e_2') \to v', s' \tag{140}$$

The desired conclusions are (140) and (138).

`let` expressions, rule 5 | Assume (40) was inferred by rule 24; then (40) takes the form

$$TE \vdash \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \Rightarrow \texttt{let } x \texttt{ = } e_1' \texttt{ in } e_2' : \mu, \varphi \tag{141}$$

Moreover, (40) and (42) must be inferred from the premises

$$TE \vdash e_1 \Rightarrow e_1' : (\tau_1, p_1), \varphi_1 \tag{142}$$

$$TE + \{x \mapsto (\sigma, p_1)\} \vdash e_2 \Rightarrow e_2' : \mu, \varphi_2 \qquad \sigma = \text{TyEffGen}(TE, \varphi_1)(\tau_1) \tag{143}$$

$$\varphi = \varphi_1 \cup \varphi_2 \tag{144}$$

$$E \vdash e_1 \to v_1 \tag{145}$$

$$E + \{x \mapsto v_1\} \vdash e_2 \to v \tag{146}$$

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 (41), (43) and (44) we have

$$\text{Consistent}(R, TE, E, s, VE) \text{ w.r.t. } \varphi_1 \cup \varphi_1' \tag{147}$$

$$R \text{ connects } \varphi_1 \cup \varphi_1' \text{ to } s \tag{148}$$

$$R' \text{ and } R \text{ agree on } \varphi_1 \cup \varphi_1' \tag{149}$$

By induction on (142), (147), (145), (148) and (149) there exist s_1 and v_1' such that

$$s, VE \vdash R'(e_1') \to v_1', s_1 \tag{150}$$

$$\text{Consistent}(R, (\tau_1, p_1), v_1, s_1, v_1') \text{ w.r.t. } \varphi_1' \tag{151}$$

By Lemma 4.1 on (41) we get

$$\text{Consistent}(R, TE, E, s_1, VE) \text{ w.r.t. } \varphi_1' \tag{152}$$

Let $\{\alpha_1, \ldots, \alpha_n\} = \text{ftv}(\mu_1) \setminus \text{ftv}(TE, \varphi_1)$ and let $\{\epsilon_1, \ldots, \epsilon_m\} = \text{fev}(\mu_1) \setminus \text{fev}(TE, \varphi_1)$. Then $\sigma = \forall \alpha_1 \cdots \alpha_n, \epsilon_1 \cdots \epsilon_m . \tau_1$. We shall now show that we can strengthen (151) to

$$\text{Consistent}(R, (\sigma, p_1), v_1, s_1, v_1') \text{ w.r.t. } \varphi_1' \tag{153}$$

Take any τ' with $\tau' \leq \sigma$. Then there exists a substitution of the form $S = \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\} \cup \{\epsilon_1 \mapsto \epsilon_1'.\varphi_1, \ldots, \epsilon_m \mapsto \epsilon_m'.\varphi_m\}$ with $S(\tau_1) = \tau'$. Since no region variable is in the domain of S, S is a region renaming of (τ_1, p_1) with respect to φ_1. Thus by Lemma 4.5 on (151) we have $\text{Consistent}(R, \mu', v_1, s_1, v_1')$ w.r.t. φ_1'. Since we can do this for every $\tau' \leq \sigma$, we have (153) by the definition of Consistent. Combining (152) and (153) we get

$$\text{Consistent}(R, TE + \{x \mapsto \sigma\}, E + \{x \mapsto v_1\}, s_1, VE + \{x \mapsto v_1'\}) \text{ w.r.t. } \varphi_2 \cup \varphi' \tag{154}$$

By (43) and (44) and $s \sqsubseteq s_1$ we have

$$R \text{ connects } \varphi_2 \cup \varphi' \text{ to } s_1 \tag{155}$$

$$R' \text{ and } R \text{ agree on } \varphi_2 \cup \varphi' \tag{156}$$

By induction on (143), (154), (146), (155) and (156) there exist s' and v' such that

$$s_1, VE + \{x \mapsto v_1'\} \vdash R'(e_2') \to v', s' \tag{157}$$

$$\text{Consistent}(R, \mu, v, s', v') \text{ w.r.t. } \varphi' \tag{158}$$

Here (158) is one of the desired results. Moreover, by rule 14 on (150) and (157) we get the desired

$$s, VE \vdash R'(\texttt{let } x \texttt{ = } e_1' \texttt{ in } e_2') \to v, s'$$

`letrec`, rule 6 In this case (40) takes the form

$$\begin{array}{l} TE \vdash \texttt{letrec } f\texttt{(}x\texttt{) = } e_1 \texttt{ in } e_2 \Rightarrow \\ \quad \texttt{letrec } f\texttt{[}\rho_1\texttt{,...,}\rho_k\texttt{](}x\texttt{) at } p \texttt{ = } e_1' \texttt{ in } e_2' : \mu, \varphi \end{array} \tag{159}$$

and is inferred by application of rule 25 to the premises

$$TE + \{f \mapsto (\forall \rho_1 \cdots \rho_k \forall \vec{\epsilon}.\tau, p)\} \vdash \lambda\, x.e_1 \Rightarrow \lambda\, x.e_1' \texttt{ at } p : (\tau, p), \varphi_1 \tag{160}$$

$$\forall \rho_1 \cdots \rho_k \forall \vec{\epsilon}.\tau = \text{RegEffGen}(TE, \varphi_1)(\tau) \tag{161}$$

$$\sigma' = \text{RegTyEffGen}(TE, \varphi_1)(\tau) \tag{162}$$

$$TE + \{f \mapsto (\sigma', p)\} \vdash e_2 \Rightarrow e_2' : \mu, \varphi_2 \tag{163}$$

$$\varphi = \varphi_1 \cup \varphi_2 \tag{164}$$

Also, (42) was inferred by rule 6 on the premise

$$E + \{f \mapsto \langle x, e_1, E, f\rangle\} \vdash e_2 \to v \tag{165}$$

Since (160) must have been inferred by rule 22, we have $\varphi_1 = \{\mathbf{put}(p)\}$. By (43) and (44) we have $R'(p) = R(p) \in \text{Dom}(s)$. Let $r_1 = R(p)$. Let o_1 be an offset with $o_1 \notin \text{Dom}(s(r_1))$. Let $v_1 = (r_1, o_1)$. Let $VE^+ = VE + \{f \mapsto v_1)\}$ and let $s^+ = s + \{v_1 \mapsto R'\langle \rho_1, \ldots, \rho_k, x, e_1', VE^+\rangle$. By Lemma 3.2 on (161), (162) and (160) we have that

$$TE + \{f \mapsto (\sigma', p)\} \vdash \lambda x.e_1 \Rightarrow \lambda x.e_1' \texttt{ at } p : (\tau, p), \varphi_1 \tag{166}$$

Let $TE^+ = TE + \{f \mapsto (\sigma', p)\}$ and let $E^+ = E + \{f \mapsto \langle x, e_1, E, f\rangle\}$. By Lemma 4.3 we have Consistent$(R, TE^+, E^+, s^+, VE^+)$ w.r.t. φ. Then by Lemma 4.1,

$$\text{Consistent}(R, TE^+, E^+, s^+, VE^+) \text{ w.r.t. } \varphi_2 \cup \varphi' \tag{167}$$

Also, by (43) and (44) we get

$$R \text{ connects } \varphi_2 \cup \varphi' \text{ to } s \tag{168}$$

$$R' \text{ and } R \text{ agree on } \varphi_2 \cup \varphi' \tag{169}$$

By induction on (163), (167), (165), (168) and (169) there exist s' and v' such that

$$s^+, VE^+ \vdash R'(e_2') \to v', s' \tag{170}$$

$$\text{Consistent}(R, \mu, v, s', v') \text{ w.r.t. } \varphi' \tag{171}$$

From (170) and rule 15 we get

$$s, VE \vdash R'(\texttt{letrec } f\texttt{[}\rho_1\texttt{,}\ldots\texttt{,}\rho_k\texttt{](}x\texttt{) at } p \texttt{ = } e_1' \texttt{ in } e_2') \to v', s' \tag{172}$$

Now (171) and (172) are the desired results.

Pairing, rule 7 In this case (40) takes the form

$$TE \vdash \texttt{(}e_1\texttt{,}e_2\texttt{)} \Rightarrow (e_1', e_2') \texttt{ at } p : \mu, \varphi \tag{173}$$

and is inferred by rule 26 on the premises

$$TE \vdash e_1 \Rightarrow e_1' : \mu_1, \varphi_1 \tag{174}$$

$$TE \vdash e_2 \Rightarrow e_2' : \mu_2, \varphi_2 \tag{175}$$

$$\mu = (\mu_1 * \mu_2, p) \tag{176}$$

$$\varphi = \varphi_1 \cup \varphi_2 \cup \{\mathbf{put}(p)\} \tag{177}$$

Moreover, (42) was inferred by rule 7 on the premises

$$E \vdash e_1 \to v_1 \tag{178}$$

$$E \vdash e_2 \to v_2 \tag{179}$$

$$v = (v_1, v_2) \tag{180}$$

Let $\varphi_1' = \varphi_2 \cup \{\mathbf{put}(p)\} \cup \varphi'$, i.e. the effect that remains after e_1' has been evaluated. Notice that $\varphi_1 \cup \varphi_1' = \varphi \cup \varphi'$, so by (41), (43) and (44) we have

$$\text{Consistent}(R, TE, E, s, VE) \text{ w.r.t. } \varphi_1 \cup \varphi_1' \tag{181}$$

$$R \text{ connects } \varphi_1 \cup \varphi_1' \text{ to } s \tag{182}$$

$$R' \text{ and } R \text{ agree on } \varphi_1 \cup \varphi_1' \tag{183}$$

By induction on (174), (181), (178), (182) and (183) there exist v_1' and s_1 such that

$$s, VE \vdash R'(e_1') \to v_1', s_1 \tag{184}$$

$$\text{Consistent}(R, \mu_1, v_1, s_1, v_1') \text{ w.r.t. } \varphi_1' \tag{185}$$

Let $\varphi_2' = \{\mathbf{put}(p)\} \cup \varphi'$, i.e. the effect that remains after e_2' has been evaluated. Since $\varphi_2 \cup \varphi_2' \subseteq \varphi \cup \varphi'$ and $s \sqsubseteq s_1$ we get by Lemma 4.1 on (41) that

$$\text{Consistent}(R, TE, E, s_1, VE) \text{ w.r.t. } \varphi_2 \cup \varphi_2' \tag{186}$$

and by (43) and (44) we also get

$$R \text{ connects } \varphi_2 \cup \varphi_2' \text{ to } s_1 \tag{187}$$

$$R' \text{ and } R \text{ agree on } \varphi_2 \cup \varphi_2' \tag{188}$$

By induction on (175), (186), (179), (187) and (188) there exist s_2 and v_2' such that

$$s_1, VE \vdash R'(e_2') \to v_2', s_2 \tag{189}$$

$$\text{Consistent}(R, \mu_2, v_2, s_2, v_2') \text{ w.r.t. } \varphi_2' \tag{190}$$

Since R connects φ to s we have $R(p) \in \text{Dom}(s)$. Thus $R(p) \in \text{Dom}(s_2)$. Let $r = R(p)$; let o be an offset not in $\text{Dom}(s_2(r))$, let $v' = (r, o)$ and let $s' = s_2 + \{v' \mapsto (v_1', v_2')\}$. By rule 16 on (184) and (189) we have

$$s, VE \vdash R'\texttt{(}e_1'\texttt{,}e_2'\texttt{)} \to v', s'$$

as required. Also, by Lemma 4.1 on (185) and (190) we get $\text{Consistent}(R, \mu_i, v_i, s', v_i')$w.r.t. φ', for $i = 1, 2$. Thus, according to the definition of Consistent, we have the desired $\text{Consistent}(R, \mu, v, s', v')$ w.r.t. φ'.

Projection, rules 8 and 9 We treat only `fst` (i.e. rule 8); the case for `snd` is similar. In this case (40) takes the form

$$TE \vdash \texttt{fst}\ e_0 \Rightarrow \texttt{fst}\ e_0' : \mu, \varphi \tag{191}$$

and is inferred by rule 27 on the premises

$$TE \vdash e_0 \Rightarrow e_0' : (\mu * \mu_2, p), \varphi_1 \tag{192}$$

$$\varphi = \varphi_1 \cup \{\mathbf{get}(p)\} \tag{193}$$

Moreover, (42) is inferred by rule 8 on the premise

$$E \vdash e_0 \to (v, v_2) \tag{194}$$

Let $\varphi_1' = \{\mathbf{get}(p)\} \cup \varphi'$, i.e. the effect that remains after the evaluation of e_0'. Notice that $\varphi \cup \varphi' = \varphi_1 \cup \varphi_1'$ so by (41), (43) and (44) we have

$$\text{Consistent}(R, TE, E, s, VE) \text{ w.r.t. } \varphi_1 \cup \varphi_1' \tag{195}$$

$$R \text{ connects } \varphi_1 \cup \varphi_1' \text{ to } s \tag{196}$$

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

By induction on (192), (195), (194), (196) and (197) there exist s' and u' such that

$$s, VE \vdash R'(e_0') \to u', s'$$

$$\text{Consistent}(R, (\mu * \mu_2, p), (v, v_2), s', u') \text{ w.r.t. } \varphi_1' \tag{198}$$

Note that $\mathbf{get}(p) \in \varphi_1'$. Thus by (198) we have $u' \in \text{Dom}(s')$ and r of $u' = R(p)$ and $s'(u') = (v', v_2')$ for some v' and v_2' and $\text{Consistent}(R, \mu, v', s', v')$ w.r.t. φ_1'. In particular,

$$s, VE \vdash R'(\texttt{fst}\ e_0') \to v', s'$$

$$\text{Consistent}(R, \mu, v, s', v') \text{ w.r.t. } \varphi'$$

as desired. This concludes the proof of Theorem 4.1.

5 Algorithms

A closed expression can be translated using our inference rules if and only if it is typable according to Milner's type discipline. Although all well-typed programs can be translated, one naturally wonders whether there is always a "best", or *principal*, translation. Intuitively speaking, we want a principal target expression to be one in which regions are kept as distinct and local as possible. We can make this more precise by introducing a partial order on target expressions. Let C be the one-hole contexts defined by

$$\begin{array}{rcl} C & ::= & [\,] \mid C\ e_2 \mid e_1\ C \mid \texttt{let}\ x\ \texttt{=}\ C\ \texttt{in}\ e_2 \mid \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ C \\ & \mid & \texttt{letrec}\ f\texttt{[}\vec{\rho}\texttt{]}\ \texttt{=}\ e_1\ \texttt{in}\ C \mid \texttt{letrec}\ f\texttt{[}\vec{\rho}\texttt{]}\ \texttt{=}\ C\ \texttt{in}\ e_2 \mid \lambda x.C \\ & \mid & \texttt{letregion}\ \rho\ \texttt{in}\ C \end{array}$$

Then define the partial order $\preceq$ to be the smallest transitive relation satisfying

$$\forall C, e, \rho, \quad C[\texttt{letregion}\ \rho\ \texttt{in}\ e] \preceq \texttt{letregion}\ \rho\ \texttt{in}\ C[e] \tag{199}$$

$$\forall C, e, e', e \preceq e' \Rightarrow C[e] \preceq C[e'] \tag{200}$$

where ρ in (199) must be a region variable such that the occurrence of ρ in $C[\rho]$ is free, to avoid capture. For closed expressions e, we can now define principality as follows: e' is a *principal* translation of e if

1. $\vdash e \Rightarrow e' : \mu', \varphi'$, for some μ' and φ'
2. Whenever $\vdash e \Rightarrow e'' : \mu'', \varphi''$ and there exists a substitution S such that $S(e'') = S(e')$ then there exists a substitution S' such that $S'(e') = e''$;
3. Whenever $\vdash e \Rightarrow e'' : \mu'', \varphi''$ and $e'' \preceq e'$ then $e' \preceq e''$.

The above concept of principality does not in general relate expressions that differ in the type schemes they associate with `letrec`-bound variables. The inference rules allow one to use no region polymorphism at all; conversely, they put no upper bound on how many bound region variables, a type scheme can contain. In practice, there is a trade-off between the locality of regions and the number of region variables that are passed around as parameters at runtime. Here our strategy is to get as much region polymorphism as possible, but to bound the number of region parameters of every region-polymorphic function by the size of its principal type in the Milner system, as detailed below.

The theoretical properties of the algorithm (soundness and completeness with respect to the inference system) are not settled at this point (see page 5.2). Our practical experience with the use of the algorithm can be summarised as follows: in the vast majority of cases, the algorithm quantifies all the region variables in the type schemes of region-polymorphic expressions that can be quantified. (In those few cases where the algorithm does not, our implementation of the algorithm prints a warning and proceeds to infer a slightly less general type scheme.)

We have not found the problem of finding principal translations that quantify as many region variables as possible particularly easy. We shall attempt an analysis of where the difficulties lie in Section 5.1.

For brevity, pairs and projections are omitted from the source language and the semantic objects in this chapter; we also drop the distinction between places and region variables.

5.1 The Inference Problem

There seems to be two related sources of complications. The first is the presence of effects, which are sets, in types. The second is the fact that we allow a form of polymorphic recursion.

5.1.1 The presence of effects in types

Like several other effect systems[13,30], our's allow types of the form $\tau \xrightarrow{\varphi} \tau'$, where the effect φ is a set. The presence of sets in types means that one has to reconsider unification as a basic tool for type inference. Substitution-based type inference algorithms, the best known of which is perhaps Milner's algorithm W, work on the following principle. Let e be an expression and let TE be a type environment giving the types of the free variables of e. Then $W(TE, e)$ attempts to find not just a type τ for e, but also a substitution S, such that $S(TE) \vdash e : \tau$. Informally speaking, the substitution pinpoints how the types in the type environment must be "refined" in order to make e typable. In effect systems an important form of "type refinement" is that of increasing an effect (under the ordering of set inclusion). For example, consider the expression

$$\lambda f.\ \texttt{if}\ \ldots\ \texttt{then}\ \lambda x.f(x)\ \texttt{else}\ (\lambda x.x + 1)$$

After the inference of the type for the `then` branch, the type environment might contain the binding: $\{f : ((\alpha, \rho_1) \xrightarrow{\emptyset} (\beta, \rho_2), \rho_3)\}$. Next, the type of $(\lambda x.x + 1)$ might be inferred to be $((\texttt{int}, \rho_1') \xrightarrow{\{\mathbf{get}(\rho_1'), \mathbf{put}(\rho_2')\}} (\texttt{int}, \rho_2'), \rho_3'))$. We want the unification of these types to refine the type of f to have the effect $\{\mathbf{get}(\rho_1'), \mathbf{put}(\rho_2')\}$ on the function arrow. Talpin and Jouvelot[30] introduce *effect variables* to achieve this. In their algorithm, one always has just an effect variable on every function arrow. In addition, their algorithm infers a set of constraints of the form $\epsilon \supseteq \varphi$. Their algorithm then alternates between solving constraint sets and inferring types. Our introduction of *arrow effects* is a variation of their scheme which allows substitution to do all "refinement" without the need for constraint sets. In the present system, the type of the `then`- branch above is $\{f : ((\alpha, \rho_1) \xrightarrow{\epsilon_1.\emptyset} (\beta, \rho_2), \rho_3)\}$ and the type of the `else`-branch is $((\texttt{int}, \rho_1') \xrightarrow{\epsilon_2.\{\mathbf{get}(\rho_1'), \mathbf{put}(\rho_2')\}} (\texttt{int}, \rho_2'), \rho_3'))$. Unification then gives the obvious substitution on region and type variables, but it also produces the *effect substitution*

$$S_e = \{\epsilon_1 \mapsto \epsilon_1.\{\mathbf{get}(\rho_1'), \mathbf{put}(\rho_2')\}, \epsilon_2 \mapsto \epsilon_1.\{\mathbf{get}(\rho_1'), \mathbf{put}(\rho_2')\}\}$$

Thus the resulting type of f in the type environment is

$$((\mathtt{int}, \rho_1') \xrightarrow{\epsilon_1.\{\mathbf{get}(\rho_1'),\mathbf{put}(\rho_2')\}} (\mathtt{int}, \rho_2'), \rho_3')$$

Notice that substitution alone is now sufficient to achieve the desired refinement of the type of f.

With the use of arrow effects, it is possible to write a unification algorithm which finds most general unifiers of types. The most interesting case is:

$$\begin{array}{l}
\mathrm{Unify}(\mu_l, \mu_r) = \mathrm{case}\ (\mu_l, \mu_r)\ \mathrm{of} \\
\quad ((\mu_l' \xrightarrow{\epsilon_l.\varphi_l} \mu_l'', \rho_l), (\mu_r' \xrightarrow{\epsilon_r.\varphi_r} \mu_r'', \rho_r)) \Rightarrow \\
\qquad \mathrm{let}\ S = \{\epsilon_l \mapsto \epsilon_r.\varphi_l \cup \varphi_r, \epsilon_r \mapsto \epsilon_r.\varphi_l \cup \varphi_r\} \\
\qquad\quad S' = \mathrm{Unify}(S\mu_l', S\mu_r') \circ S \\
\qquad\quad S'' = \mathrm{Unify}(S'\mu_l'', S'\mu_r') \circ S' \\
\qquad\quad S''' = \mathrm{Unify}(S''(\rho_l, \rho_r)) \circ S'' \\
\qquad \mathrm{in}\ S''' \\
\ldots \Rightarrow \ldots
\end{array}$$

There are two kinds of occurrences of region and effect variables that behave very differently. Let us define the *principal* (free) region and effect variables of types as follows:

$$\begin{array}{rlrl}
\mathrm{pfev}(\mathtt{int}) & = \emptyset & \mathrm{pfrv}(\mathtt{int}) & = \emptyset \\
\mathrm{pfev}(\alpha) & = \emptyset & \mathrm{pfrv}(\alpha) & = \emptyset \\
\mathrm{pfev}(\mu \xrightarrow{\epsilon.\varphi} \mu') & = \mathrm{pfev}(\mu) \cup \mathrm{pfev}\,\mu' \cup \{\epsilon\} & \mathrm{pfrv}(\mu \xrightarrow{\epsilon.\varphi} \mu') & = \mathrm{pfrv}(\mu) \cup \mathrm{pfrv}\,\mu' \\
\mathrm{pfev}(\tau, \rho) & = \mathrm{pfev}(\tau) & \mathrm{pfrv}(\tau, \rho) & = \mathrm{pfrv}(\tau) \cup \{\rho\}
\end{array}$$

Unification of types never forces the identification of variables that do not occur in principal positions. In particular, effect unification alone never forces identification of regions. Variables that occur in non-principal positions only are "stale" — they can never be instantiated further by unification. When a function type is formed for an expression of the form $\lambda x.e$, one can avoid introducing new stale variables by using the observe rule (i.e., rule 29, page 13). But stale variables are a problem in connection with polymorphic region recursion. There are cases where one can obtain a more general type scheme for a recursive function by mapping stale variables to other variables, but our algorithm does not attempt to do so, as it is not always clear which non-stale variables it should choose. Instead, stale variables are left free (at that point) and a warning is printed.

5.1.2 Polymorphic recursion

In Standard ML[22], recursive functions cannot be used polymorphically within their own declaration. At first sight, our translation rules resemble the Milner/Mycroft calculus[24], in which recursive functions can be used polymorphically within their own body. The type-checking problem for the Milner/Mycroft is equivalent to the semi-unification problem[12,

1], and semi-unification is unfortunately undecidable[16]. That is why we do not allow f to be used type-polymorphically within its own body. Still, because of arrow effects in types, the type checking problem with polymorphic region recursion is not straightforward.

One approach, which we have tried, is to use Henglein's idea of expressing the type-checking problem as a semi-unification problem[12]. Let us break the region and effect inference problem down into three smaller tasks. The first task is to infer which region annotations of the form "`at` ρ" in the translation should be equal and to find out which region variables can be quantified in the type schemes for `letrec` bound variables. The second problem is to infer effects. The third problem is to insert `letregion` expressions, based on the effect inference.

Assume that none of these tasks depends on its successor. Then the first task can be accomplished very efficiently with the aid of semi-unification. Although semi-unification itself is undecidable, the equations and inequalities that arise in our special case using Henglein's reduction[12] are always between variables and hence easy to solve.

Unfortunately, region and effect inference cannot be separated, as there are cases where the effect of a function, f, say must be present in the type environment in order to prevent quantification of a region variable in the type scheme of a `letrec`-bound function, g, say, which occurs in the scope of the binding of f.

A different approach, originating from Mycrofts seminal paper on polymorphic recursion[24], is to iterate the type checking of the declaration of a function, starting from a very general type scheme. The obvious question is whether this process is guaranteed to terminate.

We now present a lemma which suggests how one can achieve termination for our inference system. First let us say that a type scheme is *simple* if it contains quantification of region variables and effect variables only. Furthermore, let us say that a type scheme $\forall\vec{\alpha}\vec{\rho}\vec{\epsilon}.\tau$ is *well-formed* if (a) every bound region variable occurs in some principal position in τ, (b) every bound effect variable occurs in some principal position in τ and (c), in every arrow effect $\epsilon.\varphi$ occurring in τ, if ϵ is not bound, then no variable in φ is bound either. We say that a type scheme $\sigma' = \forall\vec{\alpha}\vec{\rho}\vec{\epsilon}.\tau'$ is an *instance* of a type scheme σ, written $\sigma \geq \sigma'$, if $\sigma \geq \tau'$ and $\mathrm{fev}(\sigma) \subseteq \mathrm{fev}(\sigma')$. This relation is a pre-order; on the set of well-formed type schemes it is even a partial order, given that we identify type schemes that are equal up to renaming of bound variables. Thus we get an asymmetric relation $>$ on well-formed type schemes defined by: $\sigma > \sigma'$ if $\sigma \geq \sigma'$ and $\sigma' \not\geq \sigma$.

Lemma 5.1 *Let A be a finite set of variables. There exists no infinite decreasing sequence $\sigma_0 > \sigma_1 > \sigma_2 \cdots$ of type schemes satisfying that, for all $i \geq 0$, σ_i is well-formed, simple and $\mathrm{fv}(\sigma_i) \subseteq A$.*

Proof If $\sigma \geq \sigma'$ and both are simple and well-formed, then the number of bound variables of σ' is at most the number of bound variables of σ. Moreover, $\mathrm{fv}(\sigma) \subseteq \mathrm{fv}(\sigma')$. Indeed, if a variable occurs free in σ (be it in a principal occurrence or not) then it occurs free at the corresponding position in σ'. Thus, any decreasing sequence starting from σ_0 can have length at most $a \times p \times b$, where $a = |A|$, p is the number of occurrences of type constructors in σ_0 and b is the number of distinct bound variables in σ_0.

This lemma underlies our algorithm. To make sure that we stay within a fixed number of free region and effect variables, we separate the generation of fresh region and effect variables from the inference algorithm proper. This first phase we call *spreading*; it consists of a single pass over the type-annotated source expression during which all occurrences of region and effect variables are replaced by fresh variables.

The second phase then performs region and effect inference by a recursive descent analysis of the spread expression. In the case of recursion functions, we also need to be able to match two type schemes. To this end, let us define that a substitution is *simple*, if it instantiates region variables and effect variables only.

Lemma 5.2 *There exists and algorithm,* Match*, with the following properties. Let* σ_1 *and* σ_2 *be simple, well-formed type schemes. If there exists a simple substitution* S *such that* $S(\sigma_1) \geq S(\sigma_2)$ *then* $\mathrm{Match}(\sigma_1, \sigma_2)$ *succeeds with result* S^**, which satisfies*

1. $S^*(\sigma_1) \geq S^*(\sigma_2)$
2. *For all simple* S*, if* $S(\sigma_1) \geq S(\sigma_2)$ *then there exists a simple substitution* S' *such that* $S = S' \circ S^*$.

Otherwise, $\mathrm{Match}(\sigma_1, \sigma_2)$ *fails.*

In Section 3.1, we defined $\mathrm{RegTyEffGen}(TE, \varphi)(\tau)$ to mean $\forall\vec{\alpha}\forall\vec{\rho}\forall\vec{\epsilon}.\tau$ where the bound variables were those that were free in τ but not in TE or in φ. In order to ensure that this operation only produces well-formed type schemes, we now forgo some of the quantification and redefine $\mathrm{RegTyEffGen}(TE, \varphi)(\tau)$ to mean $\forall\alpha_1 \cdots \alpha_n \forall\rho_1 \cdots \rho_k \forall\epsilon_1, \ldots \epsilon_m.\tau$, where $\{\alpha_1, \ldots, \alpha_n\} = \mathrm{ftv}(\tau) \setminus \mathrm{ftv}(TE)$, $\{\rho_1, \ldots, \rho_k\} = \mathrm{pfrv}(\tau) \setminus \mathrm{frv}(TE, \varphi)$ and $\{\epsilon_n, \ldots, \epsilon_m\} = \mathrm{pfev}(\tau) \setminus \mathrm{fev}(TE, \varphi)$.

The operation TyEffGen is a special case of the above in which quantification of region variables is omitted.

5.2 The Region and Effect Inference Algorithm

After expressions have type type-checked and their regions and effects have been spread, they conform to the following grammar:

$$
\begin{array}{rcl}
re & ::= & \mathtt{x}[\vec{\tau};\vec{\epsilon.\varphi}] : \mu \\
& | & \mathtt{f}[\vec{\tau};\vec{\rho};\vec{\epsilon.\varphi}]\ \mathtt{at}\ \rho\ :\ \ \mu \\
& | & (\lambda x : \mu.re\ \mathtt{at}\ \rho) : \mu \\
& | & re(re') \\
& | & \mathtt{let}\ x : (\forall\vec{\alpha}\,\forall\vec{\epsilon}.\tau,\ \ \rho)\ \mathtt{=}\ re\ \mathtt{in}\ re' \\
& | & \mathtt{letrec}\ f : (\forall\vec{\alpha}\,\forall\vec{\rho}\forall\vec{\epsilon}.\tau,\ \ \rho)\ \mathtt{=}\ re\ \mathtt{in}\ re' \\
& & \qquad\qquad (\text{where } re \text{ must start with a } \lambda)
\end{array}
$$

In the type scheme associated with x in the `let`-expression and the type scheme associated with f in the `letrec`-expression, there are fresh region and effect variables

everywhere, and they are all quantified in a particular order. Correspondingly, the applied occurrences have been decorated with fresh instances (matching the quantified variables in number and order). As far as type instances are concerned, these type instances are obtained from the type instances that were found during type inference by replacing all their occurrences of a region or effect variable by fresh variables. The number and order of quantified type variables was decided during type checking and is not affected by the subsequent phases.

Notice that the applied occurrences of program variables have been decorated with spread types (one type for each quantified type variable), fresh regions (one for each quantified region variable) and fresh, empty arrow effects (one for each quantified effect variable). The region and effect inference algorithm has the type indicated by

$$\boxed{\mathcal{RE}(TE, re) = (S, re', \mu, \varphi)}$$

Notice that it transforms an annotated expression to an expression of the same kind. For constructing `letregion` expressions, the inference algorithm uses the following function:

$$\begin{array}{l}\text{Retract}(S, TE, re, \mu, \varphi) \Rightarrow \\ \quad \text{let } \varphi' = \text{Observe}(S(TE), \mu)(\varphi) \\ \qquad \vec{\rho} = \text{frv}(\rho) \setminus \text{frv}(\varphi') \\ \quad \text{in } (S, \texttt{letregion } \vec{\rho} \texttt{ in } re, \mu, \varphi')\end{array}$$

The inference algorithm occurs below. The main action takes place in the case for `letrec` expressions. Initially, σ_a, (a for "assumed") is the fully spread type scheme which has *all* of its region and effect variables quantified. Starting from this optimistic assumption, we call $\mathcal{RE}^{rec}$ (also shown below). $\mathcal{RE}^{rec}$ infers a type for the body and then compares the inferred type scheme for f with the assumed type scheme. If they are equal (up to renaming of bound variables) then a usable type scheme has been found. Otherwise, the body of f is checked again, under the less general assumption about f. This continues till a fixed point is reached.

$$\begin{array}{l}\mathcal{RE}(TE, x[\vec{\tau}; \epsilon.\vec{\varphi}] : \mu_a) = \\ \quad \text{let } (\forall\vec{\alpha_b}\forall\vec{\epsilon_b}.\tau_x, \rho_x) = TE(x) \\ \qquad I = \{\vec{\alpha_b} \mapsto \vec{\tau}; \vec{\epsilon_b} \mapsto \epsilon.\vec{\varphi}\} \\ \qquad S = \text{Unify}((I(\tau_x), \rho_x), \mu_a) \\ \quad \text{in Retract}(S, TE, S(x[\vec{\tau}; \epsilon.\vec{\varphi}] : \mu_a), \quad S(\mu_a), \emptyset, \{\})\end{array}$$

$$\begin{array}{l}\mathcal{RE}(TE, f[\vec{\tau}; \vec{\rho}; \epsilon.\vec{\varphi}] \texttt{ at } \rho : \mu_a \text{ as } (\tau_a, \rho_a)) = \\ \quad \text{let } (\forall\vec{\alpha_b}\forall\vec{\rho_b}\forall\vec{\epsilon_b}.\tau_f, \quad \rho_f) = TE(f) \\ \qquad I = \{\vec{\alpha_b} \mapsto \vec{\tau}; \vec{\rho_b} \mapsto \vec{\rho}; \vec{\epsilon} \mapsto \epsilon.\vec{\varphi}\} \\ \qquad \mu = (I(\tau_f), \rho) \\ \qquad S = \text{Unify}(\mu, \mu_a) \\ \qquad \varphi = S(\{\mathbf{get}(\rho_f), \mathbf{put}(\rho)\}) \\ \quad \text{in } \text{Retract}(S, TE, S(f[\vec{\tau}; \vec{\rho}; \epsilon.\vec{\varphi}] \texttt{ at } \rho_a : \mu_a), \quad S(\mu_a), \varphi)\end{array}$$

$\mathcal{RE}(TE, (\lambda\ x : \mu_x.re_1\ \texttt{at}\ \_\ ) : \mu_\lambda) =$
  let $(S_1, re'_1, \mu_1, \varphi_1) = \mathcal{RE}(TE + \{x \mapsto \mu_x\}, re_1)$
    $S_2 = \text{Unify}(S_1(\tau_\lambda), S_1(\mu_x) \xrightarrow{\epsilon.\varphi_1} \mu_1)$, where $(\tau_\lambda, \rho_\lambda) = \mu_\lambda$ and ϵ is fresh
    μ' as $(\mu'_x \xrightarrow{\epsilon'.\varphi'} \mu'_1, \rho') = S_2(S_1(\mu_\lambda))$
  in $\text{Retract}(S_2 \circ S_1, TE, (\lambda\ x : \mu'_x.S_2(re'_1)\ \texttt{at}\ \rho') : \mu',\ \ \mu', \{\mathbf{put}(\rho')\})$

$\mathcal{RE}(TE, re_1(re_2)) =$
  let $(S_1, re'_1, \mu_1, \varphi_1) = \mathcal{RE}(TE, re_1)$
    $(S_2, re'_2, \mu_2, \varphi_2) = \mathcal{RE}(S_1(TE), S_1(re_2))$
    $(\mu'_2 \xrightarrow{\epsilon'.\varphi'} \mu, \rho) = S_2\mu_1$
    $S_3 = \text{Unify}(\mu'_2, \mu_2)$
    $\varphi = S_3(S_2(\varphi_1) \cup \varphi_2 \cup \varphi' \cup \{\epsilon'\} \cup \{\mathbf{get}(\rho)\})$
  in $\text{Retract}(S_3 \circ S_2 \circ S_1, TE, S_3((S_2(re'_1))(re'_2)),\ \ S_3(\mu), \varphi)$

$\mathcal{RE}(TE, \texttt{let}\ x : (\sigma_x, \rho_x) = re_1\ \texttt{in}\ re_2) =$
  let $(S_1, re'_1, \mu_1 \text{ as } (\tau_1, \rho_1), \varphi_1) = \mathcal{RE}(TE, re_1)$
    $\sigma = \text{TyEffGen}(S_1(TE), \varphi_1)(\tau_1)$
    $(S_2, F, (\sigma_a, \rho_a), (\sigma_i, \rho_i)) = \text{Match}(x, S_1(\sigma_x, \rho_x), (\sigma, \rho_1))$
    $(S_3, re'_2, \mu_2, \varphi_2) = \mathcal{RE}(S_2(S_1(TE)) + \{x \mapsto (\sigma_i, \rho_i)\}, S_2(S_1(F(re_2))))$
    $\varphi = S_3(S_2(\varphi_1)) \cup \varphi_2$ and $S = S_3 \circ S_2 \circ S_1$
  in $\text{Retract}(S, TE, \texttt{let}\ x : S_3(\sigma_i, \rho_i)\ \texttt{=}\ S_3(S_2(re'_1))\ \texttt{in}\ re'_2,\ \ \mu_2, \varphi)$

$\mathcal{RE}(TE, \texttt{letregion}\ \vec{\rho}\ \texttt{in}\ re) = \mathcal{RE}(TE, re)$

$\mathcal{RE}(TE, \texttt{letrec}\ f : (\sigma_a, \rho_a) = re_1\ \texttt{in}\ re_2) =$
  let $(S_1, re'_1, (\sigma_1, \rho_1), \varphi_1, F_1) = \mathcal{RE}^{rec}(TE, f, (\sigma_a, \rho_a), re_1, \text{Id}, \lambda e.e)$
    $(S_2, re'_2, \mu_2, \varphi_2) = E(S_1(TE) + \{f \mapsto (\sigma_1, \rho_1)\}, S_1(F_1(re_2)))$
    $\varphi = S_2(\varphi_1) \cup \varphi_2$
  in $\text{Retract}(S_2 \circ S_1, \texttt{letrec}\ f : S_2(\sigma_1, \rho_1)\ \texttt{=}\ S_2(re'_1)\ \texttt{in}\ re'_2,\ \ \mu_2, \varphi)$

$\mathcal{RE}^{rec}(TE, f, (\sigma, \rho), re, S, F) =$
  let $\forall\vec{\alpha_a}\vec{\rho_a}\vec{\epsilon_a}.\tau = \sigma$ where the bound variables are fresh
    $(S_1, re', \mu_1 \text{ as } (\tau_1, \rho_1), \varphi_1) = \mathcal{RE}(TE + \{f \mapsto (\sigma, \rho)\}, re)$
    $\sigma_a = \forall\vec{\alpha_a}\vec{\rho_a}\vec{\epsilon_a}.S_1(\tau)$ - assumed type scheme
    $\sigma_i = \text{TyRegEffGen}(S_1(TE + \{f \mapsto (\sigma, \rho)\}), \varphi_1)(\tau_1)$ - inferred type scheme
    $(S_2, F_2, (\sigma'_a, \rho'_a), (\sigma'_i, \rho'_i)) = \text{Match}(f, (\sigma_a, S_1(\rho)), (\sigma_i, \rho_1))$
  in if $\sigma'_a \overset{\alpha}{\equiv} \sigma'_i$ then $(S_2 \circ S_1 \circ S, S_2(re'), (\sigma'_i, \rho'_i), S_2(\varphi_1), F_2 \circ F)$
    else $\mathcal{RE}^{rec}(S_2(S_1(TE)), f, (\sigma''_i, \rho''_i), S_2(F_2(re')), S_2 \circ S_1 \circ S, F_2 \circ F)$

The Match employed in the algorithm is a slight variant of the Match referred to in the previous section. The inferred type scheme does not always have as many bound region and effect variables as the assumed type scheme. In every expression of the form $f[\vec{\tau}, \vec{\rho}, \epsilon.\vec{\varphi}]$, the number of region and effect instances (i.e., the lengths of $\vec{\rho}$ and $\epsilon.\vec{\varphi}$) must match the number of quantified region and effect variables in the type scheme for f. Therefore $\text{Match}(f, \ldots)$ also returns an *expression transformer*, F, which is a map from

expressions to expressions; $F(re)$ produces the expression obtained from re by deleting instances of region and effect variables that are no longer quantified from free applied occurrences of f in re.

Finally, we write $\mathrm{Match}(f, (\sigma_a, \rho_a), (\sigma_i, \rho_i))$ as a shorthand for

$$\begin{array}{ll} \texttt{let} & (S, F, \sigma'_a, \sigma'_i) \texttt{ = } \mathrm{Match}(f, \sigma_a, \sigma_i) \\ & S' = \{\rho_a \mapsto \rho_i\} \\ \texttt{in} & (S' \circ S, F, S'(\sigma'_a, \rho'_a), S'(\sigma'_i, \rho'_i)) \end{array}$$

At this point, the reader probably expects a theorem to the effect that the algorithm is sound with respect to the inference system and always terminates. Alas, we have to confess that, strictly speaking, the algorithm is not sound with respect to the inference rules, because the algorithm foregoes some quantification (of non-principal occurrences of variables) even though these *must* be quantified, according to the inference rules. In retrospect, this appears to be a problem with the way we have stated the inference rules and we should probably reformulate the inference rules (and the soundness proof in Section 4) so that the rules just *allow* quantification, before attempting a proof of the correctness of the algorithm. We leave this as a future project.

6 Strengths and weaknesses of region inference

In this section we summarise what we currently know about the strengths and weaknesses of region inference, considered as a compilation technique.

In Section 6.1 we discuss strengths and weaknesses that are apparent from the formal semantics and the existing literature on compiler technology. In Section 6.2 we present the results of experiments on a prototype system. Finally, in Section 6.3 we comment on the possibility of combining region allocation and garbage collection.

6.1 On the accuracy of the region inference rules

A region-annotated program can exhaust memory by allocating a large number of regions, each containing a large number of values. If the ratio of live data to dead data is high, the program is simply too demanding for the memory available. But if the ratio is low, we have a failure of region inference; such a failure can be understood as the combination of two extreme ways in which a region-annotated program can "run wild": it can continue to extend a region with new values even though most of the values in the region are dead; or it can create an excessive number of regions. We refer to these kinds of space leaks as *vertical* and *horizontal* overflow, respectively (even though an implementation need not have a fixed upper limit on the number of regions it allows, or the number of values that can be in a region).

6.1.1 Vertical overflow

Vertical overflow happens as a result of region inference not being good enough at discriminating lifetimes. The fact that region-polymorphic functions can be used polymorphically within their own declaration normally achieves that the values belonging to different invocations of recursive functions are given different lifetimes. But there is one important exception: tail-recursive functions with accumulating parameters. For example, consider the iterative version of the factorial function:

```
letrec itfac(p) =
       let n = snd p
       in  let acc = fst p
           in if n=0 then p else itfac (n*acc,n-1 )
           end
       end
in fst (itfac (1 ,10 ))
end
```

Because the `then` branch and the `else` branch must have the same type and because the *itfac* may return its argument unchanged (when n is 0), every recursive call of *itfac* puts its argument pair in the same region. In other words, the region polymorphism is not exploited. Thus, the evaluation of *itfac*`(1,` n`)` uses space which is proportional to n.

The standard cure of this common ailment in functional programming is *tail recursion optimisation*. If the return value of a function, f, is obtained by calling a function g (which may be f itself), then this call is said to be a *tail call*. In implementations that use a stack of activation records, one can optimise a tail call by overwriting the activation record for f with the activation record for g, just before transferring control to g. In the above example, this optimisation will reuse the same stack positions for all the recursive calls, so only a small, constant amount of space is needed.

A similar form of optimisation is possible in connection with region inference. The idea is to allow different *modes* of storing values into regions. The mode described so far is to store the value at the top of the region, increasing the size of the region by 1. An alternative mode is to delete all existing values in the region before storing the value; in this case we say that the value is stored at the *bottom* of the region. After region inference is complete, our implementation performs a program analysis which transforms every annotation `at` ρ to either `attop` ρ or `atbot` ρ.[7] The analysis is similar to existing analyses for globalisation and single-threadedness[29,11,28]. The analysis is helped by the fact that all regions are explicitly scoped and that all variables have types that tell what regions they need. A detailed description of the analysis is beyond the scope of this report.

In the above example, the analysis discovers that one *never* needs to use `attop`. It turns out that no more than 6 regions, each one containing at most one value, are ever

[7] Actually, there is a third possibility, called `somewhereat` ρ, which is used when ρ is bound by a `letrec`. Thus, not only region variables, but also modes of storage are passed to region-polymorphic functions at runtime.

needed, no matter how large the input to *itfac* is. We feel this is an acceptable use of memory resources.

6.1.2 Horizontal overflow

As noted by Chase[5] and Appel[2, Chap 12], stack allocation can lead to poor utilisation of memory, if the lifetime of the actual argument to a function call is short, compared to the lifetime of the call itself. Paradoxically, one way this can happen is that region inference separates regions that were better considered identical. For example, consider

```
letrec decloop(x) = if x=0 then 1 else decloop(x-1)
in decloop 100
end
```

Here *decloop* is region-polymorphic, both in the argument region and in the result region; region inference will allocate x and $x-1$ in different regions. Thus, roughly $100c$ regions will be active at the same time at one point of the execution, where c is the net growth in the number of regions, per recursive call.[8] This can be handled by ascribing a less general type to *decloop*, for if x and $x-1$ are forced into the same region, the above-mentioned liveness analysis can discover that x can be treated as an updatable variable.

Forgoing region polymorphism in tail calls is useful, as long as the liveness analysis is able to discover that space for arguments can be reused.

6.2 Experimental results

In this section we show the results of executing selected programs on a prototype implementation. The system infers types, regions and effects (in three distinct passes). A fourth pass infers storage modes (see Section 6.1.1). The resulting program (represented as an abstract syntax tree) is then interpreted by an interpreter written in Standard ML. The language accepted by the system extends the one presented in this report by allowing conditionals, arithmetic, and lists.

We emphasise that the target programs, we execute, are precisely as in the formal semantics. Hence all values are boxed. Every call of the form $f(a,b)$ involves the construction of the pair (a,b) in a region; every evaluation of an expression of the form $\lambda x.e$ allocates a closure in a region; every evaluation of a constant expression stores the constant in a region. Apart from the analysis of storage modes (which is essential for obtaining good results) we have resisted the temptation to make obvious improvements in the target programs, such as allocating the values of those regions, that have a know size, on a stack, rather than in the region store. The point is that we are primarily interested in understanding the fundamental memory behaviour of region-annotated programs and this is best done, when the effects of optimisations do not blur the picture.

The quantities measured are:

[8] The precise number of regions that will be active at the same time, without any optimisations, is 505.

(1) **Highest region allocated** , i.e. the maximum number of regions that were in existence at the same time during the execution of the program;

(2) **Total number of region allocations** , i.e. the total number a `letregion` expression that were evaluated during the execution of a program. The ratio between this number and the number of highest region allocated indicates the extent to which regions are reused;

(3) **Total number of value allocations** , i.e. the number of times a value generating expression was evaluated. As we have not performed any optimisations, this number is a property of the region inference rules and the program alone, and it does not depend on the allocation and deallocation of regions;

(4) **Maximum memory size** , i.e. the maximum number of values that were held in store in allocated regions at any one point in time during the execution.

(5) **Final memory size** , i.e. the number of values that were held in the store in allocated regions at the end of the computation.

The final number of regions is not measured; it can be deduced from the type of the expression and is not above 3 in any of the examples.

For our experimental data, we chose small programs that we expected to have interesting memory behaviour. They are shown in the table below.

`fib(15)`	The computation of the 15th Fibonacci number by the "naïve" method (e.g. [15, page 235]). The computation of `fib`(n) requires number of function calls which is exponential in n.
`sum(100)`	Here `sum`(n) computes the sum $\Sigma_{i=1}^{n} i$, by n recursive calls, none of which are tail recursive.
`sumit(100)`	As above, but the sum is computed iteratively, by n tail recursive calls.
`hsumit(100)`	As above, but computed by folding the plus operator over the list $[1, .., 100]$: `foldr (op +) 0 [1,..,100]`;
`acker(3,6)`	Ackermann's function, as defined in [15, page 239], except that our version is not curried.
`ackerMon(3,6)`	As above, but with Ackermann's function made region monomorphic.

`appel1(100)`	A program, which Appel discusses in his chapter on space complexity [2, page 134]:
	`fun s(0) = nil | s(i) = 0 :: s(i-1)`
	`fun f (n,x) =`
	`  let val z = length(x)`
	`  fun g() = f(n-1, s(100))`
	`  in`
	`  if n=0 then 0 else g()`
	`  end`
	`val result = f(100,nil);`

`appel2(100)`	Same as `appel1`, but with `g()` replaced by `g() + 1`. (Discussed by Appel[2, page 135]

`appel3(100)`	A variant of the appel1 obtained by replacing the definition of `f` by:
	`fun f(nx as (n,x)) =`
	`  let val z = length(x)`
	`  in`
	`  if n=0 then 0`
	`  else f(if true then (n-1, s(100))`
	`  else nx)`
	`  end`
	The "`if true` ..." serves to make `f` monomorphic in regions. (`appel3` is not from Appel's book.)

`quick`(n)	Hoare's Quicksort, taken from Paulson's book[25, page 98]. This version of Quicksort uses lists extensively. n is the number of integers being sorted. All measurements include the operations necessary to build the list, using a random number generator based on [25, page 96]

All results are after storage modes (`attop`/`atbot`/ `somewhereat`) have been deter-

mined (see Section 6.1.1). There is no garbage collection, apart from region allocation and de-allocation.

The results are shown in the tables below. The first table concerns those programs, that manipulate integers, booleans, pairs and closures. In all cases, but one, the final memory consists of just one region containing just one word. One observes that there is a very large number of region allocations; indeed, in three of the examples, there is one region allocation for every value stored (compare rows (2) and (3)). Since the maximal number of regions and the maximal memory sizes are small by comparison, these programs operate with very frequent allocation of very short-lived regions, that are used over and over again.

The strength of region polymorphism is illustrated by the differences observed between `acker` and `ackerMon`. The latter, where region polymorphism has been disabled, has a much larger maximal memory size, 86.890, than the former, 2.043.

The results for `sumit` illustrate the advantage of not using region polymorphism in cases, where the function is tail-recursive and the parameters serve the rôle of updatable locations. This situation is detected (in simple cases) by our storage mode analysis. When computing `sumit`(n), the number of the highest region, namely 6, and the maximum memory size, also 6, are both independent of n.

When we compute the sum by folding the plus operator over the list (`hsumit(100)`), all the results of the plus operation are also put in the one region, because the operator is a lambda-bound parameter of the fold operation and hence cannot be region-polymorphic. In this case, however, the analysis of storage modes is not clever enough to discover that `atbot` could be used, so the final memory size becomes 101, of which only one word is live.

	`fib(15)`	`sum(100)`	`sumit(100)`	`hsumit(100)`
(1) maxregion	47	205	6	12
(2) region allocs	15.029	605	406	715
(3) value writes	15.030	606	707	1.214
(4) max memsize	32	104	6	507
(5) final memsize	1	1	1	101

	`acker(3,6)`	`ackerMon(3,6)`
(1) maxregion	3.058	514
(2) region allocs	1.378.366	1.119.767
(3) value writes	1.378.367	1.550.599
(4) max memsize	2.043	86.880
(5) final memsize	1	1

Next, we show the results for those programs that manipulate lists extensively. A list occupies up to three regions: one region for the elements, one region for the constructors (cons and nil) and one region for the pairs, to which cons is applied. Thus, a list with n integers is represented by between $2n + 2$ and $3n + 1$ values, depending on the degree of sharing between the list elements. Since our lists are generated by the iterative application

of a random number generator, we have no sharing between list elements, and hence a list with n elements is represented by $3n + 1$ values.

	`quick(50)`	`quick(500)`	`quick(1000)`	`quick(5000)`
(1) maxregion	170	1.520	3.020	15.020
(2) region allocs	2.729	45.691	86.915	556.369
(3) value writes	3.684	65.266	122.793	795.376
(4) max memsize	603	8.078	10.525	61.909
(5) final memsize	152	1.502	3.002	15.002

We see that, apart from one word, the final results are the only values left after the computation.[9] Also, the maximal number of regions allocated (line 1) is roughly the same as the number of values in the final result. The ratio between the maximal memory size and the final memory size varies between roughly 4.0 and 5.5.

Finally, the results of executing `appel1(100)`, `appel2(100)` and `appel3(100)` are shown below:

	`appel1(100)`	`appel2(100)`	`appel3(100)`
(1) maxregion	911	1.111	311
(2) region allocs	81.714	81.914	81.113
(3) value writes	101.614	101.814	101.413
(4) max memsize	20.709	20.709	411
(5) final memsize	1	1	1

According to Appel, the space complexity of `appel1(`N`)` is $\Theta(N)$, the point being that although a list consisting of N elements is built N times, no two of these lists are ever needed at the same time. Appel mentions tail-call optimisation and liveness analysis as two ways of avoiding the quadratic memory use. In most heap-based implementations, however, tail-call optimisation and liveness analysis only works indirectly, because of garbage collection. In other words, if there is enough memory, `appel1(`N`)` will allocate $\Theta(N^2)$ memory cells; in fact, it would only use $\Theta(N)$ words, if one instigates a garbage collection upon each iteration of the function. Similar observations apply to `appel2(`N`)`.

We see that the region-based implementation of `appel1` and `appel2` uses $\Theta(N^2)$ space (line 4), although at the end of the computation, just one word remains allocated. However, in `appel3(100)`, where we have "in-lined" `g` and forced all the actual arguments to the different recursive invocations of `f` to be in the same regions, it becomes obvious (even to our simple storage mode analyser) that the lists can overwrite each other. Thus only $\Theta(N)$ space is used.

In conclusion, region-based implementation favours functions f that satisfy:

1. If f implements an iterative computation, then f is written tail-recursively with accumulating parameters and preferably without region-polymorphism in the recursive calls.

2. Otherwise, f is not terminated by a tail call.

[9] The one additional value stems from the last iteration of the random number generator.

6.3 Garbage collection

The interpreter used in the experiments above had no garbage collector. As we saw in Example 1, dangling pointers can arise through closures. Consequently, copying collection would be non-trivial. But suppose we add the following side-condition to rule 22:

$$\forall y \in \mathrm{FV}(\lambda x.e).\, \mathrm{frv}(TE(y)) \subseteq \mathrm{frv}(\varphi')$$

We conjecture that this single change is sufficient to ensure that no dangling references arise. Adding this side-condition simply restricts the translations that are possible; since our safety theorem in Section 4 holds for all statements $TE \vdash e \Rightarrow e' : \mu, \varphi$ in the original system, it will also hold for those that can be inferred in the restricted system. However, the accuracy of the analysis clearly suffers as a result of the modification. Since the general spirit of this exploration is to push stack allocation to its limits, we decided to stay with the more accurate analysis.

7 Conclusion and future work

We have presented an inference system which specifies a translation from the call-by-value λ-calculus with polymorphic `let` into a language, in which all memory allocation and deallocation is explicit. A central feature of the scheme is that it works for higher-order functions. Another central feature is region polymorphism, which allows different invocations of the same function to operate in different regions. The translation has been proved sound with respect to a non-standard dynamic operational semantics, in which all values are put in regions that are allocated and deallocated in a stack-like manner.

Moreover, we have presented an algorithm for performing polymorphic region inference. The theoretical properties of the algorithm are not settled, but the algorithm has been implemented and tested on programs with presumed interesting memory behaviour.

A great deal of work remains, however. Concerning the theory presented in this report, it would be interesting to see how it is affected by the addition of side-effects, exceptions and more general (Standard ML-style) recursive datatypes.

As for the usefulness of our scheme, the experiments presented in this report suggest that the scheme in many cases leads to very economical use of memory resources, even without the use of garbage collection. They also reveal that there is a potentially serious overhead in maintaining regions, especially those that are allocated and deallocated very frequently and hold few values. Magnus Vejlstrup, a student at DIKU, is working on extending effect inference to infer the sizes of regions statically. Regions with known, finite size can be allocated on a hardware stack; small regions can in principle be represented by machine registers.

A runtime system written in C and a compiler from region-annotated programs into C are currently under development by Mads Tofte and Lars Birkedal. With the new runtime system we should be able to compare execution times and memory requirements of a region-based implementation directly with those of other implementations of ML-like

languages. One particularly interesting question is whether a region-based implementation gains execution speed because of good locality of reference, compared to other implementations.

Despite all these unresolved questions, we hope that this report has convinced the reader that our scheme rests on a solid theoretical foundation and holds practical promise.

Acknowledgments

Fritz Henglein's expertise on semi-unification was essential to the development of the algorithms in Section 5. Peter Sestoft provided many useful comments and his globalisation analysis gave rise to the storage mode analysis mentioned in Section 6.1.1. Lars Birkedal wrote parts of the implementation; in particular, he replaced the original, slow semi-unification module with a fast one. Andrew Appel, David MacQueen, Flemming Nielson, Hanne Riis Nielson, Dave Schmidt and David N. Turner contributed with discussions and valuable criticism.

The work is done with support from the DART project, under the The Danish Research Council.

References

[1] J. Tiuryn A. J. Kfoury and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, April 1993.

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

[3] Henry 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.

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

[5] David R. Chase. Safety considerations for storage allocation optimizations. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 1–10, ACM Press, June 22-24 1988.

[6] J. M. Lucassen D. K. Gifford, P. Jouvelot and M.A. Sheldon. *FX-87 Reference Manual*. Technical Report MIT/LCS/TR-407, MIT Laboratory for Computer Science, Sept 1987.

[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] Joëlle Despeyroux. Proof of translation in natural semantics. In *Proc. of the 1st Symp. on Logic in Computer Science, IEEE, Cambridge, USA*, 1986.

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

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

[11] P. Fradet. Syntactic detection of single-threading using continuations. In *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, Massachusetts, August 1991. (Lecture Notes in Computer Science, vol. 523)*, pages 241–258, Springer-Verlag, 1991.

[12] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253, April 1993.

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

[14] Pierre Jouvelot and D.K. Gifford. Algebraic reconstruction of types and effects. In *Proceedings of the 1991 ACM Conference on Principles of Programming Languages*, 1991.

[15] Åke Wikström. *Functional Programming Using Standard ML. Series in Computer Science*, Prentice Hall, 1987.

[16] A. Kfoury, J. Tiuryn, and P. Urzyczyn. The undecidability of the semi-unification problem. In *Proc. 22nd Annual ACM Symp. on Theory of Computation (STOC), Baltimore, Maryland*, pages 468–476, May 1990.

[17] Donald E. Knuth. *Fundamental Algorithms.* Volume 1 of *The Art of Computer Programming*, Addison-Wesley, 1972.

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

[19] 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.

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

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

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

[23] F. L. Morris. Advice on structuring compilers and proving them correct. In *Proc. ACM Symp. on Principles of Programming Languages*, 1973.

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

[25] Laurence C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[26] D. Rémy. Typechecking records and variants in a natural extension of ML. In *Proc. 16th Annual ACM Symp. on Principles of Programming Languages*, pages 77–88, ACM, Jan. 1989.

[27] Cristina Ruggieri and Thomas 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.

[28] D.A. Schmidt. Detecting global variables in denotational specifications. *ACM Transactions on Programming Languages and Systems*, 5(2):299–310, April 1985.

[29] Peter Sestoft. *Analysis and Efficient Implementation of Functional Programs*. PhD thesis, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK 2100 Copenhagen Ø, 1992.

[30] Jean-Pierre Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3), 1992. Also in Technical Report EMP-CRI-E150, Ecole Nationale Supérieure des Mines de Paris, February 1991.

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

[32] M. Wand. Type inference for record concatenation and multiple inheritance. In *Proc. 1989 IEEE 4th Annual Symp. on Logic in Computer Science (LICS)*, pages 92–97, IEEE Computer Society Press, 1989.

[33] Andrew K. Wright and Matthias Felleisen. *A Syntactic Approach to Type Soundness*. Technical Report Rice COMP TR91-160, Rice University, 1991.

A The Definition of Consistent

Let

$$\begin{aligned} U_1 &= \text{RegEnv} \times \text{TypeAndPlace} \times \text{Val} \times \text{Store} \times \text{TargetVal} \times \text{Effect} \\ U_2 &= \text{RegEnv} \times (\text{TypeScheme} \times \text{Place}) \times \text{Val} \times \text{Store} \times \text{TargetVal} \times \text{Effect} \\ U_3 &= \text{RegEnv} \times \text{TyEnv} \times \text{Env} \times \text{Store} \times \text{VarEnv} \times \text{Effect} \\ U &= U_1 \times U_2 \times U_3 \end{aligned}$$

Moreover, for every $Q \subseteq U$, let Q_i be the projection of Q on U_i, $i = 1 \ldots 3$.

Definition A.1 (F) *The map $F : P(U) \to P(U)$ is $F(Q) = F_1(Q) \times F_2(Q) \times F_3(Q)$, where*

$F_1(Q) = \{(R, \mu, v, s, v', \varphi) \mid$ *writing μ in the form (τ, p),*
if $\mathbf{get}(p) \in \varphi$ *then* $v' \in \mathrm{Dom}(s)$ *and* r of $v' = R(p)$ *and*

1. *If v is an integer i then $\tau =$* `int` *and $s(v') = i$;*
2. *If v is a pair (v_1, v_2) then $\tau = \mu_1 * \mu_2$ and $s(v')$ is a pair (v_1', v_2') and $(R, \mu_1, v_1, s, v_1', \varphi) \in Q_1$ and $(R, \mu_2, v_2, s, v_2', \varphi) \in Q_1$;*
3. *If v is a closure $\langle x, e, E\rangle$ then $s(v')$ is a closure $\langle x, e', VE\rangle$, for some e' and VE, and there exist TE, R' and e'' such that $TE \vdash \lambda x.e \Rightarrow \lambda x.e''$* `at` *$p : \mu, \{\mathbf{put}(p)\}$ and $(R, TE, E, s, VE, \varphi) \in Q_3$ and R' and R agree on φ and $R(e'') = e'$;*
4. *If v is a closure $\langle x, e, E, f\rangle$ then $s(v')$ is a closure $\langle x, e', VE\rangle$, for some e' and VE, and there exist TE, σ, p', R' and e'' such that $TE + \{f \mapsto (\sigma, p')\} \vdash \lambda x.e \Rightarrow \lambda x.e''$* `at` *$p : \mu, \{\mathbf{put}(p)\}$ and $(R, TE + \{f \mapsto (\sigma, p')\}, E + \{f \mapsto v\}, s, VE, \varphi) \in Q_3$ and R' and R agree on φ and $R'(e'') = e'\}$*

$F_2(Q) = \{(R, (\sigma, p), v, s, v', \varphi) \mid$
if $\mathbf{get}(p) \in \varphi$ *then* $v' \in \mathrm{Dom}(s)$ *and* r of $v' = R(p)$ *and*

1. *If σ is simple then for all τ if $\sigma \geq \tau$ then $(R, (\tau, p), v, s, v', \varphi) \in Q_1$;*
2. *If σ is compound, then v is a recursive closure $\langle x, e, E, f\rangle$ and $s(v') = \langle \rho_1', \cdots, \rho_k', x, e', VE\rangle$, for some $\rho_1', \ldots, \rho_k'$, e' and VE, and there exist TE, R' and e'' such that $(R, TE + \{f \mapsto (\sigma, p)\}, E + \{f \mapsto v\}, s, VE, \varphi) \in Q_3$ and σ can be written in the form $\forall\rho_1 \cdots \rho_k \forall\alpha_1 \cdots \alpha_n \forall\epsilon_1 \cdots \epsilon_m.\tau$, where none of the bound variables occur free in (TE, p), and $TE + \{f \mapsto (\sigma, p)\} \vdash \lambda x.e \Rightarrow \lambda x.e''$* `at` *$p : (\tau, p), \{\mathbf{put}(p)\}$ and R' and R agree on φ and $R'\langle \rho_1, \cdots, \rho_k, x, e'', VE\rangle = \langle \rho_1', \cdots, \rho_k', x, e', VE\rangle\}$*

$F_3(Q) = \{(R, TE, E, s, VE, \varphi) \mid \mathrm{Dom}\, TE = \mathrm{Dom}\, E = \mathrm{Dom}\, VE$ *and for all* $x \in \mathrm{Dom}\, TE$,
$(R, TE(x), E(x), s, VE(x), \varphi) \in Q_2\}$

Notice that F is monotonic with respect to set inclusion. By Tarski's fixed point theorem, F has a maximal fixed point, $Q^{\max}$.

B Proof of Lemma 4.2

For convenience, we repeat the Lemma here:

Lemma B.1 (same as Lemma 4.2) *If* Consistent(R_1, μ, 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* $s_1 \downarrow \mathrm{frn}(R_2\varphi_2) \sqsubseteq s_2$ *then* Consistent(R_2, μ, v, s_2, v') w.r.t. φ_2.

The lemma does not require that $\mathrm{frn}(R_2\varphi_1) \subseteq \mathrm{Dom}(s_1)$.

Proof By co-induction. Let $Q = Q_1 \times Q_2 \times Q_3$, where

$$Q_1 = \{(R_2, \mu, v, s_2, v', \varphi_2) \mid \text{there exist } R_1,\ s_1 \text{ and } \varphi_1 \text{ such that Consistent}(R_1, \mu, v, s_1, v') \text{ w.r.t. } \varphi_1 \text{ and } \varphi_2 \subseteq \varphi_1 \text{ and } R_2 \text{ and } R_1 \text{ agree on } \varphi_2 \text{ and } s_1 \downarrow \mathrm{frn}(R_2\varphi_2) \sqsubseteq s_2\}$$

$$Q_2 = \{(R_2, (\sigma, p), v, s_2, v', \varphi_2) \mid \text{there exist } R_1,\ s_1 \text{ and } \varphi_1 \text{ such that Consistent}(R_1, (\sigma, p), v, s_1, v') \text{ w.r.t. } \varphi_1 \text{ and } \varphi_2 \subseteq \varphi_1 \text{ and } R_2 \text{ and } R_1 \text{ agree on } \varphi_2 \text{ and } s_1 \downarrow \mathrm{frn}(R_2\varphi_2) \sqsubseteq s_2\}$$

$$Q_3 = \{(R_2, TE, E, s_2, VE, \varphi_2) \mid \text{there exist } R_1,\ s_1 \text{ and } \varphi_1 \text{ such that Consistent}(R_1, TE, E, s_1, VE) \text{ w.r.t. } \varphi_1 \text{ and } \varphi_2 \subseteq \varphi_1 \text{ and } R_2 \text{ and } R_1 \text{ agree on } \varphi_2 \text{ and } s_1 \downarrow \mathrm{frn}(R_2\varphi_2) \sqsubseteq s_2\}$$

Show $Q \subseteq F(Q)$. Take $q = (q_1, q_2, q_3) \in Q$.

First, write q_1 in the form $(R_2, \mu, v, s_2, v', \varphi_2)$. Then there exist R_1, s_1 and φ_1 such that

$$\text{Consistent}(R_1, \mu, v, s_1, v') \text{ w.r.t. } \varphi_1 \tag{201}$$

$$\varphi_2 \subseteq \varphi_1 \tag{202}$$

$$R_2 \text{ and } R_1 \text{ agree on } \varphi_2 \tag{203}$$

$$s_1 \downarrow \mathrm{frn}(R_2\varphi_2) \sqsubseteq s_2 \tag{204}$$

Write μ in the form (τ, p) and assume $\mathbf{get}(p) \in \varphi_2$. Then $\mathbf{get}(p) \in \varphi_1$, so by (201) we have $v' \in \mathrm{Dom}(s_1)$ and r of $v' = R_1(p)$. Since p occurs in φ_2 and (203) we therefore have r of $v' = R_2(p)$ and by (204), $v' \in \mathrm{Dom}(s_2)$. There are the following cases to consider, depending on v, according to the definition of F:

$\boxed{v \text{ is an integer } i.}$ By (201) we have that $\tau = \texttt{int}$ and $s_1(v') = i$. But then by (204) we have $s_2(v') = i$. Thus $q_1 \in F_1(Q)$, as desired.

$\boxed{v \text{ is a pair } (v_1, v_2).}$ By (201) we have $\tau = \mu_1 * \mu_2$ and $s_1(v') = (v'_1, v'_2)$, for some v'_1, v'_2, and Consistent$(R_1, \mu_i, v_i, s_1, v'_i)$ w.r.t. φ_1, for $i = 1, 2$. Thus $(R_2, \mu_i, v_i, s_2, v'_i, \varphi_2) \in Q_1$, for $i = 1, 2$. Also, by (204) we have $s_2(v') = (v'_1, v'_2)$. Thus $q_1 \in F_1(Q)$, as desired.

$\boxed{v \text{ is a closure } \langle x_0, e_0, E_0\rangle.}$ Then by (201) we have $s_1(v') = \langle x_0, e_0', VE_0\rangle$, for some e_0', VE_0 and there exist TE_0, R_0 and e_0'' such that

$$TE_0 \vdash \lambda x_0.e_0 \Rightarrow \lambda x_0.e_0'' \texttt{ at } p : \mu, \{\mathbf{put}(p)\} \tag{205}$$

$$\text{Consistent}(R_1, TE_0, E_0, s_1, VE_0) \text{ w.r.t. } \varphi_1 \tag{206}$$

$$R_0 \text{ and } R_1 \text{ agree on } \varphi_1 \tag{207}$$

$$R_0(e_0'') = e_0' \tag{208}$$

By (204) we have $s_2(v') = \langle x_0, e_0', VE_0\rangle$. Further, from (206) we get

$$(R_2, TE_0, E_0, s_2, VE_0, \varphi_2) \in Q_3 \tag{209}$$

From (203), (207) and $\varphi_2 \subseteq \varphi_1$ we get

$$R_0 \text{ and } R_2 \text{ agree on } \varphi_2 \tag{210}$$

Thus by (205), (209), (210) and (208) we get $q_1 \in F_1(Q)$ once again.

$\boxed{v \text{ is a recursive closure } \langle x_0, e_0, E_0, f_0\rangle.}$ Then by (201) we have $s_1(v') = \langle x_0, e_0', VE_0\rangle$, for some e_0', VE_0. Moreover, there exist TE_0, σ_0, p_0, R_0 and e_0'' such that

$$TE_0 + \{f_0 \mapsto (\sigma_0, p_0)\} \vdash \lambda x_0.e_0 \Rightarrow \lambda x_0.e_0'' \texttt{ at } p : \mu, \{\mathbf{put}(p)\} \tag{211}$$

$$\text{Consistent}(R_1, TE_0 + \{f_0 \mapsto (\sigma_0, p_0)\}, E_0 + \{f_0 \mapsto v\}, s_1, VE_0) \text{ w.r.t. } \varphi_1 \tag{212}$$

$$R_0 \text{ and } R_1 \text{ agree on } \varphi_1 \tag{213}$$

$$R_0(e_0'') = e_0' \tag{214}$$

By (204) we have $s_2(v') = s_1(v')$. Further, from (212) we get

$$(R_2, TE_0 + \{f_0 \mapsto (\sigma_0, p_0)\}, E_0 + \{f_0 \mapsto v\}, s_2, VE_0, \varphi_2) \in Q_3 \tag{215}$$

From (203), (213) and $\varphi_2 \subseteq \varphi_1$ we get

$$R_0 \text{ and } R_2 \text{ agree on } \varphi_2 \tag{216}$$

Thus by (211), (215), (216) and (214) we get $q_1 \in F_1(Q)$ once again.

Next, write q_2 in the form $(R_2, (\sigma, p), v, s_2, v', \varphi_2)$. Let R_1, s_1 and φ_1 be such that

$$\text{Consistent}(R_1, (\sigma, p), v, s_1, v') \text{ w.r.t. } \varphi_1 \tag{217}$$

$$\varphi_2 \subseteq \varphi_1 \tag{218}$$

$$R_2 \text{ and } R_1 \text{ agree on } \varphi_2 \tag{219}$$

$$s_1 \downarrow \mathrm{frn}(R_2\varphi_2) \sqsubseteq s_2 \tag{220}$$

Assume $\mathbf{get}(p) \in \varphi_2$. Then $\mathbf{get}(p) \in \varphi_1$. Thus by (217) we have $v' \in \mathrm{Dom}(s_1)$ and r of $v' = R_1(p)$. By (219) and (220) we have

$$v' \in \mathrm{Dom}(s_2) \text{ and } s_2(v') = s_1(v') \text{ and } r \text{ of } v' = R_2(p) \tag{221}$$

We now perform case analysis on σ:

$\boxed{\sigma \text{ is simple}}$ Take and τ with $\sigma \geq \tau$. By (217) we have

$$\mathrm{Consistent}(R_1, (\tau, p), v, s_1, v') \text{ w.r.t. } \varphi_1$$

Thus $(R_2, (\tau, p), v, s_2, v', \varphi_2) \in Q_1$. Thus $q_2 \in F_2(Q)$.

$\boxed{\sigma \text{ is compound}}$ By (217) we have that v is a recursive closure $\langle x_0, e_0, E_0, f_0\rangle$ and that

$$s_1(v') = \langle \rho'_1, \ldots, \rho'_k, x_0, e'_0, VE_0\rangle \tag{222}$$

for some $\rho'_1, \ldots, \rho'_k$, e'_0 and VE_0. Further, there exist TE_0, R_0 and e''_0 such that

$$\mathrm{Consistent}(R_1, TE_0 + \{f_0 \mapsto (\sigma, p)\}, E_0 + \{f_0 \mapsto v\}, s_1, VE_0) \text{ w.r.t. } \varphi_1 \tag{223}$$

$$\text{no bound variable of } \sigma \text{ is free in } (TE_0, p) \tag{224}$$

$$\sigma = \forall\rho_1 \cdots \rho_k \forall\alpha_1 \cdots \alpha_n \forall\epsilon_1 \cdots \epsilon_m . \tau \tag{225}$$

$$TE_0 + \{f_0 \mapsto (\sigma, p)\} \vdash \lambda\, x_0\, .\, e_0 \Rightarrow \lambda\, x_0\, .\, e''_0 \ \texttt{at}\ p : (\tau, p), \{\mathbf{put}(p)\} \tag{226}$$

$$R_0 \text{ and } R \text{ agree on } \varphi_1 \tag{227}$$

$$R_0\langle \rho_1, \ldots, \rho_k, x_0, e''_0, VE\rangle = \langle \rho'_1, \ldots, \rho'_k, x_0, e'_0, VE_0\rangle \tag{228}$$

By (221) we have $s_2(v') = s_1(v')$. From (223) we get

$$(R_2, TE_0 + \{f_0 \mapsto (\sigma, p)\}, E_0 + \{f_0 \mapsto v\}, s_2, VE_0, \varphi_2) \in Q_3 \tag{229}$$

Also, from (227) and $\varphi_2 \subseteq \varphi_1$ we have

$$R_0 \text{ and } R \text{ agree on } \varphi_2 \tag{230}$$

By (229), (224), (225), (226), (230) and (228) we have $q_2 \in F_2(Q)$ in this case as well.

Finally, write q_3 in the form $(R_2, TE, E, s_2, VE, \varphi_2)$. By the definition of Q_3 we have $\mathrm{Consistent}(R_1, TE, E, s_1, VE)$ w.r.t. φ_1. Thus $\mathrm{Dom}\, TE = \mathrm{Dom}\, E = \mathrm{Dom}\, VE$ and for all $x \in \mathrm{Dom}\, TE$, $\mathrm{Consistent}(R_1, TE(x), E(x), s_1, VE(x))$ w.r.t. φ_1. Thus, by the definition of Q_2, we have $(R_1, TE(x), E(x), s_2, VE(x), \varphi_2) \in Q_2$. Thus $q_3 \in F_3(Q)$.

This proves that $q \in F(Q)$, as desired.

C Proof of Lemma 4.3

For convenience, we repeat the Lemma here:

Lemma C.1 (Same as Lemma 4.3) *If* Consistent(R, TE, E, s, VE) w.r.t. φ *and* σ *is a compound type scheme* $\forall\vec{\rho}\forall\vec{\alpha}\forall\vec{\epsilon}.\tau$ *and no bound variable of* σ *is free in* (TE, p) *and* $TE + \{f \mapsto (\sigma, p)\} \vdash \lambda x.e_1 \Rightarrow \lambda x.e_1'$ `at` $p : (\tau, p), \{\mathbf{put}(p)\}$ *and* R' *and* R *agree on* φ *and* $R(p) = r$ *and* $r \in \mathrm{Dom}(s)$ *and* $o \notin \mathrm{Dom}(s(r))$ *and* $VE^+ = VE + \{f \mapsto (r, o)\}$ *and* $s^+ = s + \{(r, o) \mapsto R'\langle\vec{\rho}, x, e_1', VE^+\rangle\}$ *and* $TE^+ = TE + \{f \mapsto (\sigma, p)\}$ *and* $E^+ = E + \{f \mapsto \langle x, e_1, E, f\rangle\}$ *then* Consistent$(R, TE^+, E^+, s^+, VE^+)$ w.r.t. φ.

Proof By co-induction. Given R, R', TE, TE^+, E, E^+, s, s^+, VE, VE^+, φ, σ, $\vec{\rho}$, $\vec{\alpha}$, $\vec{\epsilon}$, τ, f, x, e_1, e_1', p, r and o satisfying the assumptions listed in the statement of the Lemma. Define $Q = Q_1 \times Q_2 \times Q_3$, where

$$\begin{aligned} Q_1 &= Q_1^{\max} \\ Q_2 &= Q_2^{\max} \cup \{(R, (\sigma, p), \langle x, e_1, E, f\rangle, s^+, (r, o), \varphi)\} \\ Q_3 &= Q_3^{\max} \cup \{(R, TE^+, E^+, s^+, VE^+, \varphi)\} \end{aligned}$$

We wish to show $Q \subseteq F(Q)$. Take $q = (q_1, q_2, q_3) \in Q$. The only interesting cases are:

$\boxed{q_2 = (R, (\sigma, p), \langle x, e_1, E, f\rangle, s^+, (r, o), \varphi)}$ Assume $\mathbf{get}(p) \in \varphi$. We have $(r, o) \in \mathrm{Dom}(s^+)$ and $r = R(p)$, as required by the definition of F. Since σ is compound, we have to check item 2 in the definition of F_2. But this is easily done, using that $(R, TE^+, E^+, VE^+, \varphi) \in Q_3$. Thus $q_2 \in F_2(Q)$.

$\boxed{q_3 = (R, TE^+, E^+, VE^+, \varphi)}$ Since Consistent(R, TE, E, s, VE) w.r.t. φ and $s \sqsubseteq s^+$ we have Consistent(R, TE, E, s^+, VE) w.r.t. φ. Thus $\mathrm{Dom}(TE) = \mathrm{Dom}(VE) = \mathrm{Dom}(E)$ and for all $x \in \mathrm{Dom}(TE)$, Consistent$(R, TE(x), E(x), s^+, VE(x))$ w.r.t. φ, so $(R, TE(x), E(x), s^+, VE(x), \varphi) \in Q_2$. Since also $(R, (\sigma, p), \langle x, e_1, E, f\rangle, s^+, (r, o), \varphi) \in Q_2$ we have $q_3 \in F_3(Q)$.

This proves $Q \subseteq F(Q)$, as desired.

D Proof of Lemma 4.5

For convenience, we repeat the Lemma here:

Lemma D.1 (same as Lemma4.5) *If* Consistent(R, μ, v, s, v')w.r.t.φ *and* S *is a region renaming of* μ *with respect to* φ *then* Consistent$(R, S(\mu), v, s, v')$ w.r.t. φ.

Proof By co-induction. Let $Q = Q_1 \times Q_2 \times Q_3$, where

$$\begin{aligned} Q_1 = \{(R, \mu', v, s, v', \varphi) \mid &\text{ there exist } \mu \text{ and } S \text{ such that } \mu' = S(\mu) \\ &\text{and } S \text{ is a region renaming of } \mu \text{ with respect to } \varphi \text{ and} \\ &\text{Consistent}(R, \mu, v, s, v') \text{ w.r.t. } \varphi\} \end{aligned}$$

$$Q_2 = \{(R, (\sigma', p'), v, s, v', \varphi) \mid \text{there exist } (\sigma, p) \text{ and } S \text{ such that } (\sigma', p') = S(\sigma, p) \text{ and } S \text{ is a region renaming of } (\sigma, p) \text{ with respect to } \varphi \text{ and Consistent}(R, (\sigma, p), v, s, v') \text{ w.r.t. } \varphi\}$$

$$Q_3 = \{(R, TE', E, s, VE, \varphi) \mid \text{there exist } TE \text{ and } S \text{ such that } TE' = S(TE) \text{ and } S \text{ is a region renaming of } TE \text{ with respect to } \varphi \text{ and Consistent}(R, TE, E, s, VE) \text{ w.r.t. } \varphi\}$$

Show $Q \subseteq F(Q)$. Take $q = (q_1, q_2, q_3) \in Q$.

First, write q_1 in the form $(R, \mu', v, s, v', \varphi)$. Then there exist μ and S such that

$$\mu' = S(\mu) \tag{231}$$

$$S \text{ is a region renaming of } \mu \text{ with respect to } \varphi \tag{232}$$

$$\text{Consistent}(R, \mu, v, s, v') \text{ w.r.t. } \varphi \tag{233}$$

Write μ in the form (τ, p). Then $\mu' = S(\mu) = (S\tau, S(p))$. Assume $\mathbf{get}(S(p)) \in \varphi$. By (232) we have $S(p) = p$ and $\mathbf{get}(p) \in \varphi$. Thus, by (233), $v' \in \text{Dom}(s)$ and $(r \,\text{of}\, v') = R(p)$. There are the following cases to consider, according to the definition of F:

$\boxed{v \text{ is an integer } i.}$ By (233) we have $\tau = \texttt{int}$ and $s(v') = i$. Thus $S(\tau) = \texttt{int}$. Thus $q_1 \in F_1(Q)$ in this case.

$\boxed{v \text{ is a pair } (v_1, v_2).}$ By (233) we have $\tau = \mu_1 * \mu_2$ and $s(v')$ is a pair (v_1', v_2') and $\text{Consistent}(R, \mu_i, v_i, s, v_i')$ w.r.t. φ, $i = 1, 2$. Note that S is a region renaming of μ_i with respect to φ. Thus $(R, S(\mu_i), v_i, s, v_i', \varphi) \in Q_1$, $i = 1, 2$. Since $S(\tau) = S(\mu_1) * S(\mu_2)$ we therefore have $q_1 \in F_1(Q)$ in this case as well.

$\boxed{v \text{ is a closure } \langle x_0, e_0, E_0 \rangle.}$ By (233) we have $s(v')$ is a closure $\langle x_0, e_0', VE_0 \rangle$, for some e_0' and VE_0. Moreover, there exist TE_0, R_0 and e_0'' such that

$$TE_0 \vdash \lambda x_0.e_0 \Rightarrow \lambda x_0.e_0'' \texttt{ at } p : \mu, \{\mathbf{put}(p)\} \tag{234}$$

$$\text{Consistent}(R, TE_0, E_0, s, VE_0) \text{ w.r.t. } \varphi \tag{235}$$

$$R_0 \text{ and } R \text{ agree on } \varphi \tag{236}$$

$$R_0(e_0'') = e_0' \tag{237}$$

We wish to find TE_0', R_0' and e_0''' that demonstrate that $q_1 \in F_1(Q)$. In particular, we want

$$TE_0' \vdash \lambda x_0.e_0 \Rightarrow \lambda x_0.e_0''' \texttt{ at } p : \mu', \{\mathbf{put}(p)\} \tag{238}$$

where the crucial detail is that we have μ', not μ, on the right-hand side. Comparing (238) and (234), a tempting idea is simply to apply S throughout (234). However, this

will not work in general, for we cannot be sure that there will exist an R_0' such that $R_0'(S(e_0'')) = e_0'$ (compare with (237)). The reason for this is that there may be one or more region variables that occur free in e_0'', without occurring free in μ, so S is not necessarily injective on $\mathrm{frv}(e_0'')$. If S is not injective on $\mathrm{frv}(e_0'')$ then it may be impossible to find an R_0' satisfying $R_0'(S(e_0'')) = e_0'$.

To overcome this problem, we rename the problematic region variables in e_0'' (and TE_0) as follows. Let

$$\{\rho_1, \ldots, \rho_n\} = \mathrm{frv}(e_0'', TE_0) \setminus \mathrm{frv}(\mu, \varphi)$$

Let $\{\rho_1', \ldots, \rho_n'\}$ be distinct new region variables, new in the sense that for all ρ_i', $\rho_i' \notin \mathrm{frv}(S(\mu), \varphi)$. Let $S' = S + \{\rho_i \mapsto \rho_i' \mid 1 \leq i \leq n\}$, let $TE_0' = S'(TE_0)$, let $e_0''' = S'(e_0'')$ and let $R_0' = R_0 \circ ((S \downarrow \mathrm{frv}\,\mu)^{-1} \cup \{\rho_i' \mapsto \rho_i \mid 1 \leq i \leq n\})$. By Lemma 3.1 on (234) we have

$$TE_0' \vdash \lambda x_0.e_0 \Rightarrow S'(\lambda x_0.e_0'' \texttt{ at } p) : S'(\mu), S'(\{\mathbf{put}(p)\}) \tag{239}$$

Since p occurs in μ we have $S'(p) = S(p) = p$. Indeed, by the definition of S', we have $S'(\mu) = S(\mu) = \mu'$. Thus (239) reads

$$TE_0' \vdash \lambda x_0.e_0 \Rightarrow \lambda x_0.e_0''' \texttt{ at } p : \mu', \{\mathbf{put}(p)\} \tag{240}$$

By definition of S', S' is a region renaming of TE_0 with respect to φ. Thus, by (235) and the definition of Q_3,

$$(R, TE_0', E_0, s, VE_0, \varphi) \in Q_3 \tag{241}$$

Also by the definition of R_0' and the fact that S is a region renaming of μ with respect to φ, R_0' and R_0 agree on φ. Then by (236),

$$R_0' \text{ and } R \text{ agree on } \varphi \tag{242}$$

Finally, $R_0'(e_0''') = R_0'(S'(e_0'')) = R_0(e_0'')$ by the definition of S' and R_0'. Thus by (237) we have

$$R_0'(e_0''') = e_0' \tag{243}$$

From (240), (241), (242) and (243) we have that $q_1 \in F_1(Q)$ in this case as well.

v is a recursive closure $\langle x_0, e_0, E_0, f_0\rangle$. This proof case is very similar to the previous one. By (233) we have $s(v')$ is a closure $\langle x_0, e_0', VE_0\rangle$, for some e_0' and VE_0. Moreover, there exist TE_0, σ_0, p_0, R_0 and e_0'' such that

$$TE_0 + \{f_0 \mapsto (\sigma_0, p_0)\} \vdash \lambda x_0.e_0 \Rightarrow \lambda x_0.e_0'' \texttt{ at } p : \mu, \{\mathbf{put}(p)\} \tag{244}$$

$$\text{Consistent}(R, TE_0 + \{f_0 \mapsto (\sigma_0, p_0)\}, E_0 + \{f_0 \mapsto v\}, s, VE_0) \text{ w.r.t. } \varphi \tag{245}$$

$$R_0 \text{ and } R \text{ agree on } \varphi \tag{246}$$

$$R_0(e_0'') = e_0' \tag{247}$$

We wish to find TE'_0, σ'_0, p'_0, R'_0 and e'''_0 that demonstrate that $q_1 \in F_1(Q)$. In particular, we want

$$TE'_0 + \{f_0 \mapsto (\sigma'_0, p'_0)\} \vdash \lambda x_0.e_0 \Rightarrow \lambda x_0.e'''_0 \texttt{ at } p : \mu', \{\mathbf{put}(p)\} \tag{248}$$

For the same reasons as in the previous proof case, let

$$\{\rho_1, \ldots, \rho_n\} = \mathrm{frv}(e''_0, TE_0 + \{f_0 \mapsto (\sigma_0, p_0)\}) \setminus \mathrm{frv}(\mu, \varphi)$$

Let $\{\rho'_1, \ldots, \rho'_n\}$ be distinct new region variables, new in the sense that for all ρ'_i, $\rho'_i \notin \mathrm{frv}(S(\mu), \varphi)$. Let $S' = S + \{\rho_i \mapsto \rho'_i \mid 1 \leq i \leq n\}$, let $TE'_0 = S'(TE_0)$, let $\sigma'_0 = S'(\sigma_0)$ and let $p'_0 = S'(p_0)$. Further, let $e'''_0 = S'(e''_0)$ and let $R'_0 = R_0 \circ ((S \downarrow \mathrm{frv}\,\mu)^{-1} \cup \{\rho'_i \mapsto \rho_i \mid 1 \leq i \leq n\})$. By Lemma 3.1 on (244) we have

$$TE'_0 + \{f_0 \mapsto (\sigma'_0, p'_0)\} \vdash \lambda x_0.e_0 \Rightarrow \lambda x_0.e'''_0 \texttt{ at } p : \mu', \{\mathbf{put}(p)\} \tag{249}$$

By definition of S', S' is a region renaming of $TE_0 + \{f_0 \mapsto (\sigma_0, p_0)\}$ with respect to φ. Thus, by (245) and the definition of Q_3,

$$(R, TE'_0 + \{f_0 \mapsto (\sigma'_0, p'_0)\}, E_0 + \{f_0 \mapsto v\}, s, VE_0, \varphi) \in Q_3 \tag{250}$$

Also by the definition of R'_0 and the fact that S is a region renaming of μ with respect to φ, R'_0 and R_0 agree on φ. Then by (246),

$$R'_0 \text{ and } R \text{ agree on } \varphi \tag{251}$$

Finally, $R'_0(e'''_0) = R'_0(S'(e''_0)) = R_0(e''_0)$ by the definition of S' and R'_0. Thus by (247) we have

$$R'_0(e'''_0) = e'_0 \tag{252}$$

From (249), (250), (251) and (252) we have that $q_1 \in F_1(Q)$ in this case as well.

Next, write q_2 in the form $(R, (\sigma', p'), v, s, v', \varphi)$. Let σ, p and S be such that

$$(\sigma', p') = S(\sigma, p) \tag{253}$$

$$S \text{ is a region renaming of } (\sigma, p) \text{ with respect to } \varphi \tag{254}$$

$$\mathrm{Consistent}(R, (\sigma, p), v, s, v') \text{ w.r.t. } \varphi \tag{255}$$

Assume $\mathbf{get}(p') \in \varphi$, i.e., $\mathbf{get}(Sp) \in \varphi$. By (254) we have $S(p) = p$. Thus $p = p'$. Thus $\mathbf{get}(p) \in \varphi$. Thus by (255) we have $v' \in \mathrm{Dom}(s)$ and r of $v' = R(p)$. We now make the case analysis according to the definition of F:

$\boxed{\sigma' \text{ is simple}}$ Let τ' be any type with $\sigma' \geq \tau'$. We wish to prove $(R, (\tau', p'), v, s, v', \varphi) \in Q_1$. In other words, we wish to find S^+ and τ such that

$$S^+(\tau, p) = (\tau', p') \tag{256}$$

$$S^+ \text{ is a region renaming of } (\tau, p) \text{ with respect to } \varphi \tag{257}$$

$$\text{Consistent}(R, (\tau, p), v, s, v') \text{ w.r.t. } \varphi \tag{258}$$

Now (258) follows directly from (255), if we can show

$$\sigma \geq \tau \tag{259}$$

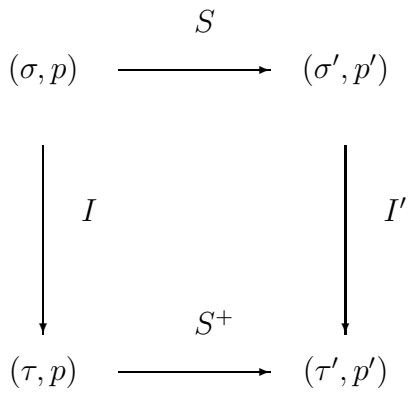
We now prove that there exist S^+ and τ that satisfy (256), (257) and (259). Since σ' is simple, it can be written in the form

$$\sigma' = \forall\alpha'_1 \cdots \alpha'_n \forall\epsilon'_1 \cdots \epsilon'_m . \tau'_0$$

Since $S(\sigma) = \sigma'$, we can write σ in the form

$$\sigma = \forall\alpha_1 \cdots \alpha_n \forall\epsilon_1 \cdots \epsilon_m . \tau_0 \quad \text{(same } m \text{ and } n\text{)}$$

Since $\sigma' \geq \tau'$ there exists a finite substitution, I', whose domain is the bound variables of σ' and which satisfies $I'(\tau'_0) = \tau'$. The following diagram illustrates the situation:



The horizontal arrows are substitutions on free variables, whereas the vertical arrows are instantiations of bound variables. S and I' are given; we seek I, τ and S^+ to make the diagram commute.

As a first naïve attempt, one might try to define

$$\begin{aligned} I(\alpha_i) &= I'(\alpha'_i) \quad i = 1..n \\ I(\epsilon_j) &= I'(\epsilon'_j) \quad j = 1..m \end{aligned}$$

Unfortunately, there may be type- or effect variables that occur free in (σ, p), are in the support of S and are free in the range of I'. Such variables can make it impossible for the diagram to commute.

A second attempt might then be to rename all variables that occur free in the range of I' to fresh variables — call the renaming substitution S_0 — and define

$$\begin{aligned} I(\alpha_i) &= S_0(I'(\alpha'_i)) \quad i = 1..n \\ I(\epsilon_j) &= S_0(I'(\epsilon'_j)) \quad j = 1..m \end{aligned}$$

Now we can find S^+ to make the diagram commute, but S^+ will not in general be a region renaming. More specifically, $S^+ \downarrow (\mathrm{frv}(\tau, p))$ need not be injective.

To overcome these problems, we partition the set of variables that occur free in the range of I' into those that should be renamed, and those that should not, as follows. Let $R = \mathrm{frv}(\mathrm{Rng}(I'))$. Partition R into the three sets:

$$\begin{aligned} R_1 &= \{\rho \in R \mid \exists \rho' \in \mathrm{frv}(\sigma, p) \cap \mathrm{Supp}(S).S(\rho') = \rho\} \\ R_2 &= \{\rho \in R \mid \rho \in \mathrm{frv}(\sigma, p) \setminus \mathrm{Supp}(S)\} \\ R_3 &= \{\rho \in R \mid \rho \notin R_1 \wedge \rho \in \mathrm{frv}(\sigma, p) \cap \mathrm{Supp}(S)\} \end{aligned}$$

Note that R_1, R_2 and R_3 are pairwise disjoint. (That $R_1 \cap R_2 = \emptyset$ follows from the fact that $S \downarrow (\mathrm{frv}(\sigma, p))$ is injective.)

Let S_3 be a bijective mapping $S_3 : R_3 \to R_3^{\mathrm{new}}$, where R_3^{new} is a set of fresh region variables. Let $S_1 : R_1 \to \mathrm{frv}(\sigma, p)$ be the map $S_1(\rho_1) = (S \downarrow \mathrm{frv}(\sigma, p))^{-1}(\rho_1)$. Moreover, let S_4 be a bijective map mapping type- and effect variables free in $\mathrm{Rng}(I')$ to fresh type- and effect variables. We now define I by

$$\begin{aligned} I(\alpha_i) &= (S_1 + S_3 + S_4)(I'(\alpha_i')) \quad i = 1..n \\ I(\epsilon_j) &= (S_1 + S_3 + S_4)(I'(\epsilon_j')) \quad j = 1..m \end{aligned}$$

Further, let $\tau = I\tau_0$ and let $S^+ = S + S_3^{-1} + S_4^{-1}$. Certainly $\sigma \geq \tau$, as required at (259). As for (257), first note that, by construction, $S^+ \downarrow \mathrm{frv}(\tau, p)$ is injective. Moreover, for all $\rho \in \mathrm{Supp}(S^+)$ we have $\rho \notin \mathrm{frv}(\varphi)$ (since R_3^{new} and $\mathrm{frv}(\varphi)$ can be assumed to be disjoint) and $S^+(\rho) \notin \mathrm{frv}(\varphi)$. (That $S^+(\rho) \notin \mathrm{frv}(\varphi)$ in the case where $\rho \in R_3^{\mathrm{new}}$ is seen as follows: $S^+(\rho) \in R_3$ so $S^+(\rho) \in \mathrm{Supp}(S)$, so $S^+(\rho) \notin \mathrm{frv}(\varphi)$, since S is a region renaming with respect to φ.) This proves (257).

Finally, (256) follows directly from the definition of S^+ and τ.

$\boxed{\sigma' \text{ is compound}}$ Then σ is compound, so by (255) we have that v is a recursive closure $\langle x_0, e_0, E_0, f_0\rangle$ and $s(v') = \langle \rho_1', \ldots, \rho_k', x_0, e_0', VE_0\rangle$, for some $\rho_1', \ldots, \rho_k'$, e_0' and VE_0. Furthermore, there exist TE_0, R_0, e_0'', $\rho_1, \ldots, \rho_k$, $\alpha_1, \ldots, \alpha_n$, $\epsilon_1, \ldots, \epsilon_m$ and τ such that

$$\mathrm{Consistent}(R, TE_0 + \{f_0 \mapsto (\sigma, p)\}, E_0 + \{f_0 \mapsto v\}, s, VE_0) \text{ w.r.t. } \varphi \tag{260}$$

$$\sigma = \forall \rho_1 \cdots \rho_k \forall \alpha_1 \cdots \alpha_n \forall \epsilon_1 \cdots \epsilon_m . \tau \tag{261}$$

$$\text{No bound variable of } \sigma \text{ is free in } (TE_0, p) \tag{262}$$

$$TE_0 + \{f_0 \mapsto (\sigma, p)\} \vdash \lambda x_0.e_0 \Rightarrow \lambda x_0.e_0'' \texttt{ at } p : (\tau, p), \{\mathbf{put}(p)\} \tag{263}$$

$$R_0 \text{ and } R \text{ agree on } \varphi \tag{264}$$

$$R_0\langle \rho_1, \ldots, \rho_k, x_0, e_0'', VE_0\rangle = \langle \rho_1', \ldots, \rho_k', x_0, e_0', VE_0\rangle \tag{265}$$

Let $\{\rho_1^{old}, \ldots, \rho_l^{old}\} = \mathrm{frv}(e_0'', TE_0, \tau) \setminus \mathrm{frv}((\sigma, p), \varphi)$. Let $\{\rho_1^{new}, \ldots, \rho_l^{new}\}$ be distinct new region variables, new in the sense that

$$\{\rho_1^{new}, \ldots, \rho_l^{new}\} \cap \mathrm{frv}(S(\sigma, p), \varphi) = \emptyset \tag{266}$$

Further, let $\alpha'_1, \ldots, \alpha'_n$ be new type variables and let $\epsilon'_1, \ldots, \epsilon'_m$ be new effect variables. Let

$$\begin{aligned} S' = S + (&\{\rho_1^{old} \mapsto \rho_1^{new}, \ldots, \rho_l^{old} \mapsto \rho_l^{new}\}, \\ &\{\alpha_1 \mapsto \alpha'_1, \ldots, \alpha_n \mapsto \alpha'_n\}, \\ &\{\epsilon_1 \mapsto \epsilon'_1.\emptyset, \ldots, \epsilon_m \mapsto \epsilon'_m.\emptyset\}) \end{aligned}$$

Let $TE'_0 = S'(TE_0)$ and let $e'''_0 = S'(e''_0)$. By Lemma 3.1 on (263) we have

$$S'(TE_0) + \{f_0 \mapsto (S'\sigma, p)\} \vdash \lambda x_0.e_0 \Rightarrow \lambda x_0.e'''_0 \texttt{ at } p : (S'\tau, p), \{\mathbf{put}(p)\} \tag{267}$$

where we have used $S'(p) = p$. Now S' is such that

$$S'(\sigma) = \forall S'\rho_1, \ldots, S'\rho_k.\forall \alpha'_1, \ldots, \alpha'_n.\forall \epsilon'_1, \ldots, \epsilon'_m.S'(\tau) \tag{268}$$

and

$$(\{S'\rho_1, \ldots, S'\rho_k\}, \{\alpha'_1, \ldots, \alpha'_n\}, \{\epsilon'_1, \ldots, \epsilon'_m\}) \cap \mathrm{fv}(TE'_0, p) = \emptyset \tag{269}$$

Notice that S' is a region renaming of $TE_0 + \{f_0 \mapsto (\sigma, p)\}$ with respect to φ. Thus by (260) we have

$$(R, TE'_0 + \{f_0 \mapsto (S'\sigma, p)\}, E_0 + \{f_0 \mapsto v\}, s, VE_0, \varphi) \in Q_3 \tag{270}$$

Finally, let $R'_0 = R_0 \circ ((S \downarrow \mathrm{frv}(\sigma, p))^{-1} \cup \{\rho_1^{new} \mapsto \rho_1^{old}, \ldots, \rho_l^{new} \mapsto \rho_l^{old}\})$. Clearly, R'_0 and R_0 agree on φ. But then, by (264), we have

$$R'_0 \text{ and } R \text{ agree on } \varphi \tag{271}$$

Finally, we wish to prove

$$R'_0\langle S'\rho_1, \ldots, S'\rho_k, x_0, e'''_0, VE_0\rangle = \langle \rho'_1, \ldots, \rho'_k, x_0, e'_0, VE_0\rangle \tag{272}$$

i.e., regarding Λ as a binding operator for region variables:

$$R'_0(\Lambda S'\rho_1, \cdots, S'\rho_k.S'e''_0) = \langle \rho'_1, \ldots, \rho'_k, x_0, e'_0, VE_0\rangle \tag{273}$$

By (265) it will suffice to prove

$$R'_0(\Lambda S'\rho_1 \cdots S'\rho_k.S'e''_0) = R_0(\Lambda \rho_1, \cdots, \rho_k.e''_0) \tag{274}$$

Take $\rho \in \mathrm{frv}(e''_0)$. We now make a simple case analysis. *Case 1:* If $\rho \notin \{\rho_1, \ldots, \rho_n\}$ then ρ will be mapped to a free occurrence of $R_0(\rho)$ on both the left-hand side and the right-hand side of (274). *Case 2:* If $\rho \in \{\rho_1, \ldots, \rho_n\}$ then the (bound) occurrence of ρ on the right-hand side is mapped to a bound occurrence of $S'\rho$ on the left-hand side.

This proves (272). By (270), (268), (267), (271) and (272) we have $q_2 \in F_2(Q)$ in this case as well.

Finally, write q_3 in the form $(R, TE', E, s, VE, \varphi)$. By the definition of Q_3 there exist TE and S such that $TE' = S(TE)$ and S is a region renaming of TE with respect to φ and $\mathrm{Consistent}(R, TE, E, s, VE)$ w.r.t. φ. In particular, $\mathrm{Dom}\, TE = \mathrm{Dom}\, E = \mathrm{Dom}\, VE$ and, for all $x \in \mathrm{Dom}\, TE$, $\mathrm{Consistent}(R, TE(x), E(x), s, VE(x))$ w.r.t. φ. Since $S(TE(x)) = TE'(x)$ and S is a region renaming of $TE(x)$ with respect to φ we therefore have $(R, TE'(x), E(x), s, VE(x), \varphi) \in Q_2$. Thus $q_3 \in F_3(Q)$.

This proves that $q \in F(Q)$, as desired.

E Proof of Lemma 4.4

For convenience, we repeat the Lemma here:

Lemma E.1 (same as Lemma 4.4) *If* Consistent(R, TE, E, s, VE) w.r.t. φ *and* $\rho \notin \mathrm{frv}(TE, \varphi)$, $r \notin \mathrm{Dom}(s)$ *and* $\varphi' \subseteq \{\mathbf{put}(\rho), \mathbf{get}(\rho)\} \cup \{\epsilon_1, \ldots, \epsilon_k\}$, *where* $\epsilon_1, \ldots, \epsilon_k$ *are effect variables* $(k \geq 0)$ *then* Consistent$(R+\{\rho \mapsto r\}, TE, E, s+\{r \mapsto \{\}\}, VE)$w.r.t.$\varphi' \cup \varphi$.

Proof By co-induction. Let s, r, ρ, φ, φ', $\epsilon_1, \ldots, \epsilon_k$ and R be given and assume $r \notin \mathrm{Dom}(s)$ and $\varphi' \subseteq \{\mathbf{put}(\rho), \mathbf{get}(\rho)\} \cup \{\epsilon_1, \ldots, \epsilon_k\}$. Let $s' = s + \{r \mapsto \{\}\}$ and let $R' = R + \{\rho \mapsto r\}$. Throughout this proof, we use S to range over *simple* substitutions, i.e. substitutions that have no region variables in their support. We now define (R, R', s, s', φ, φ' , $\epsilon_1, \ldots, \epsilon_k$ and ρ are fixed):

$$Q_1 = \{(R', \mu', v, s', v', \varphi' \cup \varphi) \mid \text{there exist } \mu \text{ and } S \text{ such that } S(\mu) = \mu' \text{ and Consistent}(R, \mu, v, s, v') \text{ w.r.t. } \varphi \text{ and } \rho \notin \mathrm{frv}(\mu, \varphi)\}$$

$$Q_2 = \{(R', (\sigma', p), v, s', v', \varphi' \cup \varphi) \mid \text{there exist } \sigma \text{ and } S \text{ such that } S(\sigma) = \sigma' \text{ and Consistent}(R, (\sigma, p), v, s, v') \text{ w.r.t. } \varphi \text{ and } \rho \notin \mathrm{frv}((\sigma, p), \varphi)\}$$

$$Q_3 = \{(R', TE', E, s', VE, \varphi' \cup \varphi) \mid \text{there exist } TE \text{ and } S \text{ such that } S(TE) = TE' \text{ and Consistent}(R, TE, E, s, VE) \text{ w.r.t. } \varphi \text{ and } \rho \notin \mathrm{frv}(TE, \varphi)\}$$

Let $Q = Q_1 \times Q_2 \times Q_3$. Show $Q \subseteq F(Q)$. By co-induction we then have $Q \subseteq Q^{\max}$. This gives the desired result for $S = \mathrm{Id}$. Incidentally, it does not appear to be possible to prove the lemma directly, i.e. without proving the stronger statement involving S; the case for type schemes causes problems: even though $\rho \notin \mathrm{frv}(\sigma, \varphi)$ there will always be an instance μ of σ such that $\rho \in \mathrm{frv}(\mu, \varphi)$.

Take $q = (q_1, q_2, q_3) \in Q$. Write q_1 in the form $(R', \mu', v, s', v', \varphi' \cup \varphi)$. Then there exist μ and S such that

$$S(\mu) = \mu' \tag{275}$$

$$\text{Consistent}(R, \mu, v, s, v') \text{ w.r.t. } \varphi \tag{276}$$

$$\rho \notin \mathrm{frv}(\mu, \varphi) \tag{277}$$

Write μ in the form (τ, p). Then $\mu' = S(\mu) = (S\tau, p)$, since S is simple. Assume $\mathbf{get}(p) \in \varphi' \cup \varphi$. By (277) we have $\rho \neq p$, so $\mathbf{get}(p) \notin \varphi'$. Thus $\mathbf{get}(p) \in \varphi$. Thus by (276) we have that $v' \in \mathrm{Dom}(s)$ and $(r \text{ of } v') = R(p)$. Thus, since $r \notin \mathrm{Dom}(s)$, we have $v' \in \mathrm{Dom}(s')$ and, since $\rho \neq p$, we have $(r \text{ of } v') = R'(p)$. There are the following cases to consider, according to the definition of F.

$\boxed{v \text{ is an integer } i.}$ By (276) and $\mathbf{get}(p) \in \varphi$ we have $\tau = \mathtt{int}$ and $s(v') = i$. Since $r \notin \mathrm{Dom}(s)$ we then have $s'(v') = i$. Also $S(\tau) = \mathtt{int}$. Thus $q_1 \in F_1(Q)$ in this case.

$\boxed{v \text{ is a pair } (v_1, v_2).}$ By (276) and $\mathbf{get}(p) \in \varphi$ we have $\tau = \mu_1 * \mu_2$ and $s(v')$ is a pair (v_1', v_2') and Consistent(R, μ_i, v_i, s, v_i') w.r.t. φ, $i = 1, 2$. Thus $S(\tau) = S(\mu_1) * S(\mu_2)$ and

$\rho \notin \mathrm{frv}(\mu_i, \varphi)$, $i = 1, 2$. By the definition of Q_1, we therefore have $(R', S(\mu_i), s', v'_i, \varphi' \cup \varphi) \in Q_1$, $i = 1, 2$. Also, since $r \notin \mathrm{Dom}(s)$, we have $s'(v') = (v'_1, v'_2)$. Thus $q_1 \in F_1(Q)$ in this case as well.

$\boxed{v \text{ is a closure } \langle x_0, e_0, E_0 \rangle.}$ By (276) and $\mathbf{get}(p) \in \varphi$ we have that $s(v')$ is a closure $\langle x_0, e'_0, VE_0 \rangle$, for some e'_0 and VE_0. Moreover, there exist TE_0, R_0 and e''_0 such that

$$TE_0 \vdash \lambda x_0.e_0 \Rightarrow \lambda x_0.e''_0 \texttt{ at } p : \mu, \{\mathbf{put}(p)\} \tag{278}$$

$$\mathrm{Consistent}(R, TE_0, E_0, s, VE_0) \text{ w.r.t. } \varphi \tag{279}$$

$$R_0 \text{ and } R \text{ agree on } \varphi \tag{280}$$

$$R_0(e''_0) = e'_0 \tag{281}$$

Without loss of generality, we can assume that $\rho \notin \mathrm{frv}(TE_0, e''_0)$; for if $\rho \in \mathrm{frv}(TE_0, e''_0)$ we can find new TE_0, e''_0 and R_0 satisfying the above equations *and* $\rho \notin \mathrm{frv}(TE_0, e''_0)$ as follows. Let ρ' be a region variable with $\rho' \notin \mathrm{frv}(TE_0, e''_0) \cup \mathrm{frv}\,\varphi$. Let $R^r = \{\rho \mapsto \rho'\}$. By (277) $R^r(p) = p$ and $R^r(\mu) = \mu$, so by Lemma 3.1 on (278) we have

$$R^r(TE_0) \vdash \lambda x_0.e_0 \Rightarrow \lambda x_0.R^r e''_0 \texttt{ at } p : \mu, \{\mathbf{put}(p)\} \tag{282}$$

Also by Lemma 4.5 on (279) we get $\mathrm{Consistent}(R, R^r(TE_0), E_0, s, VE_0)\,\mathrm{w.r.t.}\,\varphi$. Moreover, letting $R^r_0 = R_0 + \{\rho' \mapsto R_0(\rho)\}$ we get from (280) and (281) that R^r_0 and R agree on φ and $R^r_0(R^r e''_0) = e'_0$, so if $\rho \in \mathrm{frv}(TE_0, e''_0)$ we just continue using $R^r TE_0$ for TE_0, $R^r e''_0$ for e''_0 and R^r_0 for R_0. We thus assume

$$\rho \notin \mathrm{frv}(TE_0, e''_0) \tag{283}$$

By (279) and the definition of Q_3 we then have

$$(R', S(TE_0), E_0, s', VE_0, \varphi' \cup \varphi) \in Q_3 \tag{284}$$

By Lemma 3.1 on (278) we have

$$S(TE_0) \vdash \lambda x_0.e_0 \Rightarrow S(\lambda x_0.e''_0 \texttt{ at } p) : S(\mu), S(\{\mathbf{put}(p)\}) \tag{285}$$

i.e. since S is simple and $S(\mu) = \mu'$:

$$S(TE_0) \vdash \lambda x_0.e_0 \Rightarrow \lambda x_0.e''_0 \texttt{ at } p : \mu', \{\mathbf{put}(p)\} \tag{286}$$

Let $R'_0 = R_0 + \{\rho \mapsto r\}$. From (280) we get

$$R'_0 \text{ and } R' \text{ agree on } \varphi' \cup \varphi \tag{287}$$

and from (281) and (283) we get

$$R'_0(e''_0) = e'_0 \tag{288}$$

From (286), (284), (287) and (288) and the definition of F, we get $q_1 \in F_1(Q)$ in this case as well.

v is a recursive closure $\langle x_0, e_0, E_0, f_0 \rangle$. This proof case is similar to the previous one. By (276) and $\mathbf{get}(p) \in \varphi$ we have that $s(v')$ is a closure $\langle x_0, e_0', VE_0 \rangle$, for some e_0' and VE_0. Moreover, there exist TE_0, σ_0, p_0, R_0 and e_0'' such that

$$TE_0 + \{f_0 \mapsto (\sigma_0, p_0)\} \vdash \lambda x_0.e_0 \Rightarrow \lambda x_0.e_0'' \texttt{ at } p : \mu, \{\mathbf{put}(p)\} \tag{289}$$

$$\text{Consistent}(R, TE_0 + \{f_0 \mapsto (\sigma_0, p_0)\}, E_0 + \{f_0 \mapsto v\}, s, VE_0) \text{ w.r.t. } \varphi \tag{290}$$

$$R_0 \text{ and } R \text{ agree on } \varphi \tag{291}$$

$$R_0(e_0'') = e_0' \tag{292}$$

As in the previous proof case, we can choose TE_0, σ_0, p_0, R_0 and e_0'' such that in addition to the above,

$$\rho \notin \text{frv}(TE_0 + \{f_0 \mapsto (\sigma_0, p_0)\}, e_0'') \tag{293}$$

By Lemma 3.1 on (289) we have

$$S(TE_0 + \{f_0 \mapsto (\sigma_0, p_0)\}) \vdash \lambda x_0.e_0 \Rightarrow S(\lambda x_0.e_0'' \texttt{ at } p) : S(\mu), S(\{\mathbf{put}(p)\}) \tag{294}$$

i.e. since S is simple and $S(\mu) = \mu'$:

$$S(TE_0) + \{f_0 \mapsto (S(\sigma_0), p_0)\} \vdash \lambda x_0.e_0 \Rightarrow \lambda x_0.e_0'' \texttt{ at } p : \mu', \{\mathbf{put}(p)\} \tag{295}$$

By (290), (293) and the definition of Q_3 we then have

$$(R', S(TE_0) + \{f_0 \mapsto (\sigma_0, p_0)\}, E_0 + \{f_0 \mapsto v\}, s', VE_0, \varphi' \cup \varphi) \in Q_3 \tag{296}$$

Let $R_0' = R_0 + \{\rho \mapsto r\}$. From (291) and $\rho \notin \text{frv}(\varphi)$ we get

$$R_0' \text{ and } R' \text{ agree on } \varphi' \cup \varphi \tag{297}$$

and from (292) and (293) we get

$$R_0'(e_0'') = e_0' \tag{298}$$

From (295), (296), (297) and (298) and the definition of F, we get $q_1 \in F_1(Q)$ in this case as well.

Next, write q_2 in the form $(R', (\sigma', p), v, s', v', \varphi' \cup \varphi)$. Then there exist σ and S such that

$$S(\sigma) = \sigma' \tag{299}$$

$$\text{Consistent}(R, (\sigma, p), v, s, v') \text{ w.r.t. } \varphi \tag{300}$$

$$\rho \notin \text{frv}((\sigma, p), \varphi) \tag{301}$$

Assume $\mathbf{get}(p) \in \varphi' \cup \varphi$. Since by (301) we have $\rho \neq p$ we must have $\mathbf{get}(p) \in \varphi$. Then, by (300), we have $v' \in \text{Dom}(s)$ and r of $v' = R(p)$. Since $s \sqsubseteq s'$ we have $v' \in \text{Dom}(s)$ and

since $p \neq \rho$ we have r of $v' = R'(p)$. We now proceed by case analysis, still following the definition of F:

$\boxed{\sigma' \text{ is simple}}$ Then, by (299), σ is simple. Write σ in the form $\forall\alpha_1 \cdots \alpha_n \forall\epsilon_1 \cdots \epsilon_m.\tau$. Take any τ' with $\tau' \leq \sigma'$. Since $S(\sigma) = \sigma' \geq \tau'$ there exists a simple S' such that $S'(\tau) = \tau'$. Thus $S(\tau, p) = (\tau', p)$. Moreover, $\rho \notin \text{frv}((\tau, p), \varphi)$, by (301). From (300) and the definition of Consistent, we have Consistent$(R, (\tau, p), v, s, v')$ w.r.t. φ. Thus by the definition of Q_1, $(R', (\tau', p), v, s', v', \varphi' \cup \varphi) \in Q_1$. Since we can do this for every instance τ' of σ', we have $q_2 \in F_2(Q)$ as required.

$\boxed{\sigma' \text{ is compound}}$ Then σ is compound, so by (300) we have that v is a recursive closure $\langle x_0, e_0, E_0, f_0\rangle$ and $s(v') = \langle \rho'_1, \ldots, \rho'_k, x_0, e'_0, VE_0\rangle$, for some $\rho'_1, \ldots, \rho'_k$, e'_0 and VE_0. Furthermore, there exist TE_0, R_0, e''_0, $\rho_1, \ldots, \rho_k$, $\alpha_1, \ldots, \alpha_n$, $\epsilon_1, \ldots, \epsilon_m$ and τ such that

$$\text{Consistent}(R, TE_0 + \{f_0 \mapsto (\sigma, p)\}, E_0 + \{f_0 \mapsto v\}, s, VE_0) \text{ w.r.t. } \varphi \tag{302}$$

$$\sigma = \forall\rho_1 \cdots \rho_k \forall\alpha_1 \cdots \alpha_n \forall\epsilon_1 \cdots \epsilon_m.\tau \tag{303}$$

$$\text{No bound variable of } \sigma \text{ is free in } (TE_0, p) \tag{304}$$

$$TE_0 + \{f_0 \mapsto (\sigma, p)\} \vdash \lambda x_0.e_0 \Rightarrow \lambda x_0.e''_0 \texttt{ at } p : (\tau, p), \{\mathbf{put}(p)\} \tag{305}$$

$$R_0 \text{ and } R \text{ agree on } \varphi \tag{306}$$

$$R_0\langle \rho_1, \ldots, \rho_k, x_0, e''_0, VE_0\rangle = \langle \rho'_1, \ldots, \rho'_k, x_0, e'_0, VE_0\rangle \tag{307}$$

Since $s \sqsubseteq s'$, we have that $s'(v')$ is the same region closure $\langle \rho'_1, \ldots, \rho'_k, x_0, e'_0, VE_0\rangle$. As in two previous cases concerning closures, we can choose TE_0, R_0 and e''_0 such that, additionally,

$$\rho \notin \text{frv}(TE_0 + \{f_0 \mapsto (\sigma, p)\}, e''_0) \tag{308}$$

Thus from (302) we get

$$(R', S(TE_0) + \{f_0 \mapsto (S\sigma, p)\}, E_0 + \{f_0 \mapsto v\}, s', VE_0, \varphi' \cup \varphi) \in Q_3 \tag{309}$$

Let $S^+ = S + (\{\rho_1 \mapsto \rho''_1, \ldots, \rho_k \mapsto \rho''_k\}, \{\alpha_1 \mapsto \alpha''_1, \ldots, \alpha_n \mapsto \alpha''_n\}, \{\epsilon_1 \mapsto \epsilon''_1.\emptyset, \ldots, \epsilon_m \mapsto \epsilon''_m.\emptyset\})$, where alle the double-primed variables are distinct and new. By Lemma 3.1 on (305) we have

$$S'(TE_0) + \{f_0 \mapsto (S'\sigma, S'p)\} \vdash \lambda x_0.e'_0 \Rightarrow S'(\lambda x_0.e''_0 \texttt{ at } p) : (S'\tau, S'p), S'\{\mathbf{put}(p)\} \tag{310}$$

By (303) we have $S'(TE_0) = S(TE_0)$. Also, $S'(p) = S(p) = p$ and $S'(\sigma) = S(\sigma)$. Thus (310) reads

$$S(TE_0) + \{f_0 \mapsto (S\sigma, p)\} \vdash \lambda x_0.e'_0 \Rightarrow \lambda x_0.S'e''_0 \texttt{ at } p : (S'\tau, p), \{\mathbf{put}(p)\} \tag{311}$$

Because of the way we defined S', we have

$$S(\sigma) = \forall\rho''_1 \cdots \rho''_k \forall\alpha''_1 \cdots \alpha''_n \forall\epsilon''_1 \cdots \epsilon''_m.S'(\tau) \tag{312}$$

where none of the bound variables on the right-hand side occur free in $(S(TE_0), p)$. Finally, let $R'_0 = R_0 + \{\rho \mapsto r\}$. By (306) and $\rho \notin \mathrm{frv}(\varphi)$ we have

$$R'_0 \text{ and } R' \text{ agree on } \varphi' \cup \varphi \tag{313}$$

It remains to prove

$$R'_0\langle \rho''_1, \ldots, \rho''_k, x_0, S'e''_0, VE_0\rangle = \langle \rho'_1, \ldots, \rho'_k, x_0, e'_0, VE_0\rangle \tag{314}$$

But $\langle \rho''_1, \ldots, \rho''_k, x_0, S'e''_0, VE_0\rangle = \langle \rho_1, \ldots, \rho_k, x_0, e''_0, VE_0\rangle$ up to renaming of bound variables. Thus (314) follows from (307) and (308) as desired. By (309), (311), (312), (313), (314) and $S(\sigma) = \sigma'$ we have $q_2 \in F_2(Q)$ in this case as well.

Finally, write q_3 in the form $(R', TE', E, s', VE, \varphi' \cup \varphi)$. Then there exist TE and S such that $S(TE) = TE'$, Consistent(R, TE, E, s, VE) w.r.t. φ and $\rho \notin \mathrm{frv}(TE, \varphi)$. In particular, $\mathrm{Dom}\, TE = \mathrm{Dom}\, E = \mathrm{Dom}\, VE$. Thus $\mathrm{Dom}\, TE' = \mathrm{Dom}\, E = \mathrm{Dom}\, VE$. Let x be a variable in the domain of TE'. Then Consistent$(R, TE(x), E(x), s, VE(x))$ w.r.t. φ and $\rho \notin \mathrm{frv}(TE(x), \varphi)$. Thus $(R', TE'(x), E(x), s', VE(x), \varphi' \cup \varphi) \in Q_2$. Since this holds for all $x \in \mathrm{Dom}\, TE'$ we have $q_3 \in F_3(Q)$, as desired.

This proves that $q \in F(Q)$, as desired.