Unification and Polymorphism in Region Inference

Mads Tofte, Department of Computer Science, University of Copenhagen
Lars Birkedal, School of Computer Science, Carnegie Mellon University

Dedicated to Robin Milner on the occasion of his 60th birthday.

Abstract

Region Inference is a technique for inferring lifetimes of values in strict, higher-order programming languages such as Standard ML. The purpose of this paper is to show how ideas from Milner's polymorphic type discipline can serve as a basis for region inference, even in the presence of a limited form of polymorphic recursion.

1 Introduction

Most programming languages provide some facility for dynamic allocation and de-allocation of memory cells. The reason for this is that the number of cells which the program wishes to allocate and use at some point during the computation often is much larger than the number of cells which have to exist at any one point of the computation.

The stack discipline which originated with Algol 60[15] is a particularly elegant discipline for dynamic memory management. Restricted versions of the original stack discipline are used in many languages in current use, e.g., C and Pascal. In the stack discipline, every point of memory allocation is matched by a point of de-allocation, and these points are easy to identify from the program text. The evident connection between allocation and de-allocation makes it possible for programmers to reason about the lifetimes of storage cells. However, the stack discipline imposes rather severe restrictions on the programming language. For example, functions are not allowed to return lists or functions as results.

To overcome these limitations, many programming languages assume an additional memory area, the *heap*, which holds values whose final size or lifetime is not known when allocation of memory to hold the value takes place. In some languages, for example C, it is the responsibility of the programmer to allocate blocks of memory and to free them once they are no longer needed. While it is easy to allocate memory, it is in general extremely hard to know when it is safe to de-allocate memory. Intuitively, a memory cell can be reclaimed (it is "garbage"), if and only if its contents is not needed by the remainder of the computation. This is an undecidable property. To make matters worse, de-allocating memory too early can cause the program to crash, while de-allocating memory too late can cause "space leaks", i.e., programs that hold on to much more memory than is necessary. Moreover, unlike what is the case for the stack discipline, the programming language does not help the programmer in reasoning about lifetimes of storage cells.

The term "garbage collection" is traditionally used for a range of heap memory management techniques, including reference counting, copying collection and generational collection (see [25] for an excellent overview).[1] Common to all of these techniques is that there is a strict separation between the program which allocates memory, called the *mutator*, and the part of the runtime system, called the *garbage collector*, which manages recycling of memory. For some garbage collection schemes, it is hard for users to know when garbage collection takes place and how long it will take. Garbage collection was first used with Lisp; it is used both in implementations of functional and object oriented languages.

[1] This use of language is arguably misleading: the stack discipline also collects garbage and it does so very efficiently, by popping the stack.

Garbage collection techniques have developed very significantly over the past decade and the time spent by garbage collection is often modest (say less than 5% of the total running time). However, the strict separation between allocation and de-allocation means that there is no language support for reasoning about the lifetimes of storage cells, and thus it is in general very difficult for programmers to reason about how much memory their programs will use.

Region-based memory management[23, 1, 2] is yet a form of automatic management of dynamic allocation. Conceptually, the store consists of a stack of *regions*. A region can be thought of as a heap which can grow dynamically depending on how many values it needs to hold. Regions are allocated and de-allocated in a stack-like manner, i.e., a region is not de-allocated unless it is the topmost region on the region stack. All values are put into regions, including for example lists and function closures. The only mechanism for freeing memory is to pop the region stack, which is a constant time operation, for a suitable choice of concrete representation of regions.

Rather than forcing the programmer to introduce and eliminate regions, we have taken the approach to use an existing language, Standard ML[11], as the source language. The region scheme, including the algorithms presented in this paper, are implemented in the ML Kit with Regions [21] which compiles Standard ML Core Language programs to C and HP PA-RISC code. The ML Kit subjects source programs to a particular static analysis, *region inference*. The analysis decides where regions should be allocated and de-allocated. Region inference also decides, for each value-producing expression in the program, into which region the value will be put. Region inference handles recursive datatypes and higher-order functions. Moreover, region inference preserves a close connection between program structure and lifetimes of values, making it possible for humans to reason about lifetimes and making it easy for profiling tools to report actual memory usage in terms of regions which can be related to the program text. Finally, the memory management directives which region inference inserts during compilation each use only constant time and space at runtime. Thus programs can be run as they are written, without hidden extra costs of unbounded size (in contrast to what is the case for other existing ML implementations).

Some programs are ill-suited for region inference. An example of such a program is an interpreter for S, K and I combinators. Here the region inference will not be able to distinguish lifetimes of different terms and memory usage will be linear in running time. However, even programs which were originally thought to require garbage collection (for instance Knuth-Bendix completion) have been made to run in much less space using regions than using traditional garbage collection; the region scheme is not magic, but it can often be used to obtain programs that have time and memory performance that is better than those reported for systems that use traditional garbage collection techniques. This is as one might expect, at a superficial level at least, for while most garbage collection techniques are general techniques which treat all programs equally, region inference specialises memory management to the particular program that is being compiled.

The relevance of region inference to this Festschrift is that Milner's work on type inference and type checking[10] has provided many of the technical insights which underlie region inference. (Other origins of region inference are listed in Section 3.) In particular, the idea of using unification in type checking and the notions of polymorphic type schemes and generic instance are heavily exploited in work on regions. As it happens, region inference is not restricted to languages that use ML-style type polymorphism. But as we describe below, there are close connections between region- and effect polymorphism and the ability to extend the stack discipline to cover language constructs found in SML, specifically recursive datatypes and higher-order functions.

The rest of this paper is organised as follows. We first introduce the region-annotated types that underlie region inference, illustrating the basic ideas by means of a well-known program: the towers of Hanoi. We then introduce a technical notion, *consistency*, which is required for the statement and proof of a key lemma. Informally speaking, this lemma (Lemma 6) says that Robinson's result[17] concerning most general unifiers of terms generalises to the case where terms are types which are annotated with regions and effects, provided the annotation is done consistently. We define operations corresponding to Milner's notions of generic instance and generalisation in the setting of region-annotated types and introduce a particular class of type schemes which is general enough to allow a limited form of polymorphic recursion in regions and yet restricted enough that one can prove that region inference always terminates.

```
fun hanoi(n,from,to,other,acc) =
   if n=0 then (from,to)::acc
   else hanoi(n-1, from, other, to, (from, to) :: hanoi(n-1, other, to, from, acc))
val solution  = hanoi(20,"a","b","c",nil)
```

Figure 1: The Towers of Hanoi (source program).

2 Region-annotated programs

In this section we introduce the three kinds of region annotations which region inference inserts into source programs. Readers who are familiar with region inference may want to proceed directly to Section 3.

The Towers of Hanoi is the name of a game which involves one monk, three vertical pegs (A, B and C) an n discs of different sizes. The discs have holes in them, so that the monk can stack them on top of each other on the pegs. Initially, there are n discs on A, stacked in decreasing size (with the smallest disc at the top) while B and C contain no discs. The problem for the monk is to move the discs around between the pegs, one at the time, in such a way that at the end, C contains all the discs and A and B contain no discs. The moves have to be such that no disc is ever placed on top of a smaller disc.

The Standard ML program in Figure 1 solves the problem. $hanoi(n, from, to, other, acc)$ moves $n+1$ discs from the *from* peg to the *to* peg, using also the *other* peg, if necessary. Moves are accumulated onto the list *acc*.

Region inference translates this source program into the program shown in Figure 2. Every value-producing expression has been decorated with an annotation of the form "`at` ρ", where ρ is a *region variable*. The annotation indicates the region into which the value should be put. In line 18, for example, the constant 1 is put into region ρ_{26}.

The construct "`letregion` ρ `in` e `end`" binds ρ within e. At runtime, first a region is allocated and bound to ρ; then e is evaluated, perhaps using the region for storing and fetching values; finally, upon reaching "`end`", the region is de-allocated. In line 18, for example, ρ_{26} is only needed for the computation of $n-1$. Regions can also contain tuples (see ρ_{18} and ρ_{24}, for example) and closures (not illustrated by the example).

Notice that the region inference algorithm has given *hanoi* six additional formal region parameters: $\rho_6, \ldots, \rho_{11}$. These are enclosed in square brackets to distinguish them from parameters which the programmer writes. Correspondingly, the references to *hanoi* (lines 10, 17 and 32) each have six *actual region parameters*. The region parameters give the regions of the argument and result of the function. Result regions indicate where the function should put the result value, if it creates one. For example, ρ_9 is the region where the pairs which represent moves are put and ρ_7 holds the spine of the list of moves. ρ_6 is the region where the argument quintuple to *hanoi* is stored (we treat all tuples as boxed).

Notice that the three calls to *hanoi* have different actual region arguments. We refer to this feature as *region polymorphism*, a term that will be justified once we have introduced region-annotated types and type schemes. Note that even within the body of *hanoi*, the actual region arguments to *hanoi* differ from the formal region parameters. We refer to this feature of the system as *region-polymorphic recursion*; it is essential for obtaining good results.

By examining the region annotations one can determine that the maximal memory requirement for the computation will be $c + c_1 \times n + c_2 \times (2^{n+1} - 1)$ words, where n is the first argument to *hanoi* (here 20), c is a (small) constant, c_1 is the number of words it takes to represent the fixed size regions in lines 9 and 15 and c_2 is the number of words it takes to represent one move in the accumulating parameter. Here the third term clearly dominates. Indeed, for the ML Kit, c_2 is 4 (4 words = 16 bytes) and when running the program under the ML Kit one finds the expected maximal memory usage of approximately $c_2\text{words} \times 4\text{bytes/word} \times 2^{21} = 32$ Mb. The run takes 1.85 seconds of user time on an HP 9000s700 (C100). The corresponding numbers for the same

```
fun hanoi at ρ5[ρ6,ρ7,ρ8,ρ9,ρ10,ρ11](var6) =
      let val n  = #0 var6;  val from  = #1 var6;
         val to  = #2 var6; val other  = #3 var6;
         val acc  = #4 var6
      in  letregion ρ13
         in (case letregion ρ14 in (n = (0 at ρ14))at ρ13 end (*ρ14*)
           of true => let val v170  = ((from, to) at ρ9, acc) at ρ8 in :: at ρ7 v170 end
           |  false =>
              letregion ρ16, ρ18, ρ19 (* for region tuple, quintuple and n-1, respectively*)
               in hanoi[ρ18,ρ7,ρ8,ρ9,ρ10,ρ19] at ρ16
                 (letregion ρ20 in (n-1at ρ20) at ρ19 end (*ρ20*),
                  from, other, to,
                  let val v171  =
                        ((from, to) at ρ9,
                         letregion ρ22, ρ24, ρ25
                           (* for region tuple, quintuple and n-1, respectively *)
                         in hanoi[ρ24,ρ7,ρ8,ρ9,ρ10,ρ25] at ρ22
                           (letregion ρ26 in (n-1at ρ26) at ρ25 end (*ρ26*),
                            other, to, from, acc
                           ) at ρ24
                         end (*ρ22, ρ24, ρ25*)
                        ) at ρ8
                  in  :: at ρ7 v171
                  end
                 ) at ρ18
               end (*ρ16, ρ18, ρ19*)
           ) (*case*)
         end (*ρ13*)
       end ;
val solution  =
   letregion ρ27, ρ29, ρ30
   in hanoi[ρ29,ρ1,ρ2,ρ3,ρ4,ρ30] at ρ27
     (20 at ρ30, "a" at ρ4, "b" at ρ4, "c" at ρ4, nil at ρ1) at ρ29
   end (*ρ27, ρ29, ρ30*)
```

Figure 2: The region-annotated version of *hanoi*.

program running under Standard ML of New Jersey, version 93, on the same machine are 75 Mb and 28.36 seconds, respectively. This difference in running times is particularly noteworthy when one considers that the ML Kit boxes all tuples, including the tuples of region variables that are passed to region-polymorphic functions; that is what regions ρ_{16}, ρ_{22} and ρ_{27} are used for.

3 Related Work

The basic ideas of the region inference scheme are described in [23]. An extended version, including a proof that the region inference rules are sound, may be found in [24]. Other analysis which have been combined with region inference are described in [1, 2].

The emphasis of this paper is on using unification to constrain region variables and arrow effects. (Arrow effects decorate function arrows in the region type system with information of how the function uses regions.) In the case of arrow effects, the difficulty is that the unification is of sets rather than terms. Similar forms of unification have been studied in record typing[16] and in type systems for polymorphic references[9].

The region inference algorithm that is currently used in the ML Kit is described and proved correct in [20]. It is a syntax-directed algorithm, similar to Milner's algorithm W, and it relies on the results about unification proved in the present paper.

A different approach to region inference is to use constraints. Constraints have been used in previous work on ML type inference[5], subtyping [12, 4] and effect systems[19, 18, 14, 13]. We are currently exploring using constraints for region inference. The relative merits of the two approaches to region inference (syntax-directed region inference versus constraint generation and constraint solving) are not clear at the time of writing. In both cases, however, termination of the algorithms depends crucially on the limitations on polymorphic recursion which we introduce in the present paper.

4 Region Inference Rules

In order to give an overview of the inference problem we are addressing, this section contains the inference rules which guide our region inference algorithms. (Presenting a full algorithm would require more space than is available.) Concepts and notation which will be defined in the subsequent sections of the paper are marked by forward references.

For the purpose of stating invariants about the region algorithms, it is useful to keep careful track of the sets of region and effect variables that are used by the algorithm. Thus we introduce (in Section 6) a notion of a *basis*, $B = (Q, \Phi)$ where Q is a set of region variables and Φ is a set of *arrow effects*. Arrow effects decorate function arrows in derivations. An arrow effect takes the form $\epsilon.\varphi$, where ϵ is an effect variable and φ is an effect. Amongst the invariants which are essential for obtaining the result about principal unifiers (Section 8), is that whenever $\epsilon_1.\varphi_1$ and $\epsilon_2.\varphi_2$ are both members of Φ and $\epsilon_1 = \epsilon_2$ then $\varphi_1 = \varphi_2$. The invariants about sets of region variables and arrow effects that turn out to be important are collected in the definition of *consistent bases* (Section 6). Section 6 also defines the *disjoint union* of bases ($B \uplus B'$) found in the rules.

The inference rules allow one to infer statements of the form $B, TE \vdash e : \mu, \varphi$, read: in basis B and type environment TE, e has type with place μ and effect φ. Effects and types with places are defined in Section 5. Given a consistent basis B, only some types with places make sense. This is captured by a definition (in Section 5) of what it means for a type with place μ to be *consistent in a basis,* B, written $B \vdash \mu$.

A *type environment*, TE, is a finite map from program variables to pairs of the form (σ, ρ), where σ is a region-annotated type scheme (Section 9) and ρ is a region variable.

The expressions that appear in the rules are called "target expressions", since they are output from region inference. The terms contain enough type, region and effect information that the entire proof of the conclusion $B, TE \vdash e : \mu, \varphi$ is fully determined by B, TE and e. Some of this information is not necessary at runtime. Therefore, in examples we often abbreviate `letregion` B `in` e `end` to `letregion` $\rho_1, \ldots, \rho_k$ `in` e `end`, where $\{\rho_1, \ldots, \rho_k\} = Q$ *of* B (Section 6). Similarly, we often abbreviate

$$\texttt{letrec}\ f : (\sigma, \rho_0)(x)\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end}$$

to `letrec` $f[\rho_1, \ldots, \rho_k]$ `at` ρ_0 `(x) =` e_1 `in` e_2 `end`, where $\rho_1, \ldots, \rho_k$ are the bound region variables of σ.

In target expressions, every binding occurrence of a variable is decorated by a region type scheme (Section 9). Within the scope of the binding, each non-binding occurrence of the variable is annotated by its own *instantiation list* (Section 11) which defines an instance of the type scheme particular to that occurrence. Given a consistent basis B, only some instantiation lists are meaningful; Section 11 defines what it means for an instantiation list il to be *consistent in a basis,* B, written $B \vdash il$. In examples (and at runtime) we omit the first and third component of instantiation list triples, so that x_{il} becomes simply $x[\rho_1, \ldots, \rho_k]$, where $(\_, [\rho_1, \ldots, \rho_k], \_) = il$.

Given a consistent basis B, only some region-annotated type schemes make sense: Section 10 defines what it means for a region-annotated type scheme σ to be *consistent in a basis,* B, written $B \vdash \sigma$. Poitwise extension gives a relation $B \vdash TE$.

In the following, we write $B \vdash (TE, il, \mu)$ as a shorthand for $B \vdash TE$, $B \vdash il$ and $B \vdash \mu$. It turns out that $B, TE \vdash e : \mu, \varphi$ implies that B is consistent and $B \vdash (TE, \mu, \varphi)$. In rule (6) we use the function *Observe* defined by $\mathit{Observe}(B, \varphi_1) = \varphi_1 \cap \mathit{fv}(B)$.

In the rules, S denotes a *substitution*, as defined in Section 7.

$$\frac{\begin{array}{c} B \vdash (TE, il, \mu) \qquad TE(x) = (\sigma, \rho) \qquad \mu = (\tau, \rho) \\ \sigma = \forall \alpha_1 \cdots \alpha_n \epsilon_1 \cdots \epsilon_m . \tau_0 \qquad il = ([\tau_1, \ldots, \tau_n], [\,], [\epsilon'_1 . \varphi'_1, \ldots, \epsilon'_m . \varphi'_m]) \\ S = (\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}, \{\}, \{\epsilon_1 \mapsto \epsilon'_1 . \varphi'_1, \ldots, \epsilon_m \mapsto \epsilon'_m . \varphi'_m\}) \\ \tau = S(\tau_0) \end{array}}{B, TE \vdash x_{il} : \mu, \emptyset} \tag{1}$$

$$\frac{\begin{array}{c} B \vdash (TE, il, \mu) \qquad TE(x) = (\sigma, \rho) \qquad \mu = (\tau, \rho') \\ \sigma = \forall \alpha_1 \cdots \alpha_n \rho_1 \cdots \rho_k \epsilon_1 \cdots \epsilon_m . \tau_0 \qquad il = ([\tau_1, \ldots, \tau_n], [\rho'_1, \ldots, \rho'_k], [\epsilon'_1 . \varphi'_1, \ldots, \epsilon'_m . \varphi'_m]) \\ S = (\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}, \{\rho_1 \mapsto \rho'_1, \ldots, \rho_k \mapsto \rho'_k\}, \{\epsilon_1 \mapsto \epsilon'_1 . \varphi'_1, \ldots, \epsilon_m \mapsto \epsilon'_m . \varphi'_m\}) \\ \tau = S(\tau_0) \end{array}}{B, TE \vdash x_{il} \; \mathtt{at} \; \rho' : \mu : \{\rho, \rho'\}} \tag{2}$$

$$\frac{B \vdash TE \qquad B, TE + \{x \mapsto \mu_x\} \vdash e_1 : \mu_1, \varphi_1 \quad \varphi_0 \supseteq \varphi_1 \quad (\{\rho\}, \{\epsilon . \varphi_0\}) \subseteq B}{B, TE \vdash \lambda x : \mu_x . e_1 : (\mu_x \xrightarrow{\epsilon . \varphi_0} \mu_1, \rho), \{\rho\}} \tag{3}$$

$$\frac{B, TE \vdash e_1 : (\mu' \xrightarrow{\epsilon_0 . \varphi_0} \mu, \rho_0), \varphi_1 \qquad B, TE \vdash e_2 : \mu', \varphi_2}{B, TE \vdash e_1 \; e_2 : \mu, \varphi_1 \cup \varphi_2 \cup \{\epsilon_0, \rho_0\} \cup \varphi_0} \tag{4}$$

$$\frac{\begin{array}{c} B \vdash (TE, \varphi_1) \quad \sigma = \forall \alpha_1 \cdots \alpha_n \rho_1 \cdots \rho_k \epsilon_1 \cdots \epsilon_m . \tau_0 \quad \hat{\sigma} = \forall \rho_1 \cdots \rho_k \epsilon_1 \cdots \epsilon_m . \tau_0 \\ B_1 = \mathit{bound}(\sigma) \qquad B \uplus B_1 \text{ exists} \qquad \tau_0 = \mu_x \xrightarrow{\epsilon_0 . \varphi_0} \mu_1 \qquad \mathit{btv}(\sigma) \cap \mathit{ftv}(TE) = \emptyset \\ B \uplus B_1, TE + \{f \mapsto (\hat{\sigma}, \rho_0)\} \vdash \lambda \, x : \mu_x . e_1 : (\tau_0, \rho_0), \varphi_1 \\ B, TE + \{f \mapsto (\sigma, \rho_0)\} \vdash e_2 : \mu, \varphi_2 \end{array}}{B, TE \vdash \mathtt{letrec} \; f : (\sigma, \rho_0)(x) \mathtt{=} \; e_1 \; \mathtt{in} \; e_2 \; \mathtt{end} : \mu, \varphi_1 \cup \varphi_2} \tag{5}$$

$$\frac{B \uplus B_1 \text{ exists} \qquad B \uplus B_1, TE \vdash e_1 : \mu : \varphi_1 \qquad B \vdash (TE, \mu) \qquad \varphi = \mathit{Observe}(B, \varphi_1)}{B, TE \vdash \mathtt{letregion} \; B_1 \; \mathtt{in} \; e_1 : \mu : \varphi_1 \; \mathtt{end} : \mu : \varphi} \tag{6}$$

5 Types and Effects

In this section we first motivate the region-annotated types and type schemes that are used in region inference and then define them formally.

5.1 Region Variables and Effects

The region scheme annotates every type constructor with a region variable. Also, type schemes can universally quantify region variables and (as in Milner's type discipline) type variables. The type scheme of *hanoi* of Figure 2 is approximately the following:

$$\begin{array}{l} \forall \rho_6 \rho_7 \rho_8 \rho_9 \rho_{10} \rho_{11} . \big((\mathit{int}, \rho_{11}) * (\alpha, \rho_{10}) * (\alpha, \rho_{10}) * (\alpha, \rho_{10}) * (((\alpha, \rho_{10}) * (\alpha, \rho_{10}), \rho_9) \mathit{list}_{\langle \rho_8 \rangle}, \rho_7), \\ \qquad \rho_6 \big) \xrightarrow{\epsilon} (((\alpha, \rho_{10}) * (\alpha, \rho_{10}), \rho_9) \mathit{list}_{\langle \rho_8 \rangle}, \rho_7) \\ \qquad \text{where } \epsilon = \{\rho_6, \rho_7, \rho_8, \rho_9, \rho_{11}\} \end{array}$$

Here α is paired with ρ_{10}, *list* with ρ_7 and the tuple constructor $(\_ * \_ * \_ * \_ * \_)$ is paired with ρ_6. The datatype constructor *list* has an additional, so-called *auxiliary* region variable, here ρ_8. This is the region where the pairs to which cons (`::`) is applied are put. The set ϵ which appears on the function arrow is an *effect*; it contains all the regions which the evaluation of the function body might access. *hanoi* reads from region ρ_6 (when it extracts the components of its argument quintuple), it reads n from region ρ_{11} and it puts values into regions ρ_7, ρ_8 and ρ_9. The effect reveals that *hanoi* does not access ρ_{10} at all.

In [23] there is a distinction between put and get effects; for example, the above effect would be written $\{\mathbf{get}(\rho_6), \mathbf{put}(\rho_7), \mathbf{put}(\rho_8), \mathbf{put}(\rho_9), \mathbf{get}(\rho_{11})\}$. This distinction is useful for other analyses and optimisations which can be combined with region inference[2]; however, for the purpose of this paper, the distinction between **put** and **get** is not important and is therefore omitted.

5.2 Effect Variables and Arrow Effects

In the type scheme for *hanoi* we use ϵ and "where" as meta-notation to make the type scheme easier to read. A crucial next step, invented in work on effect inference[18], is to make effect variables part of the language of types, on a par with type variables and region variables. Moreover, we represent "where $\epsilon = \{\rho_6, \rho_7, \rho_8, \rho_9, \rho_{11}\}$" by a formal object "$\epsilon.\{\rho_6, \rho_7, \rho_8, \rho_9, \rho_{11}\}$" which is called an *arrow effect*, because arrow effects appear on function arrows. Using an effect variable and an arrow effect, the type scheme for *hanoi* becomes:

$$\begin{aligned}\sigma_h = \forall \alpha\rho_6\rho_7\rho_8\rho_9\rho_{10}\rho_{11}\epsilon_{12}.\big(&(\mathit{int}, \rho_{11}) * (\alpha, \rho_{10}) * (\alpha, \rho_{10}) * (\alpha, \rho_{10}) * \\ &(((\alpha, \rho_{10}) * (\alpha, \rho_{10}), \rho_9)\mathit{list}_{\langle\rho_8\rangle}, \rho_7), \rho_6\big) \\ &\xrightarrow{\epsilon_{12}.\{\rho_6, \rho_7, \rho_8, \rho_9, \rho_{11}\}} (((\alpha, \rho_{10}) * (\alpha, \rho_{10}), \rho_9)\mathit{list}_{\langle\rho_8\rangle}, \rho_7)\end{aligned}$$

As we shall see in Section 11 instantiation of type schemes allows universally quantified effects to increase; so one can equally think of $\epsilon.\varphi$ as meaning "ϵ, where $\epsilon \supseteq \varphi$". Note that the scope of the effect variable is now given by $\forall$, which can bind type, region and effect variables. Capture-avoiding renaming of bound variables is permitted in the usual way.

In a first-order programming language, one could do region inference without effect variables. But for a higher-order language, effect variables are essential for representing the effects of (unknown) functions. To illustrate this, assume we continue our example program with the declarations:

> `fun` *app f x* = *f*(*x*)
> `fun` *run n* = *app hanoi* (*n*, "*a*", "*b*", "*c*", *nil*)

The function *app* has the type scheme

$$\sigma_{app} = \forall \alpha_1\alpha_2\rho_1\rho_2\rho_3\epsilon_1\epsilon_2\epsilon_3.\big((\alpha_1, \rho_1) \xrightarrow{\epsilon_1.\emptyset} (\alpha_2, \rho_2), \rho_3\big) \xrightarrow{\epsilon_2.\{\rho_4\}} \big((\alpha_1, \rho_1) \xrightarrow{\epsilon_3.\{\rho_3, \epsilon_1\}} (\alpha_2, \rho_2), \rho_4\big)$$

The arrow effect $\epsilon_3.\{\rho_3, \epsilon_1\}$ is justified as follows: ρ_3 has to be in the effect since the evaluation of $f(x)$ involves accessing ρ_3, the region holding the function f. Also, the effect of the expression $f(x)$ has to be at least the effect of the function f, irrespective of what actual arguments we pass in for f; this is represented by letting ϵ_1 be in the arrow effect. In other words, *app* has an "effect dependent" type: the effect of *app f* depends on the effect of f.

To see how this dependency comes into use, consider the application *app hanoi* in the declaration of *run*. Let α', ρ'_6, ρ'_7, ρ'_8, ρ'_9, ρ'_{10}, ρ'_{11} and ϵ'_{12} be fresh variables. We can give the occurrence of *hanoi* in the declaration of *run* the following instance of σ_h:

$$\begin{aligned}\tau_h = &\big((\mathit{int}, \rho'_{11}) * (\alpha', \rho'_{10}) * (\alpha', \rho'_{10}) * (\alpha', \rho'_{10}) * (((\alpha', \rho'_{10}) * (\alpha', \rho'_{10}), \rho'_9)\mathit{list}_{\langle\rho'_8\rangle}, \rho'_7), \rho'_6\big) \\ &\xrightarrow{\epsilon'_{12}.\{\rho'_6, \rho'_7, \rho'_8, \rho'_9, \rho'_{11}\}} (((\alpha', \rho'_{10}) * (\alpha', \rho'_{10}), \rho'_9)\mathit{list}_{\langle\rho'_8\rangle}, \rho'_7)\end{aligned}$$

Similarly, the following type is an instance of σ_{app}:

$$\tau_0 = (\tau_h, \rho_{100}) \xrightarrow{\epsilon_{101}.\varphi'} (\tau_h, \rho_{102})$$

where ρ_{100}, ϵ_{101} and ρ_{102} are fresh (φ' will be determined shortly). Here we have instantiated the arrow effect $\epsilon_1.\emptyset$ of σ_{app} to the arrow effect $\epsilon'_{12}.\{\rho'_6, \rho'_7, \rho'_8, \rho'_9, \rho'_{11}\}$ of τ_h. But when we then instantiate the arrow effect $\epsilon_3.\{\rho_3, \epsilon_1\}$ from σ_{app}, the dependency of ϵ_3 upon ϵ_1 implies that the effect to which ϵ_3 is instantiated must contain all that ϵ'_{12} stands for, i.e., the effect $\{\rho'_6, \rho'_7, \rho'_8, \rho'_9, \rho'_{11}\}$. In fact, the definition of generic instance presented in Section 11 will take φ' to be $\{\rho_{100}, \epsilon'_{12}, \rho'_6, \rho'_7, \rho'_8, \rho'_9, \rho'_{11}\}$, where the presence of ϵ'_{12} keeps track of the fact that the result of evaluating *app hanoi*, when applied to $(n, "a", "b", "c", nil)$, will call *hanoi*.

To sum up, effect variables are useful for tracking function applications in functional languages. Moreover, using effect polymorphism to represent function dependencies has two advantages:

1. The effect dependencies of *app* can be determined from its declaration alone, without looking at how the function is subsequently used;

2. In the scope of the declaration of *app*, if there are two applications, *app g* and *app h*, say, then the arrow effects of g and h need not be the same;

Both advantages are inherited from Milner-style polymorphism.

5.3 Definitions

We assume a denumerably infinite set TyVar of *type variables*. We use α to range over type variables. Next, *ML types*, τ^{ML}, and *ML type schemes*, σ^{ML}, are given by the grammar:

$$\begin{array}{lll} \tau^{ML} & ::= \texttt{int} \mid \alpha \mid \tau^{ML} \to \tau^{ML} & \text{ML type} \\ \sigma^{ML} & ::= \tau^{ML} \mid \forall\alpha.\sigma^{ML} & \text{ML type scheme} \end{array}$$

For the purpose of region inference, we assume two additional denumerably infinite sets:

$$\begin{array}{ll} \rho \in \text{RegVar} & \text{region variables} \\ \epsilon \in \text{EffVar} & \text{effect variables} \end{array}$$

TyVar, RegVar and EffVar are assumed to be pairwise disjoint. An *atomic effect* is a region variable or an effect variable; an *effect* is a finite set of atomic effects:

$$\begin{array}{rll} \eta \in \text{AtEff} = \text{RegVar} \cup \text{EffVar} & & \text{atomic effect} \\ \varphi \text{ or } \{\eta_1, \ldots, \eta_k\} \in \text{Effect} = \mathit{Fin}(\text{AtEff}) & & \text{effect} \\ \epsilon.\varphi \in \text{ArrEff} = \text{EffVar} \times \text{Effect} & & \text{arrow effect} \end{array}$$

Annotated types are given by:

$$\begin{array}{lll} \tau & ::= \texttt{int} \mid \alpha \mid \mu \xrightarrow{\epsilon.\varphi} \mu & \text{annotated type} \\ \mu & ::= (\tau, \rho) & \text{annotated type with place} \end{array}$$

The set of annotated types is denoted Type and the set of annotated types with places is denoted TypeWithPlace; "annotated type" is abbreviated to "type" when confusion with "ML type" is unlikely. In a function type $\mu \xrightarrow{\epsilon.\varphi} \mu'$ the object $\epsilon.\varphi$ is called an *arrow effect*. Formally, an arrow effect is a pair of an effect variable and an effect; we refer to ϵ and φ as the *handle* and the *latent effect*, respectively.

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

We refer to the above semantic objects (i.e., type variables, region variables, effect variables, atomic effects, effects, arrow effects, types and types with places) collectively as *basic* semantic objects and use o to range over these. Erasure of region and effect information from annotated types yields ML types; it is defined recursively by:

$$\text{ML}(\alpha) = \alpha; \quad \text{ML}(\texttt{int}) = \texttt{int}; \quad \text{ML}(\tau, \rho) = \text{ML}(\tau)$$

$$\text{ML}(\mu \xrightarrow{\epsilon.\varphi} \mu') = \text{ML}(\mu) \to \text{ML}(\mu')$$

Let τ or μ be a type or a type with place. The *arrow effects of τ (or μ)*, written $\mathit{arreff}(\tau)$ (or $\mathit{arreff}(\mu)$) is the set of arrow effects defined by

$$\mathit{arreff}(\alpha) = \emptyset; \quad \mathit{arreff}(\mathtt{int}) = \emptyset; \quad \mathit{arreff}(\tau, \rho) = \mathit{arreff}(\tau)$$

$$\mathit{arreff}(\mu \xrightarrow{\epsilon.\varphi} \mu') = \{\epsilon.\varphi\} \cup \mathit{arreff}(\mu) \cup \mathit{arreff}(\mu')$$

The set of type variables, effect variables and region variables that occur free in a semantic object A will be denoted $\mathit{ftv}(A)$, $\mathit{fev}(A)$ and $\mathit{frv}(A)$, respectively. For basic semantic objects, these functions are defined by:

$$\mathit{ftv}(\mathtt{int}) = \emptyset \quad \mathit{ftv}(\alpha) = \{\alpha\} \quad \mathit{ftv}(\mu \xrightarrow{\epsilon.\varphi} \mu') = \mathit{ftv}(\mu) \cup \mathit{ftv}(\mu') \quad \mathit{ftv}(\tau, \rho) = \mathit{ftv}(\tau)$$

$$\mathit{frv}(\rho) = \{\rho\} \qquad \mathit{frv}(\tau, \rho) = \{\rho\} \cup \mathit{frv}(\tau) \qquad \mathit{frv}(\mathtt{int}) = \mathit{frv}(\alpha) = \emptyset$$

$$\mathit{frv}(\mu \xrightarrow{\epsilon.\varphi} \mu') = \mathit{frv}(\mu) \cup \mathit{frv}(\epsilon.\varphi) \cup \mathit{frv}(\mu') \qquad \mathit{frv}(\epsilon.\varphi) = \varphi \cap \mathrm{RegVar}$$

$$\mathit{fev}(\rho) = \emptyset \qquad \mathit{fev}(\tau, \rho) = \mathit{fev}(\tau) \qquad \mathit{fev}(\mathtt{int}) = \mathit{fev}(\alpha) = \emptyset$$

$$\mathit{fev}(\mu \xrightarrow{\epsilon.\varphi} \mu') = \mathit{fev}(\mu) \cup \mathit{fev}(\epsilon.\varphi) \cup \mathit{fev}(\mu') \qquad \mathit{fev}(\epsilon.\varphi) = \{\epsilon\} \cup (\varphi \cap \mathrm{EffVar})$$

We denote the set of free variables of A (of either kind) by $\mathit{fv}(A)$, i.e., $\mathit{fv}(A) = \mathit{frv}(A) \cup \mathit{ftv}(A) \cup \mathit{fev}(A)$.

A *finite map* is a function whose domain is finite. When A and B are sets, 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 $\mathit{Rng}(f)$, respectively; the restriction of f to a set, A, is written $f \downarrow A$. A finite map is often written explicitly in the form $\{a_1 \mapsto b_1, \ldots, a_k \mapsto b_k\}$, $k \geq 0$; in particular, the empty map is $\{\}$. For every $f : A \xrightarrow{\text{fin}} B$ and $g : B \to C$ the finite map $g \bullet f : A \xrightarrow{\text{fin}} C$ is defined by $\mathrm{Dom}(g \bullet f) = \mathrm{Dom}(f)$ and $(g \bullet f)(x) = g(f(x))$.

6 Consistent Bases

To obtain eager recycling of memory, regions should be kept distinct, unless they are forced to be equal. Although not all the concepts have been formally introduced yet, the reader might want to take a sneak preview of the region inference rules in Section 4. These rules specify which region annotations are permitted in a skeletal language. The region inference algorithm (not presented) will keep regions distinct, unless they are required by the rules to be equal. The only rule which forces regions to be equal is rule (4) which concerns function application. Since types are decorated with regions (and arrow effects) the two occurrences of μ' in the premises of the rule demands that function and argument agree not just on types but also on regions and effects. In order to satisfy the premises of the inference rule, the region inference algorithm needs to be able to unify types which contain region and effect information.

Hindley[6] and Milner[10] discovered that Robinson's unification algorithm can be used for unifying types. We assume that type inference has already been carried out by the time region inference is performed. Thus the problem we are faced with is to unify region-annotated types which have the same underlying ML types but which may have different region and effect annotations. This problem would be very simple, were it not for the fact that types can contain effects, which are sets. It is not the case that if a set unification problem has a solution (a substitution which when applied to the two sets makes them equal) then it has a most general solution.

However, the use of arrow effects suggests an alternative to using set unification. Assume we are given two arrow effects $\epsilon_1.\varphi_1$ and $\epsilon_2.\varphi_2$ which we want to "unify". (For simplicity, assume $\{\epsilon_1, \epsilon_2\} \cap (\varphi_1 \cup \varphi_2) = \emptyset$.) We now simply select one of the effect variables, say ϵ_1, and let it stand for $\varphi_1 \cup \varphi_2$:

$$\begin{tikzcd} \epsilon_1.\varphi_1 \arrow[dr, "S"] & & \epsilon_2.\varphi_2 \arrow[dl, "S"'] & \text{Before unification} \\ & \epsilon_1.(\varphi_1 \cup \varphi_2) & & \text{After unification} \end{tikzcd}$$

In the picture, S is a substitution. (Substitutions will be defined in Section 7.)

Another way of thinking of $\epsilon_1.\varphi_1$ and $\epsilon_2.\varphi_2$ is as two constraints $\epsilon_1 \supseteq \varphi_1$ and $\epsilon_2 \supseteq \varphi_2$[18]; then the "unification" of the two arrow effect corresponds to replacing ϵ_2 by ϵ_1 everywhere and replacing the two constraints by a single constraint $\epsilon_1 \supseteq (\varphi_1 \cup \varphi_2)$.

Although the region inference rules (1) – (6) can be expressed using constraints rather than arrow effects along the lines of [18], constraint sets complicate the definitions of type schemes, generic instance, generalisation and the *Observe* operation given later in this paper. The best approach seems to be to use arrow effects for presenting the region inference system and then either also use arrow effects in the algorithms (as we do in this paper) or relate the region rules presented in this paper to a separate inference system which uses constraints that are solved separately. We are currently also exploring the latter possibility.

It is convenient to be able to see what effect a given effect variable in a given type denotes without reference to a constraint set. Therefore, within the scope of a binding occurrence of an effect variable ϵ, we want ϵ to be the handle of one and only one effect. There are other natural well-formedness conditions which we expect to hold. For example, in a type scheme of the form

$$\forall \epsilon_1 \epsilon_2.((\mu_1 \xrightarrow{\epsilon_1.\varphi_1} \mu_2), \rho_1) \xrightarrow{\epsilon_2.\varphi_2} \mu_3$$

if we have $\epsilon_1 \in \varphi_2$ then we should also have $\varphi_1 \subseteq \varphi_2$. These invariants are formalised by the notion of effect consistency defined below. As we shall see later, types which are decorated with arrow effects taken from an effect consistent set have most general unifiers, in a sense which will be made precise below.

Definition 1 *A set Φ of arrow effects is* effect consistent*, written $\vdash \Phi$, if*

1. *Φ is* functional*: for all $(\epsilon_1.\varphi_1) \in \Phi$ and $(\epsilon_2.\varphi_2) \in \Phi$ if $\epsilon_1 = \epsilon_2$ then $\varphi_1 = \varphi_2$;*
2. *Φ is* closed*: for all $(\epsilon.\varphi) \in \Phi$ and $\epsilon' \in \varphi$ there exists φ' such that $(\epsilon'.\varphi') \in \Phi$;*
3. *Φ is* transitive*: for all $(\epsilon_1.\varphi_1) \in \Phi$ and $(\epsilon_2.\varphi_2) \in \Phi$ if $\epsilon_2 \in \varphi_1$ then $\varphi_2 \subseteq \varphi_1$;* □

For any functional Φ, the *effect map of* Φ, written $\hat{\Phi}$, is the partial map from effect variables to effects defined by $\hat{\Phi}(\epsilon) = \varphi$, if $\epsilon.\varphi \in \Phi$. We define $frv(\Phi) = \cup\{frv(\epsilon.\varphi) \mid \epsilon.\varphi \in \Phi\}$ and $fev(\Phi) = \cup\{fev(\epsilon.\varphi) \mid \epsilon.\varphi \in \Phi\}$.

A *basis* is a pair $B = (Q, \Phi)$, where Q is a set of region variables and Φ is a set of arrow effects. The set of bases is called Basis. We say that B is *consistent*, written $\vdash B$, if $Q \supseteq frv(\Phi)$ and $\vdash \Phi$. We define $frv(B) = Q \cup frv(\Phi)$ and $fev(B) = fev(\Phi)$. Notice that if $B = (Q, \Phi)$ is consistent then $frv(B) = Q$ since $Q \supseteq frv(\Phi)$. Moreover, if $B = (Q, \Phi)$ is consistent then $fev(B) = \mathrm{Dom}(\hat{\Phi})$, since Φ is closed. The *domain* of a consistent basis $B = (Q, \Phi)$, written $\mathrm{Dom}(B)$, is the set $Q \cup fev(\Phi)$. A basis $B = (Q, \Phi)$ is *finite* if Q and Φ are both finite sets. When B is a basis we write Q *of* B and Φ *of* B for the first and second component of B, respectively.

Example 5.1 The basis $B_1 = (\{\rho_1\}, \{\epsilon_1.\emptyset, \epsilon_1.\{\rho_1\}\})$ is not consistent: ϵ_1 cannot both denote the empty effect and the effect $\{\rho_1\}$. Formally, B_1 is not functional. However, $B_2 = (\{\rho_1\}, \{\epsilon_1.\emptyset, \epsilon_2.\{\rho_1\}\})$ is consistent. □

Example 5.2 The basis $B_3 = (\{\rho_1, \rho_2\}, \{\epsilon_1.\{\rho_1\}, \epsilon_2.\{\epsilon_1, \rho_2\}\})$ is not consistent: the effect which ϵ_2 denotes has ϵ_1 as a member but is not a superset of the set which ϵ_1 denotes. Formally: B_3 is not transitive. However, $B_4 = (\{\rho_1, \rho_2\}, \{\epsilon_1.\{\rho_1\}, \epsilon_2.\{\epsilon_1, \rho_1, \rho_2\}\}$ is consistent. □

Let o range over basic semantic objects and let $B = (Q, \Phi)$ be a basis. We say that o is *consistent in* B, if the sentence $B \vdash o$ can be inferred from the following rules:

$$\frac{\vdash B \quad \rho \in Q \textit{ of } B}{B \vdash \rho} \qquad \frac{\vdash B \quad \epsilon.\varphi \in \Phi \textit{ of } B}{B \vdash \epsilon.\varphi} \qquad \frac{\vdash B \quad \eta \in \textit{fv}(B)}{B \vdash \eta}$$

$$\frac{\vdash B}{B \vdash \texttt{int}} \qquad \frac{\vdash B}{B \vdash \alpha} \qquad \frac{B \vdash \tau \qquad B \vdash \rho}{B \vdash (\tau, \rho)} \qquad \frac{B \vdash \mu_1 \quad B \vdash \epsilon.\varphi \quad B \vdash \mu_2}{B \vdash \mu_1 \xrightarrow{\epsilon.\varphi} \mu_2}$$

$$\frac{\epsilon \notin \textit{fev}(\Phi) \cup \varphi \quad \vdash (Q, \Phi \cup \{\epsilon.\varphi\})}{(Q, \Phi) \vdash \varphi}$$

All these rules, except the last one, should be straightforward: they simply state that a semantic object is consistent in a basis $B = (Q, \Phi)$ only if (a) the basis is consistent by itself and (b) all the region and effect variables appearing in o are free in B and (c) if an arrow effect $\epsilon.\varphi$ occurs in o then φ is uniquely determined by the equation $\varphi = \hat{\Phi}(\epsilon)$. The last inference rule says that φ is consistent in B if, imagining that we pick a fresh effect variable ϵ', the extended basis $B \cup (\emptyset, \epsilon'.\varphi)$ would become consistent. Alternatively, this rule can be expressed as the conjunction of (a) $B = (Q, \Phi)$ is consistent, (b) $\varphi \subseteq \textit{fv}(B)$ and (c) for all $\epsilon \in \varphi$ one has $\hat{\Phi}(\epsilon) \subseteq \varphi$.

Example 5.3 Let $\mu = (\texttt{int}, \rho_1)$. The types $\mu \xrightarrow{\epsilon_1.\emptyset} \mu$ and $\mu \xrightarrow{\epsilon_1.\{\rho_1\}} \mu$ cannot be consistent in the same basis. However, $\mu \xrightarrow{\epsilon_1.\emptyset} \mu$ and $\mu \xrightarrow{\epsilon_2.\{\rho_1\}} \mu$ are both consistent in the basis B_2 from Example 5.1. □

Example 5.4 Consider the basis B_4 from Example 5.2. The effect $\{\rho_1\}$ is consistent in B_4; so is $\{\epsilon_1, \rho_1, \rho_2\}$, but not $\{\epsilon_2, \rho_1\}$. □

As usual, we often write "$B \vdash o$" to mean "there exists a proof tree with conclusion $B \vdash o$." It is easy to see that $B \vdash o$ implies that B is consistent and that $\textit{fv}(o) \subseteq \text{Dom}(B)$.

Let $B_1 = (Q_1, \Phi_1)$ and $B_2 = (Q_2, \Phi_2)$ be bases. We define the *union* of B_1 and B_2, written $B_1 \cup B_2$ to be $(Q_1 \cup Q_2, \Phi_1 \cup \Phi_2)$. We define the *disjoint union* of B_1 and B_2, written $B_1 \uplus B_2$, to be $B_1 \cup B_2$ provided $B_1 \cup B_2$ is consistent and $\text{Dom}(B_1) \cap \text{Dom}(B_1) = \emptyset$, and undefined otherwise. Note that $B_1 \uplus B_2$ can be defined even in cases where B_1 and B_2 are not consistent by themselves. Typically, this concept is used for expressing disjointness of bound variables, for example in rule (5), where B is consistent and $B \uplus B_1$ is consistent, but B_1 not necessarily is consistent, since it can contain occurrences of region and effect variables that are in $\text{Dom}(B)$.

7 Substitution

Suppose for example that we want to unify the two types $\tau_1 = (\texttt{int}, \rho_1) \xrightarrow{\epsilon_1.\{\rho_1, \rho_2\}} (\texttt{int}, \rho_2)$ and $\tau_2 = (\texttt{int}, \rho_3) \xrightarrow{\epsilon_2.\emptyset} (\texttt{int}, \rho_3)$. These two types could arise during region inference, for example as the type of the `then` branch and the `else` branch, respectively, of the conditional:

```
if true then (fn x => x+1) else (fn y => y)
```

From the region variables that decorate `int` in τ_1 and τ_2 we see that a unifier for τ_1 and τ_2 has to map ρ_1, ρ_2 and ρ_3 to the same region variable, say ρ_1. But a unifier for τ_1 and τ_2 must also map the arrow effects in τ_1 and τ_2 to the same arrow effect, for example $\epsilon_1.\{\rho_1\}$. But how can a substitution map $\epsilon_2.\emptyset$ to $\epsilon_1.\{\rho_1\}$? Certainly it cannot, if substitutions simply map effect variables to effect variables. Letting substitutions map effect variables to effects and defining $S(\epsilon.\varphi)$ to be $S(\epsilon).S(\varphi)$ is not an option either for $S(\epsilon)$ would not be an effect variable and $S(\epsilon).S(\varphi)$ therefore not an arrow effect. The solution is to let substitutions map effect variables to arrow effects. For example, we allow $S_0^e = \{\epsilon_1 \mapsto \epsilon_1.\{\rho_1\}, \epsilon_2 \mapsto \epsilon_1.\{\rho_1\}\}$ as a substitution. The definition of substitution given below is such that $S_0^e(\epsilon_1.\{\rho_1\}) = \epsilon_1.\{\rho_1\} = S_0^e(\epsilon_2.\emptyset)$, i.e, that S_0^e is a unifier for $\epsilon_1.\{\rho_1\}$ and $\epsilon_2.\emptyset$.

Formally, a *type substitution* is a map from type variables to types; a *region substitution* is a map from region variables to region variables; an *effect substitution* is a map from effect variables to arrow effects. We we use S^t, S^r and S^e to range over type, region and effect substitutions, respectively. A *substitution* is a triple (S^t, S^r, S^e); we use S to range over substitutions. Substitution on basic semantic objects is defined as follows. Let $S = (S^t, S^r, S^e)$; then

Effects

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

Arrow Effects

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

Types and Region Variables

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

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

$$S(\mu \xrightarrow{\epsilon.\varphi} \mu') = S(\mu) \xrightarrow{S(\epsilon.\varphi)} S(\mu')$$

We write $S_2 \circ S_1$ for the composition of S_2 and S_1: $(S_2 \circ S_1)(x) = S_2(S_1(x))$. The *identity* substitution $\mathrm{Id} = (\mathrm{Id}^t, \mathrm{Id}^r, \mathrm{Id}^e)$, defined by $\mathrm{Id}^r(\rho)$ for all $\rho \in \mathrm{RegVar}$, $\mathrm{Id}^t(\alpha) = \alpha$ for all $\alpha \in \mathrm{TyVar}$, and $\mathrm{Id}^e(\epsilon) = \epsilon.\emptyset$ for all $\epsilon \in \mathrm{EffVar}$.

The *support* of a type substitution S^t, written $Supp(S^t)$, is the set $\{\alpha \in \mathrm{TyVar} \mid S^t(\alpha) \neq \alpha\}$. Similarly for region substitutions. The *support* of an effect substitution S^e, written $Supp(S^e)$, is the set $\{\epsilon \in \mathrm{EffVar} \mid S^e(\epsilon) \neq \epsilon.\emptyset\}$. The *support* of a substitution S, written $Supp(S)$, is the union of the supports of its three components. The *yield* of S, written $Yield(S)$, is $fv(Rng(S^t \downarrow (Supp(S^t))) \cup Rng(S^r \downarrow (Supp(S^r))) \cup Rng(S^e \downarrow (Supp(S^e))))$. The set of variables *involved in* S, written $Inv(S)$, is defined to be $Supp(S) \cup Yield(S)$.

A *finite substitution* is a triple $S = (S^t, S^r, S^e)$ where S^t, S^r and S^e are finite maps of the appropriate types. When S' is a substitution and $S = (S^t, S^r, S^e)$ is a finite substitution, we write $S' \bullet S$ to mean the finite substitution $(S' \bullet S^t, S' \bullet S^r, S' \bullet S^e)$. The composition $S' \bullet S$ is useful when S instantiates the bound variables of a type scheme and S' is a substitution applied to the whole type scheme (see the proof of Lemma 7 for an example).

Every finite substitution can be extended a (total) substitution on basic semantic objects and we shall often tacitly assume that this extension has taken place before the substitution is applied.

Application of a substitution S to a basis $B = (Q, \Phi)$ yields the basis $S(B) = (\{S(\rho) \mid \rho \in Q\}, \{S(\epsilon.\varphi) \mid \epsilon.\varphi \in \Phi\})$.

7.1 Contraction

It turns out, that unification always can be expressed as the composition of substitutions of a particularly simple kind, namely the so-called elementary contractions:

Definition 2 *Let $B = (Q, \Phi)$ be a consistent basis. A substitution S is an* elementary contraction *of B if*

1. $S = \mathrm{Id}$; *or*

2. $S = (\{\}, \{\rho_1 \mapsto \rho_2\}, \{\})$, *for some* $\rho_1, \rho_2 \in Q$*; or*

3. $S = (\{\}, \{\}, \{\epsilon_1 \mapsto \epsilon_2.\emptyset\})$, *where* $\{\epsilon_1, \epsilon_2\} \subseteq \mathrm{Dom}(\hat{\Phi})$ *and* $\hat{\Phi}(\epsilon_1) = \hat{\Phi}(\epsilon_2)$*; or*

4. $S = (\{\}, \{\}, \{\epsilon \mapsto \epsilon.\varphi\})$, *for some* ϵ *and* φ *with* $\epsilon \in \mathrm{Dom}(\hat{\Phi})$ *and* $B \vdash \varphi$ *and* $\varphi \supseteq \hat{\Phi}(\epsilon)$*;* □

In terms of constraints, the third item allows us to collapse two constraints $\epsilon_1 \supseteq \varphi$ and $\epsilon_2 \supseteq \varphi$ into a single constraint $\epsilon_2 \supseteq \varphi'$ (where φ' is φ with any occurrence of ϵ_1 replaced by ϵ_2). Similarly, the fourth item corresponds to strengthening a constraint $\epsilon \supseteq \varphi_1$ to $\epsilon \supseteq \varphi_1 \cup \varphi'$, provided $\varphi \cup \varphi'$ is consistent in B. (In particular, one cannot introduce region and effect variables which are not already in $\mathrm{Dom}(B)$.)

Elementary contractions are idempotent. If S is an elementary contraction of a consistent basis B, then $S(B)$ is consistent.

Let B be a consistent basis. We say that a substitution S is a *contraction* of B if it can be written as a composition $S = S_n \circ \ldots \circ S_1$, $n \geq 1$, such that S_1 is an elementary contraction of B and for all i with $1 < i \leq n$, S_i is an elementary contraction of $(S_{i-1} \circ \ldots \circ S_1)(B)$.

We say that B_1 is less than or equal to B_2, written $B_1 \leq B_2$, if there exists a contraction S of B_2 such that $B_1 = S(B_2)$. Note that if S is an elementary contraction of B and $S(B) \neq B$ then there exists no contraction S' of $S(B)$ such that $S'(S(B)) = B$. Thus $\leq$ is a partial order on consistent bases. We write $B_1 < B_2$ to mean $B_1 \leq B_2$ and $B_1 \neq B_2$.

Lemma 3 *Let B_0 be a finite, consistent basis. There exists no infinite descending chain:*

$$B_0 > B_1 > B_2 > \cdots$$

A proof of the above lemma may be found in [20]. The lemma plays a crucial rôle in the proof that the region inference algorithm terminates. The region inference algorithm iterates over the program, contracting the basis and modifying the program annotations in the process. Things are arranged so that if the program has been traversed without making the basis smaller, then the annotated program is well-typed according to the region inference rules. Lemma 3 ensures that this will eventually happen.

Lemma 4 *Let o be a basic semantic object and let S be a contraction of a basis B. If $B \vdash o$ then $S(B) \vdash S(o)$.*

In particular, if B is consistent and S is a contraction then $S(B)$ is consistent.

Proof By induction on the number of elementary contractions out of which S is constructed, with an inner induction on the depth of $B \vdash o$. □

8 Unification

8.1 Unification of Arrow Effects

Let $B = (Q, \Phi)$ be a consistent basis and assume $\epsilon_1.\varphi_1 \in \Phi$ and $\epsilon_2.\varphi_2 \in \Phi$. A substitution S is a *unifier for* $\epsilon_1.\varphi_1$ *and* $\epsilon_2.\varphi_2$ *in* B if $S(B)$ is consistent and $S(\epsilon_1.\varphi_1) = S(\epsilon_2.\varphi_2)$. A substitution S^* is a *most general unifier for* $\epsilon_1.\varphi_1$ *and* $\epsilon_2.\varphi_2$ *in* B, if S^* is a unifier for $\epsilon_1.\varphi_1$ and $\epsilon_2.\varphi_2$ in B and for every unifier S for $\epsilon_1.\varphi_1$ and $\epsilon_2.\varphi_2$ in B there exists a substitution S' such that

1. $S(\rho) = S'(S^*(\rho))$, for all $\rho \in Q$;

2. $S(\epsilon.\varphi) = S'(S^*(\epsilon.\varphi))$, for all $\epsilon.\varphi \in \Phi$;

The following algorithm takes two arrow effects as arguments and returns an effect substitution which is a most general unifier for the two arrow effects:

$$
\begin{array}{l}
\mathit{unifyArrEff}(\epsilon_1.\varphi_1, \epsilon_2.\varphi_2) = \\
\quad \text{let} \\
\qquad S_1^e = \text{if } \varphi_2 \subseteq \varphi_1 \text{ then } \mathrm{Id}^e \text{ else } \{\epsilon_1 \mapsto \epsilon_1.(\varphi_1 \cup \varphi_2)\} \\
\qquad S_2^e = \text{if } \varphi_1 \subseteq \varphi_2 \text{ then } \mathrm{Id}^e \text{ else } \{\epsilon_2 \mapsto \epsilon_2.(\varphi_1 \cup \varphi_2)\} \\
\qquad S_3^e = \text{if } \epsilon_1 = \epsilon_2 \text{ then } \mathrm{Id}^e \text{ else } \{\epsilon_2 \mapsto \epsilon_1.\emptyset\} \\
\quad \text{in} \\
\qquad S_3^e \circ S_2^e \circ S_1^e \\
\quad \text{end}
\end{array}
$$

Lemma 5 *Let $B = (Q, \Phi)$ be a consistent basis and let $\epsilon_1.\varphi_1 \in \Phi$ and $\epsilon_2.\varphi_2 \in \Phi$. Then $S^e = \mathit{unifyArrEff}(\epsilon_1.\varphi_1, \epsilon_2.\varphi_2)$ succeeds and*

1. *S^e is a most general unifier for $\epsilon_1.\varphi_1$ and $\epsilon_2.\varphi_2$ in B;*
2. *S^e is a contraction of B;*
3. *$S^e(B) = B$ implies $S^e = \mathrm{Id}^e$;*

Proof Let $(Q, \Phi) = B$. Since $B \vdash \varphi_1 \cup \varphi_2$ and $\varphi_1 \cup \varphi_2 \supseteq \varphi_1 = \hat{\Phi}(\epsilon_1)$, we have that S_1^e is an elementary contraction, even if $\varphi_2 \not\subseteq \varphi_1$. Thus $S_1^e(B)$ is consistent. Next, S_2^e is an elementary contraction of $S_1^e(B)$, since either $S_2^e = \mathrm{Id}^e$ or $S_1^e(B) \vdash \varphi_1 \cup \varphi_2$ and $\varphi_1 \cup \varphi_2 \supseteq \widehat{S_1^e(\Phi)}(\epsilon_2) = \varphi_2$. Thus $S_2^e(S_1^e(B))$ is consistent. Next, $S_2^e(S_1^e(\epsilon_1.\varphi_1)) = S_2^e(\epsilon_1.(\varphi_1 \cup \varphi_2)) = \epsilon_1.(\varphi_1 \cup \varphi_2)$ and $S_2^e(S_1^e(\epsilon_2.\varphi_2)) = S_2^e(\epsilon_2.\varphi_2) = \epsilon_2.(\varphi_1 \cup \varphi_2)$. Thus S_3^e is an elementary contraction of $S_2^e(S_1^e(B))$, even if $\epsilon_1 \neq \epsilon_2$. Thus S^e is a contraction of B.

The following calculation shows that S^e unifies $\epsilon_1.\varphi_1$ and $\epsilon_2.\varphi_2$: $S^e(\epsilon_1.\varphi_1) = S_3^e(S_2^e(S_1^e(\epsilon_1.\varphi_1))) = S_3^e(S_2^e(\epsilon_1.(\varphi_1 \cup \varphi_2))) = S_3^e(\epsilon_1.(\varphi_1 \cup \varphi_2)) = S_3^e(\epsilon_2.(\varphi_1 \cup \varphi_2)) = S_3^e(S_2^e(\epsilon_2.\varphi_2)) = S_3^e(S_2^e(S_1^e(\epsilon_2.\varphi_2)))$. Since S^e is a contraction of B we have that $S^e(B)$ is consistent, so S^e really is a unifier for $\epsilon_1.\varphi_1$ and $\epsilon_2.\varphi_2$ in B.

Finally, we show that S^e is a most general unifier. Let S be any unifier for $\epsilon_1.\varphi_1$ and $\epsilon_2.\varphi_2$ in B. Let $\epsilon'.\varphi' = S(\epsilon_1.\varphi_1) = S(\epsilon_2.\varphi_2)$ and let $S' = \{\epsilon' \mapsto \epsilon'.\varphi'\} \circ S$. We now show that this S' has the desired properties. Certainly, $S(\rho) = S'(S^e(\rho))$, for all $\rho \in Q$, since S^e is an effect substitution. Next, take an arbitrary $\epsilon.\varphi \in \Phi$. We proceed by case analysis:

$\boxed{\epsilon \neq \epsilon_1 \wedge \epsilon \neq \epsilon_2 \wedge \epsilon_1 \notin \varphi \wedge \epsilon_2 \notin \varphi}$ Here $S'(S^e(\epsilon.\varphi)) = S'(\epsilon.\varphi) = \{\epsilon' \mapsto \epsilon'.\varphi'\}(S(\epsilon.\varphi)) = S(\epsilon.\varphi)$ since $S(\epsilon.\varphi)$ and $\epsilon'.\varphi'$ both are members of $S(\Phi)$, which, by assumption, is effect consistent.

$\boxed{\epsilon \neq \epsilon_1 \wedge \epsilon \neq \epsilon_2 \wedge (\epsilon_1 \in \varphi \vee \epsilon_2 \in \varphi)}$ Let $\epsilon''.\varphi'' = S(\epsilon)$. We first prove

$$S(\varphi) = S'(S^e(\varphi)) \tag{7}$$

We have $S(\varphi) = S(\varphi \cup \varphi_1 \cup \varphi_2)$ since $\{\epsilon_1, \epsilon_2\} \cap \varphi \neq \emptyset$ and therefore, by consistency, $\{\epsilon_1\} \cup \varphi_1 \subseteq \varphi$ or $\{\epsilon_2\} \cup \varphi_2 \subseteq \varphi$, so that $S(\{\epsilon_1\} \cup \varphi_1) = S(\{\epsilon_2\} \cup \varphi_2) \subseteq S(\varphi)$. Moreover, we have $S(\{\epsilon_2 \mapsto \epsilon_1.\emptyset\}(\varphi \cup \varphi_1 \cup \varphi_2)) \subseteq S(\varphi \cup \varphi_1 \cup \varphi_2)$, even if $\epsilon_1 \notin \varphi$; for in that case we must have $\epsilon_2 \in \varphi$ (by case assumption) and therefore $\mathit{fv}(S(\epsilon_1)) \subseteq \{\epsilon'\} \cup \varphi' \subseteq S(\varphi)$. We do not necessarily have $S(\{\epsilon_2 \mapsto \epsilon_1.\emptyset\}(\varphi \cup \varphi_1 \cup \varphi_2)) = S(\varphi \cup \varphi_1 \cup \varphi_2)$.[2] But $\varphi' \cup S(\{\epsilon_2 \mapsto \epsilon_1.\emptyset\}(\varphi \cup \varphi_1 \cup \varphi_2)) = S(\varphi \cup \varphi_1 \cup \varphi_2)$, since $S(\epsilon_1.\varphi_1) = S(\epsilon_2.\varphi_2) = \epsilon'.\varphi'$. Thus $S(\varphi) = S(\varphi \cup \varphi_1 \cup \varphi_2) = \varphi' \cup S(\{\epsilon_2 \mapsto \epsilon_1.\emptyset\}(\varphi \cup \varphi_1 \cup \varphi_2)) = S'(S^e(\varphi))$, proving (7). Now the following computation gives the desired result: $S'(S^e(\epsilon.\varphi)) = S'(\epsilon.S^e(\varphi)) = \{\epsilon' \mapsto \epsilon'.\varphi'\}(S(\epsilon.S^e(\varphi))) = \{\epsilon' \mapsto \epsilon'.\varphi'\}(\epsilon''.(\varphi'' \cup S(S^e(\varphi)))) = \epsilon''.(\varphi'' \cup \varphi' \cup S(S^e(\varphi))) = \epsilon''.(\varphi'' \cup S'(S^e(\varphi))) = \epsilon''.(\varphi'' \cup S(\varphi)) = S(\epsilon.\varphi)$.

$\boxed{\epsilon = \epsilon_1}$ Since B is consistent we have $\epsilon.\varphi = \epsilon_1.\varphi_1$, so $S'(S^e(\epsilon.\varphi)) = \{\epsilon' \mapsto \epsilon'.\varphi'\}(S(S^e(\epsilon_1.\varphi_1))) =$

[2] Example: Consider $\epsilon_1.\varphi_1 = \epsilon_1.\{\epsilon_2\}$, $\epsilon_2.\varphi_2 = \epsilon_2.\emptyset$; let $S(\epsilon_1) = \epsilon'.\emptyset$ and $S(\epsilon_2) = \epsilon'.\{\epsilon', \rho\}$. Then $S(\epsilon_1.\{\epsilon_2\}) = S(\epsilon_2.\emptyset) = \epsilon'.\{\epsilon', \rho\}$, $S^e(\epsilon_1) = S^e(\epsilon_2) = \epsilon_1.\{\epsilon_1\}$ but $S(\{\epsilon_1\}) = \{\epsilon'\} \subset \{\epsilon', \rho\} = S(\{\epsilon_1, \epsilon_2\})$.

$\{\epsilon' \mapsto \epsilon'.\varphi'\}(S(\epsilon_1.\{\epsilon_2 \mapsto \epsilon_1.\emptyset\}(\varphi_1 \cup \varphi_2))) = \epsilon'.\varphi' = S(\epsilon_1.\varphi_1) = S(\epsilon.\varphi)$.

$\boxed{\epsilon = \epsilon_2}$ Since B is consistent we have $\epsilon.\varphi = \epsilon_2.\varphi_2$, so $S'(S^e(\epsilon.\varphi)) = \{\epsilon' \mapsto \epsilon'.\varphi'\}(S(S^e(\epsilon_2.\varphi_2))) = \{\epsilon' \mapsto \epsilon'.\varphi'\}(S(\epsilon_1.\{\epsilon_2 \mapsto \epsilon_1.\emptyset\}(\varphi_1 \cup \varphi_2))) = \epsilon'.\varphi' = S(\epsilon_2.\varphi_2) = S(\epsilon.\varphi)$. □

8.2 Unification of Types

We now extend the above results about unification to cover annotated types. Let $B = (Q, \Phi)$ be a consistent basis and let τ_1 and τ_2 be types satisfying $B \vdash \tau_1$ and $B \vdash \tau_2$. A substitution S is a *unifier of* τ_1 *and* τ_2 *in* B if S takes the form $(\{\}, S^r, S^e)$, $S(B)$ is consistent and $S(\tau_1) = S(\tau_2)$. Note that a unifier is not allowed to have type variables in its support (we do not attempt to do ML type inference and region inference simultaneously.) A substitution S^* is a *most general unifier of* τ_1 *and* τ_2 *in* B, if S^* is a unifier of τ_1 and τ_2 in B and for every unifier S of τ_1 and τ_2 in B there exists a substitution S' such that:

1. $S(\rho) = S'(S^*(\rho))$, for all $\rho \in Q$;
2. $S(\epsilon.\varphi) = S'(S^*(\epsilon.\varphi))$, for all $\epsilon.\varphi \in \Phi$;

Notice that the condition in item (1) is "for all $\rho \in Q$", not just "for all $\rho \in \mathit{frv}(\tau_1, \tau_2)$"; similarly, note that the condition in item (2) is "for all $\epsilon.\varphi \in \Phi$", not just "for all $\epsilon.\varphi \in \mathit{arreff}(\tau_1) \cup \mathit{arreff}(\tau_2)$". This stronger property of principal unifiers will be used several times in what follows. One can prove that if S^* is a most general unifier for τ_1 and τ_2 in B then $\mathit{Inv}(S^*) \subseteq \mathit{fv}(\tau_1, \tau_2)$.

The algorithm below computes most general unifiers in consistent bases:

$\mathit{unifyRho}(\rho_1, \rho_2) =$
  if $\rho_1 = \rho_2$ then Id
  else $(\{\}, \{\rho_2 \mapsto \rho_1\}, \{\})$

$\mathit{unifyMu}((\tau_1, \rho_1), (\tau_2, \rho_2)) =$
  let
    $S_1 = \mathit{unifyTy}(\tau_1, \tau_2)$
    $S_2 = \mathit{unifyRho}(S_1\rho_1, S_1\rho_2)$
  in
    $S_2 \circ S_1$
  end

$\mathit{unifyTy}(\tau_1, \tau_2) =$ case (τ_1, τ_2) of
  $(\mathtt{int}, \mathtt{int}) \Rightarrow$ Id
  $\mid (\alpha, \beta) \Rightarrow$ if $\alpha = \beta$ then Id else FAIL
  $\mid (\mu_1 \xrightarrow{\epsilon_1.\varphi_1} \mu_2, \mu_1' \xrightarrow{\epsilon_1'.\varphi_1'} \mu_2') \Rightarrow$
    let
      $S_1 = \mathit{unifyMu}(\mu_1, \mu_1')$
      $S_2 = \mathit{unifyArrEff}(S_1(\epsilon_1.\varphi_1), S_1(\epsilon_1'.\varphi_1'))$
      $S_3 = \mathit{unifyMu}(S_2(S_1(\mu_2)), S_2(S_1(\mu_2')))$
    in
      $S_3 \circ S_2 \circ S_1$
    end
  $\mid (\_, \_) \Rightarrow$ FAIL

Lemma 6 *Let B be a consistent basis and assume $B \vdash \tau_1$ and $B \vdash \tau_2$ and* $\mathrm{ML}(\tau_1) = \mathrm{ML}(\tau_2)$. *Then $S^* = \mathit{unifyTy}(\tau_1, \tau_2)$ succeeds and*

1. *S^* is a most general unifier of τ_1 and τ_2 in B;*
2. *S^* is a contraction of B;*
3. *$S^*(B) = B$ implies $S^* = \mathrm{Id}$;*

Similarly for types with places (μ) and region variables.

Proof By induction on the structure of $\mathrm{ML}(\tau_1)$. We show only the case $\tau_1 = \mu_1 \xrightarrow{\epsilon_1.\varphi_1} \mu_2$, in which case $\mathrm{ML}(\tau_1) = \mathrm{ML}(\tau_2)$ gives that τ_2 must take the form $\mu_1' \xrightarrow{\epsilon_1'.\varphi_1'} \mu_2'$. We have $B \vdash \mu_1$, $B \vdash \mu_1'$ and $\mathrm{ML}(\mu_1) = \mathrm{ML}(\mu_1')$. Thus by induction

$$S_1 \text{ is a most general unifier for } \mu_1 \text{ and } \mu_1' \text{ in } B \tag{8}$$

$$S_1 \text{ is a contraction of } B \tag{9}$$

$$S_1(B) = B \text{ implies } S_1 = \mathrm{Id} \tag{10}$$

By (9) and Lemma 4 we have $S_1(B) \vdash S_1(\tau_1)$ and $S_1(B) \vdash S_1(\tau_2)$. Therefore, by Lemma 5, $S_2 = \mathit{unifyArrEff}(S_1(\epsilon_1.\varphi_1), S_1(\epsilon'_1.\varphi'_1))$ succeeds and

$$S_2 \text{ is a most general unifier for } S_1(\epsilon_1.\varphi_1) \text{ and } S_1(\epsilon'_1.\varphi'_1) \text{ in } S_1(B) \tag{11}$$

$$S_2 \text{ is a contraction of } S_1(B) \tag{12}$$

$$S_2(S_1(B)) = S_1(B) \text{ implies } S_2 = \mathrm{Id} \tag{13}$$

By Lemma 4 on (12) we get $S_2(S_1(B)) \vdash S_2(S_1(\tau_1))$ and $S_2(S_1(B)) \vdash S_2(S_1(\tau_2))$. Also, $\mathrm{ML}(S_2(S_1\mu_2)) = \mathrm{ML}(\mu_2) = \mathrm{ML}(\mu'_2) = \mathrm{ML}(S_2(S_1\mu'_2))$, since S_1 and S_2 are contractions. By induction, $S_3 = \mathit{unifyMu}(S_2(S_1(\mu_2)), S_2(S_1(\mu'_2)))$ succeeds and

$$S_3 \text{ is a most general unifier for } S_2(S_1(\mu_2)) \text{ and } S_2(S_1(\mu'_2)) \text{ in } S_2(S_1(B)) \tag{14}$$

$$S_3 \text{ is a contraction of } S_2(S_1(B)) \tag{15}$$

$$S_3(S_2(S_1(B))) = S_2(S_1(B)) \text{ implies } S_3 = \mathrm{Id} \tag{16}$$

Thus $\mathit{unifyTy}(\tau_1, \tau_2)$ succeeds with result $S^* = S_3 \circ S_2 \circ S_1$. From (8), (11) and (14) we get that S^* is a unifier for τ_1 and τ_2 in B. From (9), (12) and (15) we get that S^* is a contraction of B. From (9), (12) and (15) together with (10), (13) and (16) we get that $S^*(B) = B$ implies $S^* = \mathrm{Id}$. It remains to prove that S^* is a *most general* unifier for τ_1 and τ_2 in B. Let S be any unifier for τ_1 and τ_2 in B and let $(Q, \Phi) = B$. Now S is a unifier for μ_1 and μ'_1 in B. Thus, by (8), there exists an S'_1 such that

$$\forall \rho \in Q.S(\rho) = S'_1(S_1(\rho)) \wedge \forall \epsilon.\varphi \in \Phi.S(\epsilon.\varphi) = S'_1(S_1(\epsilon.\varphi)) \tag{17}$$

Thus, since S is a unifier for $\epsilon_1.\varphi_1$ and $\epsilon'_1.\varphi'_1$, S'_1 is a unifier for $S_1(\epsilon_1.\varphi_1)$ and $S_1(\epsilon'_1.\varphi'_1)$ in $S_1(B)$. Thus, by (11), there exists a S'_2 such that

$$\forall \rho \in S_1(Q).S'_1(\rho) = S'_2(S_2(\rho)) \wedge \forall \epsilon.\varphi \in S_1(\Phi).S'_1(\epsilon.\varphi) = S'_2(S_2(\epsilon.\varphi)) \tag{18}$$

Since S is a unifier for μ_2 and μ'_2 in B and (17) and (18) we have that S'_2 is a unifier for $S_2(S_1(\mu_2))$ and $S_2(S_1(\mu'_2))$ in $S_2(S_1(B))$. Thus, by (14), there exists an S'_3 such that

$$\forall \rho \in S_2(S_1(Q)).S'_2(\rho) = S'_3(S_3(\rho)) \wedge \forall \epsilon.\varphi \in S_2(S_1(\Phi)).S'_2(\epsilon.\varphi) = S'_3(S_3(\epsilon.\varphi)) \tag{19}$$

By (17), (18) and (19) we get the desired

$$\forall \rho \in Q.S(\rho) = S'_3(S^*(\rho)) \wedge \forall \epsilon.\varphi \in \Phi.S(\epsilon.\varphi) = S'_3(S^*(\epsilon.\varphi)) \tag{20}$$

□

It turns out that the premises of lemma 6 are always met in the region inference algorithm we use (this is proved in [20]). Therefore, unification never fails. Informally, the fact that the premise $\mathrm{ML}(\tau_1) = \mathrm{ML}(\tau_2)$ is satisfied is not surprising, for

1. region inference takes place after type inference; and
2. roughly, region inference simply re-typechecks the program using annotated versions of the types that were determined by type inference, so if two region-annotated types are unified, they will have the same underlying ML type.

Example 7.1 The two types

$$\tau_1 = (\mathtt{int}, \rho_1) \xrightarrow{\epsilon_2.\{\rho_1,\rho_3,\rho_4\}} (\mathtt{int}, \rho_4)$$

and

$$\tau_2 = (\mathtt{int}, \rho_5) \xrightarrow{\epsilon_6.\{\rho_5,\rho_7,\rho_8\}} (\mathtt{int}, \rho_8)$$

are consistent in the basis $B = (\{\rho_1, \rho_3, \rho_4, \rho_5, \rho_7, \rho_8\}, \{\epsilon_2.\{\rho_1, \rho_3, \rho_4\}, \epsilon_6.\{\rho_5, \rho_7, \rho_8\}\}$. The result of $\mathit{unifyTy}(\tau_1, \tau_2)$ is the substitution $S = (S^t, S^r, S^e)$, where $S^t = \mathrm{Id}$, $S^r = \{\rho_5 \mapsto \rho_1, \rho_8 \mapsto \rho_4\}$ and $S^e = \{\epsilon_6 \mapsto \epsilon_2.\{\rho_1, \rho_3, \rho_4, \rho_7\}, \epsilon_2 \mapsto \epsilon_2.\{\rho_1, \rho_3, \rho_4, \rho_7\}\}$. By Lemma 6, S is a contraction of B and S is a most general unifier for τ_1 and τ_2 in B. □

In Example 7.1, notice that S does not map ρ_3 and ρ_7 to the same region variable: region and effect variables which occur in effects only in the types that are unified (i.e., they do not occur paired with a type constructor or as the handle of an arrow effect) are not affected by the unification algorithm. This distinction between different occurrences is important, also for the termination of region inference, so we define it formally. For basic semantic objects o, we define sets $\mathit{pfrv}(o)$ and $\mathit{pfev}(o)$ as follows:

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

The set $\mathit{pfev}(o)$ is called the set of *primary (free) effect variables of* o; the set $\mathit{pfrv}(o)$ is called the set of *primary (free) region variables of* o. We say that an effect variable ϵ is a *secondary effect variable* in τ if $\epsilon \in \mathit{fev}(\tau) \setminus \mathit{pfev}(\tau)$; similarly, we say that a region variable ρ is *secondary* in τ if $\rho \in \mathit{frv}(\tau) \setminus \mathit{pfrv}(\tau)$.

Given types τ_1 and τ_2 with $\mathrm{ML}(\tau_1) = \mathrm{ML}(\tau_2)$, $\mathit{unifyTy}(\tau_1, \tau_2)$ returns a substitution which has only region and effect variables which are primary in τ_1 or τ_2 in its support. (In Example 7.1, ρ_7 and ρ_3 are secondary variables; all the other variables are primary.) This is as one would expect. For in order to obtain good separation of lifetimes, unification should only identify two region variables when they must denote the same region. Secondary occurrences of region variables do not force identification of regions. For example, amongst the less general unifiers for the two arrow effects $\epsilon.\{\rho_1, \rho_2\}$ and $\epsilon'.\{\rho_3, \rho_4\}$ we have $S = (\{\epsilon \mapsto \epsilon'.\emptyset\}, \{\rho_1 \mapsto \rho_3, \rho_2 \mapsto \rho_4\})$ and $S' = (\{\epsilon \mapsto \epsilon'.\emptyset\}, \{\rho_1 \mapsto \rho_4, \rho_2 \mapsto \rho_3\})$. These unifiers lead to smaller effects than the most general unifier, but they also lead to unnecessary identifications of regions and are therefore undesirable.

9 Type Schemes

Type schemes resemble the type schemes of Damas and Milner[3] but with additional quantification over region variables and effect variables:

$$\sigma ::= \forall \alpha_1 \cdots \alpha_n \rho_1 \cdots \rho_k \epsilon_1 \cdots \epsilon_m . \tau \qquad (n \geq 0,\ k \geq 0 \text{ and } m \geq 0)$$

Here τ is the *body* of σ, written $\mathit{body}(\sigma)$. The *bound variables of* σ, written $\mathit{bv}(\sigma)$ is the set $\{\alpha_1, \ldots, \alpha_n\} \cup \{\rho_1, \ldots, \rho_k\} \cup \{\epsilon_1, \ldots, \epsilon_m\}$; we define the set of *bound type variables of* σ, written $\mathit{btv}(\sigma)$, to be the set $\{\alpha_1, \ldots, \alpha_n\}$. The *free variables of* σ, written $\mathit{fv}(\sigma)$, is the set $\mathit{fv}(\tau) \setminus \mathit{bv}(\sigma)$. The bound variables of a type scheme must be distinct. The *arity* of σ is the triple (n, k, m).

The erase function, ML, of Section 5 is extended to type schemes by

$$\mathrm{ML}(\forall \alpha_1 \cdots \alpha_n \rho_1 \cdots \rho_k \epsilon_1 \cdots \epsilon_m . \tau) = \forall \alpha_1 \cdots \alpha_n . \mathrm{ML}(\tau)$$

We sometimes regard a type τ as the trivial type scheme $\forall . \tau$. Type schemes that arise from each other by renaming of bound variables are considered equal.

A type τ' is an *instance of* $\sigma = \forall\alpha_1\cdots\alpha_n\rho_1\cdots\rho_k\epsilon_1\cdots\epsilon_m.\tau$, written $\sigma \geq \tau'$, if there exists a substitution S such that $Supp(S) \subseteq bv(\sigma)$ and $S(\tau) = \tau'$. When we want to make a particular S explicit, we say that τ' is an instance of σ *via* S, written $\sigma \geq \tau'$ *via* S. Furthermore, we say that a type scheme $\sigma' = \forall\alpha'_1\cdots\alpha'_{n'}\rho'_1\cdots\rho'_{k'}\epsilon'_1\cdots\epsilon'_{m'}.\tau'$ is an *instance of* σ *(via* S*)*, written $\sigma \geq \sigma'$ (via S), if $\sigma \geq \tau'$ and $bv(\sigma') \cap fv(\sigma) = \emptyset$. As in Milner's type system one has:

Lemma 7 *Let σ and σ' be type schemes. Then $\sigma \geq \sigma'$ iff for every τ, $\sigma' \geq \tau$ implies $\sigma \geq \tau$.*

Proof Assume that for every τ, $\sigma' \geq \tau$ implies $\sigma \geq \tau$. By renaming the bound variables of σ', if necessary, achieve that $bv(\sigma') \cap fv(\sigma) = \emptyset$. Then $\sigma' \geq body(\sigma')$ and thus, by assumption, $\sigma \geq body(\sigma')$, showing $\sigma \geq \sigma'$. Conversely, assume $\sigma \geq \sigma'$ *via* S. Without loss of generality, we can assume that S is finite and that $\mathrm{Dom}(S) = bv(\sigma)$. Assume $\sigma' \geq \tau$ *via* S'. We then have $\sigma \geq \tau$ *via* $S' \bullet S$. To prove this, it suffices to prove

$$(S' \circ S)\tau_0 = (S' \bullet S)\tau_0 \tag{21}$$

where $\tau_0 = body(\sigma)$, for by assumption we have $(S' \circ S)\tau_0 = \tau$. We prove (21) by induction on the structure of τ_0. We show only the case $\epsilon.\varphi \in \mathit{arreff}(\tau_0)$. Note that $(S' \bullet S)(\varphi) = (S' \circ S)(\varphi)$, since $bv(\sigma') \cap fv(\sigma) = \emptyset$. Now there are two cases. First, assume $\epsilon \in \mathrm{Dom}(S)$. Let $\epsilon'.\varphi' = S(\epsilon)$ and let $\epsilon''.\varphi'' = S'(\epsilon')$. Thus $(S' \bullet S)(\epsilon.\varphi) = \epsilon''.(\varphi'' \cup S'(\varphi') \cup (S' \bullet S)(\varphi)) = \epsilon''.(\varphi'' \cup S'(\varphi') \cup (S' \circ S)(\varphi)) = (S' \circ S)(\epsilon.\varphi)$, as desired. Second, assume $\epsilon \notin \mathrm{Dom}(S)$. Then $(S' \bullet S)(\epsilon.\varphi) = \epsilon.((S' \bullet S)(\varphi)) = \epsilon.((S' \circ S)(\varphi)) = (S' \circ S)(\epsilon.\varphi)$, since $\epsilon \in fv(\sigma)$ and $bv(\sigma') \cap fv(\sigma) = \emptyset$. $\square$

As a consequence, $\geq$ is a transitive relation on type schemes. We say that σ and σ' are *equivalent*, written $\sigma \equiv \sigma'$, if $\sigma \geq \sigma'$ and $\sigma' \geq \sigma$. This relation is larger than just renaming of bound variables. For example, letting $\mu_0 = (\texttt{int}, \rho_0)$ we have

$$\forall\epsilon\rho.\mu_0 \xrightarrow{\epsilon.\{\rho\}} \mu_0 \equiv \forall\epsilon\rho\rho'.\mu_0 \xrightarrow{\epsilon.\{\rho,\rho'\}} \mu_0 \tag{22}$$

Thus $\geq$ induces a partial order on equivalence classes of type schemes. This partial order is non-well-founded (just like the corresponding partial order on ML type schemes).

There are certain type schemes we want to reject as ill-formed, because they are in fundamental conflict with the idea of basing region inference on unification of consistent types.

Definition 8 *A type scheme $\sigma = \forall\alpha_1\cdots\alpha_n\rho_1\cdots\rho_k\epsilon_1\cdots\epsilon_m.\tau$ is said to be* well-formed *if, for every arrow effect $\epsilon.\varphi$ occurring on a function arrow in τ if $\epsilon \notin bv(\sigma)$ then $fv(\varphi) \cap bv(\sigma) = \emptyset$. Otherwise, σ is said to be* ill-formed.

In other words, well-formedness requires that if the handle of an arrow effect in τ is free then the entire arrow effect is free.

We shall later impose a restriction which bans ill-formed type schemes. That this is to be expected is illustrated by the following example.

Example 8.1 Consider the type scheme $\sigma = \forall\rho.(\texttt{int}, \rho) \xrightarrow{\epsilon.\{\rho\}} (\texttt{int}, \rho)$ which is ill-formed. The problem with σ is that if it is instantiated twice, via different substitutions, say $S_1 = \{\rho \mapsto \rho_1\}$ and $S_2 = \{\rho \mapsto \rho_2\}$, then the two resulting types $(\texttt{int}, \rho_1) \xrightarrow{\epsilon.\{\rho_1\}} (\texttt{int}, \rho_1)$ and $(\texttt{int}, \rho_2) \xrightarrow{\epsilon.\{\rho_2\}} (\texttt{int}, \rho_2)$ are not consistent in any basis: ϵ cannot both "denote" the set $\{\rho_1\}$ and the set $\{\rho_2\}$. If, however, ϵ had been quantified as well, the type schemes would have been well-formed and one could pick fresh effect variables each time the type scheme were instantiated. $\square$

10 Polymorphic Recursion

In practice, much of the ability of the region inference to reclaim memory comes from rule (5). The crucial point is that within e_1, the type environment associates f with the region type scheme $\hat{\sigma}$, so that free occurrences of f in e_1 potentially can be given types different from τ_0. We refer to this feature as region-polymorphic recursion. (An example of region-polymorphic recursion at work was presented in Section 2.)

Notice that rule (5) does not allow polymorphic recursion in types: the type scheme for f in e_1 is only $\hat{\sigma}$, not σ. Thus the region rules are faithful to the treatment of recursive functions in ML as far as type polymorphism is concerned[10]. The reason for this limitation is that type inference for (type-) polymorphic recursion is equivalent to semi-unification[5, 8], which is undecidable[7].

However, it can be proved that every well-typed source expression can be region-annotated in accordance with the region inference rules[24]. Moreover, in the absence of region-polymorphic recursion, Milner's notion of principal type schemes extends to region inference[22]. In the presence of polymorphic recursion, however, it is not known whether (in some sense) it is possible to infer a principal region type scheme for every ML-typable expression. There seems to be a tension between, on the one hand, the desire for obtaining region type schemes that are as general as possible and on the other hand the desire to know that region inference always terminates.

We have chosen to resolve this (alleged) tension in the favour of termination: the algorithm described in [20] always finds a region annotation which is legal according to the region inference rules, although in doing so, it may fail to give every recursive function in the program a region type scheme which is the most general one permitted by the region inference rules.

The algorithm in [20] handles polymorphic recursion using a two-phase approach. The first phase, called *spreading*, generates a (possibly large) basis by choosing fresh region and effect variables wherever a region or effect annotation is required in the target program. The second phase, called *fixed point resolution*, collapses region variables (and effect variables) by unifying region-annotated types. Unification of region-annotated types produces contractions of the basis (see Section 8), and a cornerstone of the termination proof is that no finite basis can be contracted indefinitely (Lemma 3). Thus a crucial idea in the algorithm is to separate the generation of fresh variables from the region inference proper. The same idea is used for ensuring termination in our constraint-based approach mentioned in Section 3.

But is it possible to know how many region and effect variables one needs for doing region inference for a given program without actually doing the region inference? This is where the distinction between secondary and primary region (and effect) variables in types becomes important. Given an ML type τ^{ML}, let us consider the set of region annotated types τ for which $\mathrm{ML}(\tau) = \tau^{ML}$. For every such τ, the number of region and effect variables that appear in primary positions in τ is bounded by the size of τ: one at most needs one effect variable for each function arrow in τ^{ML} and one region variable for each occurrence of a type constructor or a type variable in τ^{ML}. However, it is not clear how many secondary region and effect variables one might wish to have on function arrows that appear in τ^{ML}. But in that case, how many bound region and effect variables does it take at most to make a region-annotated type scheme $\hat{\sigma}$ of the form found in rule (5)? Since there is no obvious bound on the number of secondary region and effect variables in τ_0, and since there is in general nothing that prevents secondary region and effect variables from being universally quantified in $\hat{\sigma}$, there is no obvious bound on how many bound region and effect variables one will need for $\hat{\sigma}$.

Why does one need a bound on the arity of $\hat{\sigma}$? Because each free occurrence of f within e_1 and e_2 potentially gives rise to generation of k fresh region variables and m fresh effect variables, where $(\_, k, m)$ is the arity of $\hat{\sigma}$. Without a bound on k and m, how can we generate all the fresh region and effect variables we need in the initial phase?

To avoid an ever-expanding number of quantified secondary region and effect variables in region-annotated type schemes, our algorithm never produces type schemes which have quantified region or effect variables which occur in secondary positions of the body of the type scheme only (see Section 12). Formally, we define:

Definition 9 *A type scheme* $\sigma = \forall\alpha_1\cdots\alpha_n\rho_1\cdots\rho_k\epsilon_1\cdots\epsilon_m.\tau$ *is said to have* structural quantification *if* $\{\rho_1,\ldots,\rho_k\} \subseteq \mathit{pfrv}(\tau)$, $\{\epsilon_1,\ldots,\epsilon_m\} \subseteq \mathit{pfev}(\tau)$ *and* $\{\alpha_1,\ldots,\alpha_n\} \subseteq \mathit{ftv}(\tau)$.

The operation by which we form type schemes (see Section 12.2) only produces type schemes which have structural quantification.

Example 9.1 The following expression has a type which contains a secondary region variable:

$$\begin{array}{l}\lambda z:(\alpha,\rho_0).\texttt{let}\ \ x = 1\ \texttt{at}\ \ \rho_1\\ \qquad\qquad \texttt{in}\ \ (\lambda y:(\texttt{int},\rho_2).(x+y)\ \texttt{at}\ \rho_3)\ \texttt{at}\ \rho_4:\\ \qquad\qquad \underbrace{(\alpha,\rho_0)\xrightarrow{\epsilon_1.\{\rho_1,\rho_4\}}((\texttt{int},\rho_2)\xrightarrow{\epsilon_2.\{\rho_1,\rho_2,\rho_3\}}(\texttt{int},\rho_3),\rho_4)}_{\tau_0}\end{array}$$

Here ρ_1 is a secondary region variable in τ_0; it represents the region of a temporary value which is not necessarily in the same region as the argument or result of either of the two functions but which is captured in the function value (closure) of the function $\lambda y\cdots$. If the above function is used in a declaration of a region-polymorphic function

$$\texttt{letrec}\ f(z)\ \texttt{=}\ \texttt{let}\ x = 1\cdots$$

then one might want to give f the type scheme $\sigma = \forall\rho_0\rho_1\rho_2\rho_3\rho_4\epsilon_1\epsilon_1.\tau_0$ but this type scheme has non-structural quantification. Instead, the *RegEffClos* algorithm (Section 12.2) will construct the less general type scheme $\sigma = \forall\rho_0\rho_2\rho_3\rho_4\epsilon_1\epsilon_1.\tau_0$. $\square$

We are now ready to extend the definition of consistency to cover type schemes. Let $\sigma = \forall\alpha_1\cdots\alpha_n\rho_1\cdots\rho_k\epsilon_1\cdots\epsilon_m.\tau$ be a type scheme with structural quantification. The *bound basis of* σ, written $\mathit{bound}(\sigma)$, is the basis $(\{\rho_1,\ldots,\rho_k\},\{\epsilon.\varphi \in \mathit{arreff}(\tau) \mid \epsilon \in \mathit{bv}(\sigma)\})$.

Definition 10 *Type scheme* $\sigma = \forall\alpha_1\cdots\alpha_n\rho_1\cdots\rho_k\epsilon_1\cdots\epsilon_m.\tau$ *is* consistent in basis B, *written* $B \vdash \sigma$, *if* B *is consistent,* σ *has structural quantification and, letting* $B' = \mathit{bound}(\sigma)$, *the basis* $B \uplus B'$ *exists and is consistent and* $B \uplus B' \vdash \tau$.

Notice that if $B \vdash \sigma$ and $\sigma = \forall\alpha_1\cdots\alpha_n\rho_1\cdots\rho_k\epsilon_1\cdots\epsilon_m.\tau$ then, since σ has structural quantification, we have $\mathrm{Dom}(\widehat{\Phi_\sigma}) = \{\epsilon_1,\ldots,\epsilon_m\}$, where $\Phi_\sigma = \Phi$ *of* $\mathit{bound}(\sigma)$.

Lemma 11 *If* $B \vdash \sigma$ *then* σ *is well-formed.*

Proof Assume $B \vdash \sigma$, where $\sigma = \forall\alpha_1\cdots\alpha_n\rho_1\cdots\rho_k\epsilon_1\cdots\epsilon_m.\tau$. Let $B' = (Q',\Phi') = \mathit{bound}(\sigma)$. By $B \vdash \sigma$ we have that the basis $B \uplus B'$ exists, is consistent and $B \uplus B' \vdash \tau$. Take $\epsilon.\varphi \in \mathit{arreff}(\tau)$ and assume $\epsilon \notin \{\epsilon_1,\ldots,\epsilon_m\}$. Now $B \uplus B' \vdash \tau$ implies $\epsilon.\varphi \in \Phi$ *of* $(B \uplus B')$. Since $B \uplus B'$ is consistent and $\epsilon \notin \mathit{bv}(\sigma)$ we have $\epsilon.\varphi \in B$. But then, since B itself is consistent we have $\mathit{fv}(\epsilon.\varphi) \subseteq \mathit{fv}(B)$ and thus $\mathit{fv}(\epsilon.\varphi) \cap (Q',\mathrm{Dom}(\widehat{\Phi'})) = \emptyset$. $\square$

The definition of consistency does not depend on any particular choice of bound variables in σ, except that the bound variables must be chosen disjoint from B.

We emphasise that consistency of type schemes is introduced for algorithmic reasons only. Soundness of the region inference rules has been proved without assuming consistency[23, 24]. Also, we have not found the restriction to consistent type schemes (and the loss in polymorphism it entails) to be serious in practice.

Definition 12 *A type environment TE is* consistent in $B = (Q,\Phi)$, *written* $B \vdash \mathit{TE}$, *if for all* $x \in \mathrm{Dom}(\mathit{TE})$, *letting* $(\sigma,\rho) = \mathit{TE}(x)$, *one has* $B \vdash \sigma$ *and* $\rho \in Q$.

This definition is used in the inference rules in Section 4.

11 Instantiation of Type Schemes

An *instantiation list* is a triple of the form

$$([\tau_1, \ldots, \tau_n], [\rho_1, \ldots, \rho_k], [\epsilon_1.\varphi_1, \ldots, \epsilon_m.\varphi_m]) \qquad n \geq 0, k \geq 0, m \geq 0$$

The triple (n, k, m) is called the *arity* of the instantiation list. We use il to range over instantiation lists. Instantiation lists annotate program variables, see rules (1) and (2). We say that il is *consistent in basis* B, written $B \vdash il$, if $B \vdash \tau_i$, for all $i = 1..n$, $B \vdash \rho_i$, for all $i = 1..k$, and $B \vdash \epsilon_i.\varphi_i$, for all $i = 1..m$.

Let σ be a type scheme $\forall\alpha_1 \cdots \alpha_n \rho_1 \cdots \rho_k \epsilon_1 \cdots \epsilon_m.\tau$ and let

$$il = ([\tau_1, \ldots, \tau_n], [\rho'_1, \ldots, \rho'_k], [\epsilon'_1.\varphi'_1, \ldots, \epsilon'_m.\varphi'_m])$$

be an instantiation list with the same arity as σ. Let $S = (\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}, \{\rho_1 \mapsto \rho'_1, \ldots, \rho_k \mapsto \rho'_k\}, \{\epsilon_1 \mapsto \epsilon'_1.\varphi'_1, \ldots, \epsilon_m \mapsto \epsilon'_m.\varphi'_m\})$. Then we refer to the type $\tau' = S(\tau)$ as the *instance of* σ *given by* il; we write $\sigma \geq \tau'$ *via* il to mean that τ' is the instance of σ given by il.

Consider the situation where B is a basis, σ a type scheme, il an instantiation list of the same arity as σ and assume $B \vdash \sigma$ and $B \vdash il$. Surprisingly, perhaps, it is not necessarily the case that the instance of σ via il is consistent in B. The reason is that the instantiation may produce arrow effects which are strictly larger than the arrow effects found in B.

Example 10.1 Consider $B = (\{\rho_0\}, \{\epsilon_0.\emptyset\})$, $\sigma = \forall\epsilon\rho.(\mathtt{int}, \rho_0) \xrightarrow{\epsilon.\{\rho\}} (\mathtt{int}, \rho)$ and $S = (\{\rho \mapsto \rho_0\}, \{\epsilon \mapsto \epsilon_0.\emptyset\})$. The instance of σ via S is the type $\tau' = (\mathtt{int}, \rho_0) \xrightarrow{\epsilon_0.\{\rho_0\}} (\mathtt{int}, \rho_0)$, which is not consistent in B, since the arrow effect $\epsilon_0.\emptyset$ has "grown" to become $\epsilon_0.\{\rho_0\}$.Another way to explain this "growth" is as follows. Since ϵ and ρ are quantified in σ, the arrow effect $\epsilon.\{\rho\}$ can be thought of a quantified constraint: $\forall\epsilon, \rho.\big(\epsilon \supseteq \{\rho\} \Rightarrow (\mathtt{int}, \rho_0) \xrightarrow{\epsilon} (\mathtt{int}, \rho)\big)$. Thus instantiating ϵ and ρ to ϵ_0 and ρ_0, respectively, results in the constraint $\epsilon_0 \supseteq \{\rho_0\}$. We represent this "growth" of the set which ϵ_0 denotes by application of the contraction $\{\epsilon_0 \mapsto \epsilon_0.\{\rho_0\}\}$ to B. □

In general, the operation of instantiating σ using il in B returns not just a type, τ', but also a contraction, which can be applied to B to obtain a basis in which τ' is consistent.

The instantiation algorithm is defined by the mutually recursive function declarations shown below. All the functions return an effect substitution, which is a contraction of the basis; the main function is $inst(\sigma, il)$ which returns both an effect substitution and a type, the latter being the instance of σ via il.

$$
\begin{array}{l}
\mathit{instArrEff}(S^e, \epsilon.\varphi) = \\
\quad \text{if } \epsilon \notin \mathrm{Dom}(S^e) \text{ then } \mathrm{Id}^e \\
\quad \text{else let } \epsilon_1.\varphi_1 = S^e(\epsilon) \\
\qquad\quad \text{in if } S^e(\varphi) \subseteq \varphi_1 \text{ then } \mathrm{Id}^e \\
\qquad\qquad \text{else } \{\epsilon_1 \mapsto \epsilon_1.(\varphi_1 \cup S^e(\varphi))\} \\
\qquad\quad \text{end} \\
\\
\mathit{instTy}(S^e, \tau) = \text{case } \tau \text{ of} \\
\quad \alpha \Rightarrow \mathrm{Id}^e \\
\quad \mid \mathtt{int} \Rightarrow \mathrm{Id}^e \\
\quad \mid \mu_1 \xrightarrow{\epsilon.\varphi} \mu_2 \Rightarrow \\
\qquad \text{let } S^e_1 = \mathit{instMu}(S^e, \mu_1) \\
\qquad\quad S^e_2 = \mathit{instMu}(S^e_1 \bullet S^e, S^e_1(\mu_2)) \\
\qquad\quad S^e_3 = \mathit{instArrEff}(S^e_2 \bullet (S^e_1 \bullet S^e), \\
\qquad\qquad\qquad (S^e_2 \circ S^e_1)(\epsilon.\varphi)) \\
\qquad \text{in } S^e_3 \circ S^e_2 \circ S^e_1 \text{ end}
\end{array}
$$

$$
\begin{array}{l}
\mathit{instMu}(S^e, (\tau, \rho)) = \mathit{instTy}(S^e, \tau) \\
\\
\mathit{inst}(\sigma, il) = \\
\quad \text{let } \forall\alpha_1 \cdots \alpha_n \rho_1 \cdots \rho_k \epsilon_1 \cdots \epsilon_m.\tau = \sigma \\
\qquad ([\tau_1, \cdots, \tau_n], [\rho'_1, \cdots, \rho'_k], [\epsilon'_1.\varphi'_1, \cdots, \epsilon'_m.\varphi'_m]) = il \\
\qquad\quad \text{(FAIL if } \sigma \text{ and } il \text{ have different arity)} \\
\qquad S = (S^t, S^r, S^e), \text{ where} \\
\qquad\quad S^t = \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\} \\
\qquad\quad S^r = \{\rho_1 \mapsto \rho'_1, \ldots, \rho_k \mapsto \rho'_k\} \\
\qquad\quad S^e = \{\epsilon_1 \mapsto \epsilon'_1.\varphi'_1, \ldots, \epsilon_m \mapsto \epsilon'_m.\varphi'_m\} \\
\qquad S^e_1 = \mathit{instTy}(S^e, S^r(S^t(\tau))) \\
\quad \text{in } (S^e_1, S^e_1(S^e(S^r(S^t\tau)))) \text{ end}
\end{array}
$$

Lemma 13 *Let S^e be a finite effect substitution. Assume $\vdash B$, $B \uplus B'$ exists, $B \uplus B' \vdash \tau$, $B \vdash Rng(S^e)$ and $\mathrm{Dom}(S^e) = \mathrm{Dom}(\widehat{\Phi'})$, where $\Phi' = \Phi$ of B' . Then $S_1^e = instTy(S^e, \tau)$ succeeds and*

1. *S_1^e is a contraction of B;*
2. *$S_1^e(B) \vdash (S_1^e \bullet S^e)(S_1^e(\tau))$;*
3. *$S_1^e(B) = B$ implies $S_1^e = \mathrm{Id}^e$.*

Similarly with instMu and μ for instTy and τ, and with instArrEff and $\epsilon.\varphi$ for instTy and τ.

Proof Write B in the form (Q, Φ) and write B' in the form (Q', Φ'). We first prove the result for *instArrEff* and $\epsilon.\varphi$ in place of *instTy* and τ. As in the definition of *instArrEff*, make the following case analysis.

$\boxed{\epsilon \notin \mathrm{Dom}(S^e)}$ Here $S_1^e = \mathrm{Id}^e$, which is a contraction of B. Moreover, since $B \uplus B'$ exists and $B \uplus B' \vdash \epsilon.\varphi$ and $\epsilon \notin \mathrm{Dom}(S^e)$ and $\mathrm{Dom}(S^e) = \mathrm{Dom}(\hat{\Phi}')$ we have $B \vdash \epsilon.\varphi$. Thus $(S_1^e \bullet S^e)(S_1^e(\epsilon.\varphi)) = S^e(\epsilon.\varphi) = \epsilon.\varphi$. (In short, if the instantiation substitution S^e does not have ϵ in its domain then it has no variable in φ in its domain.) Thus (2) and (3) hold.

$\boxed{\epsilon \in \mathrm{Dom}(S^e) \wedge S^e(\varphi) \subseteq \varphi_1}$ Here $S_1^e = \mathrm{Id}^e$, which is a contraction of B. Moreover, $(S_1^e \bullet S^e)(S_1^e(\epsilon.\varphi)) = S^e(\epsilon.\varphi) = \epsilon_1.(\varphi_1 \cup S^e(\varphi)) = \epsilon_1.\varphi_1$. Since $\epsilon_1.\varphi_1 \in Rng(S^e)$ and $B \vdash Rng(S^e)$ we thus have (2) as desired. Also (3) holds.

$\boxed{\epsilon \in \mathrm{Dom}(S^e) \wedge S^e(\varphi) \not\subseteq \varphi_1}$ Here $S_1^e = \{\epsilon_1 \mapsto \epsilon_1.(\varphi_1 \cup S^e(\varphi))\}$. Since $B \vdash \epsilon.\varphi$ and $B \vdash Rng(S^e)$ we have $B \vdash S^e(\varphi)$. Thus S_1^e is a contraction of B and since $S^e(\varphi) \not\subseteq \varphi_1$ we have $S_1^e(B) < B$. Thus (3) holds trivially. As for (2), we have

$$
\begin{array}{lll}
 & (S_1^e \bullet S^e)(S_1^e(\epsilon.\varphi)) & \\
= & S_1^e(S^e(\epsilon.\varphi)) & \text{since } Inv(S_1^e) \cap \mathrm{Dom}(S^e) = \emptyset \\
= & S_1^e(\epsilon_1.(\varphi_1 \cup S^e(\varphi))) & \\
= & \epsilon_1.(\varphi_1 \cup S^e(\varphi)) & \text{by the definition of } S_1^e
\end{array}
$$

Now $\epsilon_1.\varphi_1 \in \Phi$ so $S_1^e(\epsilon_1.\varphi_1) \in S_1^e(\Phi)$. Moreover, $S_1^e(\epsilon_1.\varphi_1) = \epsilon_1.(\varphi_1 \cup S^e(\varphi) \cup S_1^e(\varphi_1)) = \epsilon_1.(\varphi_1 \cup S^e(\varphi))$. Thus $S_1^e B \vdash (S_1^e \bullet S^e)(S_1^e(\epsilon.\varphi))$ as required.

We now proceed to the cases for *instTy* and *instMu*, which are done simultaneously, by structural induction on τ and μ. The only interesting case is the following:

$\boxed{\tau \equiv \mu_1 \xrightarrow{\epsilon.\varphi} \mu_2}$ Since $B \uplus B' \vdash \tau$ we have $B \uplus B' \vdash \mu_1$, $B \uplus B' \vdash \mu_2$ and $B \uplus B' \vdash \epsilon.\varphi$. Following the definition of $instTy(S^e, \tau)$ we have by induction that $S_1^e = instMu(S^e, \mu_1)$ succeeds and

$$S_1^e \text{ is a contraction on } B \tag{23}$$

$$S_1^e(B) \vdash (S_1^e \bullet S^e)(S_1^e \mu_1) \tag{24}$$

$$S_1^e(B) = B \text{ implies } S_1^e = \mathrm{Id}^e \tag{25}$$

By Lemma 4 on (23) and $B \uplus B' \vdash \mu_2$ we get $S_1^e(B \uplus B') \vdash S_1^e(\mu_2)$. Moreover, $S_1^e(B) \uplus S_1^e(B')$ exists and is equal to $S_1^e(B \uplus B')$. Moreover, $\mathrm{Dom}(S_1^e \bullet S^e) = \mathrm{Dom}(S^e) = \mathrm{Dom}(\widehat{\Phi_1'})$ and $S_1^e(B) \vdash Rng(S_1^e \bullet S^e)$, where $\Phi_1' = \Phi$ *of* $S_1^e(B')$. Thus by induction we have that $S_2^e = instMu(S_1^e \bullet S^e, S_1^e(\mu_2))$ succeeds and that

$$S_2^e \text{ is a contraction on } S_1^e(B) \tag{26}$$

$$S^e_{21}(B) \vdash (S^e_{21} \bullet S^e)(S^e_{21}\mu_2) \tag{27}$$

$$S^e_{21}(B) = S^e_1(B) \text{ implies } S^e_2 = \mathrm{Id}^e \tag{28}$$

where S^e_{21} is defined to be $S^e_2 \circ S^e_1$. (Here we use the general fact that for all substitutions S_1, S_2 and S_3, if S_1 is finite then $S_3 \bullet (S_2 \bullet S_1) = (S_3 \circ S_2) \bullet S_1$.) By (26) and (23) we get that $S^e_{21}(B)$ is consistent and that $S^e_{21}(B) \uplus S^e_{21}(B')$ exists and equals $S^e_{21}(B \uplus B')$. By Lemma 4 on $B \uplus B' \vdash \epsilon.\varphi$ and $B \vdash \mathit{Rng}(S^e)$ we have $S^e_{21}(B) \uplus S^e_{21}(B') \vdash S^e_{21}(\epsilon.\varphi)$ and $S^e_{21}(B) \vdash \mathit{Rng}(S^e_{21} \bullet S^e)$. Also $\mathrm{Dom}(S^e_{21} \bullet S^e) = \mathrm{Dom}(S^e) = \mathrm{Dom}(\widehat{\Phi'_2})$, where $\Phi'_2 = \Phi$ *of* $S^e_{21}(B')$. Thus by the first part of the lemma we have that $S^e_3 = \mathit{instArrEff}(S^e_{21} \bullet S^e, S^e_{21}(\epsilon.\varphi))$ succeeds and that

$$S^e_3 \text{ is a contraction of } S^e_{21}(B) \tag{29}$$

$$S^e_{321}(B) \vdash (S^e_{321} \bullet S^e)(S^e_{321}(\epsilon.\varphi)) \tag{30}$$

$$S^e_{321}(B) = S^e_{21}(B) \text{ implies } S^e_3 = \mathrm{Id}^e \tag{31}$$

where $S^e_{321} = S^e_3 \circ S^e_{21}$. Thus $\mathit{instTy}(S^e, \tau)$ succeeds with result S^e_{321}. By (23), (26) and (29) we get that S_{321} is a contraction of B, as required. Next, from (24) we get $S^e_{321}(B) \vdash (S^e_3 \circ S^e_2)((S^e_1 \bullet S^e)(S^e_1\mu_1))$, i.e., since $\mathit{Inv}(S^e_{321}) \cap \mathrm{Dom}(S^e) = \emptyset$,

$$S^e_{321}(B) \vdash (S^e_{321} \bullet S^e)(S^e_{321}(\mu_1)) \tag{32}$$

Similarly, from (27) we get

$$S^e_{321}(B) \vdash (S^e_{321} \bullet S^e)(S^e_{321}(\mu_2)) \tag{33}$$

By (30), (32) and (33) we get the desired $S^e_{321}(B) \vdash (S^e_{321} \bullet S^e)(S^e_{321}(\tau))$. Finally, assume $S^e_{321}(B) = B$. Then $S^e_{321}(B) = S^e_{21}(B) = S^e_1(B) = B$, so $S^e_{321} = \mathrm{Id}^e$, by (25), (28) and (31). □

Theorem 14 *Assume $B \vdash \sigma$ and $B \vdash il$ (by which we mean that $B \vdash e$ for each list element e of il). Also assume that σ and il have the same arity. Then $(S^e_1, \tau') = inst(\sigma, il)$ succeeds and*

1. *S^e_1 is a contraction of B;*
2. *$S^e_1(\sigma) \geq \tau'$ via $S^e_1(il)$ and $S^e_1(B) \vdash \tau'$;*
3. *$S^e_1(B) = B$ implies $S^e_1 = \mathrm{Id}^e$;*

Proof Let $B' = (Q', \Phi') = \mathit{bound}(\sigma)$ and let τ be as in the definition of *inst*. Since $B \vdash \sigma$ we have $\vdash B$ and $B \uplus B' \vdash \tau$. Since $B \vdash il$ we have that $B \uplus (\emptyset, S^r(\Phi'))$ exists and that $B \uplus (\emptyset, \widehat{S^r(\Phi')}) \vdash S^r(S^t\tau)$, using the declarations in the definition of *inst*. Also, $\mathrm{Dom}(S^e) = \mathrm{Dom}(\widehat{S^r(\Phi')})$ and $B \uplus (\emptyset, S^r(\Phi')) \vdash \mathit{Rng}(S^e)$. Thus by Lemma 13 we have that $S^e_1 = \mathit{inst}(S^e, S^r(S^t(\tau)))$ succeeds and that

$$S^e_1 \text{ is a contraction of } B \tag{34}$$

$$S^e_1(B) \vdash (S^e_1 \bullet S^e)(S^e_1(S^r(S^t(\tau)))) \tag{35}$$

$$S^e_1(B) = B \text{ implies } S^e_1 = \mathrm{Id}^e \tag{36}$$

Thus $\mathit{inst}(\sigma, il)$ succeeds with result $(S^e_1, \tau') = (S^e_1, S^e_1(S^e(S^r(S^t(\tau)))))$. Moreover, (34) and (36) are two of the desired results. Also, by (34) and the fact that $B \uplus B'$ exists, we have $\mathit{Inv}(S^e_1) \cap \mathrm{Dom}(S^e) = \emptyset$. Thus $(S^e_1 \bullet S^e)(S^e_1(S^r(S^t(\tau)))) = S^e_1(S^e(S^r(S^t(\tau)))) = (S^e_1 \bullet S^t, S^e_1 \bullet S^r, S^e_1 \bullet S^e)(S^e_1\tau)$. Furthermore, by (34) we have that $S^e(\sigma) = \forall \alpha_1 \cdots \alpha_n \rho_1 \cdots \rho_k \epsilon_1 \cdots \epsilon_m . S^e(\tau)$, without any renaming of bound variables being necessary. Thus $S^e_1(\sigma) \geq \tau'$ *via* $S^e_1(il)$ and $S^e_1(B) \vdash \tau'$ follow from (35). □

12 Generalisation of Region and Effect Variables

In this section, we describe how to form type schemes from types. In order to ensure that the produced type schemes are consistent, an operation called *Below* is used.

12.1 The *Below* Operation

Let $B = (Q, \Phi)$ be a consistent basis and let $B_0 = (Q_0, \Phi_0)$ be a basis with $B_0 \subseteq B$ — by which we mean $Q_0 \subseteq Q$ and $\Phi_0 \subseteq \Phi$. Clearly, Φ_0 is functional and transitive, but it need not be closed. Also, we do not necessarily have $Q_0 \supseteq \mathit{frv}(\Phi_0)$. Thus B_0 is not necessarily consistent. However, there is a smallest basis (with respect to $\subseteq$), denoted $\mathit{Below}(B, B_0)$, which is consistent and satisfies $B \supseteq \mathit{Below}(B, B_0) \supseteq B_0$. It is defined thus: let Φ_1 be the smallest closed subset of Φ satisfying $\Phi_1 \supseteq \Phi_0$. (This exists since B is consistent and $B_0 \subseteq B$.) Further, let Q_1 be $Q_0 \cup \mathit{frv}(\Phi_1)$. Then $\mathit{Below}(B, B_0)$ is defined to be (Q_1, Φ_1).

12.2 The *RegEffClos* operation

Damas and Milner[3] define a closure operation, let us write it $\mathit{Clos}_{TE}(\tau)$, which forms a type scheme from a type τ and a type environment TE. We now define the corresponding operation for forming region-annotated type schemes. Let τ be a type and let φ_1 be an effect. Further, let B_0 and B be bases with $B \supseteq B_0$ and $B \vdash \tau$. (The basis B_0 contains region and effect variables that must not be quantified, for example because they occur free in the type environment.)

1 $\quad \mathit{RegEffClos}(B, B_0, \varphi)(\tau) =$
2 $\quad\quad$ let
3 $\quad\quad\quad (Q, \Phi) = B$ and $(Q_0, \Phi_0) = B_0$
4 $\quad\quad\quad Q_0' = Q_0 \cup \mathit{frv}(\varphi) \cup \mathit{frv}(\tau) \setminus \mathit{pfrv}(\tau)$
5 $\quad\quad\quad \Phi_0' = \Phi_0 \cup \{\epsilon.\varphi' \in \Phi \mid \epsilon \in (\varphi \cup (\mathit{fev}(\tau) \setminus \mathit{pfev}(\tau)))\}$
6 $\quad\quad\quad B_1$ as $(Q_1, \Phi_1) = \mathit{Below}(B, (Q_0', \Phi_0'))$
7 $\quad\quad\quad \{\epsilon_1.\varphi_1, \ldots, \epsilon_m.\varphi_m\} = \Phi \setminus \Phi_1$
8 $\quad\quad\quad \{\rho_1, \ldots, \rho_k\} = Q \setminus Q_1$
9 $\quad\quad\quad \sigma = \forall \rho_1 \cdots \rho_k\, \epsilon_1 \cdots \epsilon_m.\tau$
10 $\quad$ in
11 $\quad\quad (B_1, \sigma)$
12 $\quad$ end

Here Q_0' and Φ_0' contain the region and effect variables that must not be generalised. In addition to the variables in B_0, Q_0' and Φ_0' include all region and effect variables in φ and secondary region and effect variables in τ (see lines 4 and 5). In lines 7 and 8 we determine the set of region and effect variables that are generalised. The generalisation takes place in line 9.

For every type τ and basis B with $B \vdash \tau$, we write $\mathit{Below}(B, (B_0, \tau))$ as a shorthand for $\mathit{Below}(B, (B_0 \cup (\mathit{frv}(\tau), \mathit{arreff}(\tau))))$.

Lemma 15 *Let $B \vdash \tau$, $B \vdash \varphi$, let B_0 be a basis and assume $B_0 \subseteq B$ and $B = \mathit{Below}(B, (B_0, \tau))$. Then $(B_1, \sigma) = \mathit{RegEffClos}(B, B_0, \varphi)(\tau)$ succeeds and $B_1 \vdash \sigma$ and $B_1 \vdash \varphi$ and $B \supseteq B_1 \supseteq B_0$.*

Proof The call $(B_1, \sigma) = \mathit{RegEffClos}(B, B_0, \varphi)(\tau)$ clearly succeeds and $B \supseteq B_1 \supseteq B_0$ follows from the definition of *Below*. From $B = \mathit{Below}(B, (B_0, \tau))$ and the definition of Q_0' and Φ_0' we get $\{\rho_1, \ldots, \rho_k\} \subseteq \mathit{pfrv}(\tau)$ and $\{\epsilon_1, \ldots, \epsilon_m\} \subseteq \mathit{pfev}(\tau)$. Let $B' = (Q \setminus Q_1, \Phi \setminus \Phi_1)$. Then $B' = \mathit{bound}(\sigma)$. Also, $B_1 \uplus B'$ exists and is equal to B, which is consistent. Also $B \vdash \tau$. Thus we have $B_1 \vdash \sigma$, according to Definition 10. Also, $B_1 \vdash \varphi$ holds by the definition of Q_0' and Φ_0'. $\square$

An example of the use of *RegEffClos* was provided in Example 10.1.

13 Conclusion

A comparison between the type inference rules of Milner's polymorphic type discipline[3] and the region inference rules in Section 4 leads to the conclusion that upon something simple, something complicated will inevitably be erected. In our experience, however, Milner's type discipline carries even fairly heavy load well and we think that it is fascinating that it can be used for reasoning not just about the soundness of programs but also for reasoning about memory management.

References

[1] A. Aiken, M. Fähndrich, and R. Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Proc. of the ACM SIGPLAN '95 Conference on Programming Languages and Implementation (PLDI)*, pages 174–185, La Jolla, CA, June 1995. ACM Press.

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

[3] 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.

[4] Y. Fuh and P. Mishra. Type inference with subtypes. *Theoretical Computer Science (TCS)*, 73:155–175, 1990.

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

[6] R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc.*, 146:29–60, Dec. 1969.

[7] A. Kfoury, J. Tiuryn, and P. Urzyczyn. The undecidability of the semi-unification problem. *Information and Computation*, pages 83–101, 1993.

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

[9] X. Leroy. *Typage polymorphe d'un langage algorithmique*. PhD thesis, University Paris VII, 1992. English version: Polymorphic Typing of an Algorithmic Language, INRIA Research Report no. 1778, October 1992.

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

[11] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[12] J. Mitchell. Coercion and type inference. In *Proc. 11th ACM Symp. on Principles of Programming Languages (POPL)*, 1984.

[13] F. Nielson, H. R. Nielson, and T. Amtoft. Polymorphic subtyping for effect analysis: the algorithm. Technical Report LOMAPS-DAIMI-16, Department of Computer Science, University of Aarhus (DAIMI), April 1996.

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

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

[16] 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.

[17] J. Robinson. A machine-oriented logic based on the resolution principle. *J. Assoc. Comput. Mach.*, 12(1):23–41, 1965.

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

[19] J.-P. Talpin and P. Jouvelot. The type and effect discipline. In *Proceedings of the seventh IEEE Conference on Logic in Computer Science*, pages 162–173, June 1992. Also, (extended version) technical report EMP/CRI/A-206, Ecole des Mines de Paris, April 1992.

[20] M. Tofte and L. Birkedal. A region inference algorithm. (Submitted for publication; available at `http://www.diku.dk/users/tofte/publ/toplassubm.a4.ps.Z`), November 1996.

[21] M. Tofte, L. Birkedal, M. Elsman, N. Hallenberg, T. H. Olesen, P. Sestoft, and P. Bertelsen. Programming with regions in the ML Kit. Technical Report DIKU-TR-97/12, Department of Computer Science, University of Copenhagen, 1997. (http://www.diku.dk/research-groups/topps/activities/topps/kit2).

[22] M. Tofte and J.-P. Talpin. Data region inference for polymorphic functional languages (technical summary). Technical Report EMP/CRI/A-229, Ecole des Mines de Paris, 1992.

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

[24] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

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