A Region Inference Algorithm

MADS TOFTE
University of Copenhagen
and
LARS BIRKEDAL
Carnegie Mellon University

Region Inference is a program analysis which infers lifetimes of values. It is targeted at a runtime model in which the store consists of a stack of regions and memory management predominantly consists of pushing and popping regions, rather than performing garbage collection. Region Inference has previously been specified by a set of inference rules which formalize when regions may be allocated and deallocated. This article presents an algorithm which implements the specification. We prove that the algorithm is sound with respect to the region inference rules and that it always terminates even though the region inference rules permit polymorphic recursion in regions. The algorithm is the result of several years of experiments with region inference algorithms in the ML Kit, a compiler from Standard ML to assembly language. We report on practical experience with the algorithm and give hints on how to implement it.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*dynamic storage management*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*functional constructs*; *type structure*

General Terms: Algorithms, Reliability, Theory

Additional Key Words and Phrases: Regions, Standard ML

1. INTRODUCTION

Most programming languages have some mechanism for allocation and deallocation of dynamic data structures. In the Algol stack discipline [Dijkstra 1960; Naur 1963], every point of allocation is matched by a point of deallocation; these points are easily identified in the program text, which helps reasoning about memory usage.

However, the pure stack discipline is rather restrictive. For example, it applies neither to recursive data types nor to higher-order call-by-value functions. Therefore, most languages have additional facilities for memory management. In C, the programmer must explicitly allocate and free heap memory (using `malloc` and

Work supported by the Danish Research Council and by U.S. National Science Foundation Grant CCR-9409997.
Authors' addresses: M. Tofte, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark email tofte@diku.dk; L. Birkedal, School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213; email: birkedal@cs.cmu.edu.

`free`). Allocation is easy, but in general, it is hard to know when to deallocate memory. If memory is deallocated too soon, the program may crash because of a pointer into a location which has been reused for a different purpose. If deallocation is too conservative, the program may eventually use up all the memory resources; in colloquial terms, it may "leak memory." Both phenomena are well known in practice.

Finally, in languages such as Standard ML and Java, programs allocate memory for storing values and objects. A separate subroutine in the runtime system, the garbage collector, periodically reclaims memory for use by the program, based on a traversal of some or all the values stored in memory. (For an excellent review of garbage collection techniques, see Wilson [1992]. Compared to the above two schemes, garbage collection restores safety, but not predictability.

For many applications, programmer control over both time and memory resources is indispensable.

Region Inference is a static program analysis which estimates lifetimes of values. It is targeted at a runtime model in which the store consists of a stack of regions. At runtime, all values (e.g., records, lists, references, and function closures) are put into regions. The fundamental difference from (traditional) garbage collection techniques is that all decisions about where to allocate and deallocate regions are made *statically*, by region inference; furthermore, for every value-producing expression, region inference decides into which region the value is to be put.

Region Inference has been developed for the typed call-by-value lambda calculus [Tofte and Talpin 1994; 1997] and it is used in the ML Kit with Regions [Tofte et al. 1997], a compiler from Standard ML [Milner et al. 1997] to HP PA-RISC assembly code.

There are several reasons why region inference is attractive. First, region inference restores the coupling of allocation and deallocation; after region inference, lifetimes are explicit as block structure in the program, making it easier to reason about memory usage.

Second, in some cases, the compiler can prove the absence of memory leaks or warn about the possibility of memory leaks [Tofte et al. 1997]. There are also cases where region inference can allow deallocation of regions into which there are still pointers, because it can prove that these references will never be followed. Reference-tracing garbage collectors are more conservative in these cases: they assume that data reachable from live data must themselves be live.

Third, when region inference is combined with other analyses [Birkedal et al. 1996], one can obtain very compact and reasonably fast object programs. Modified versions of benchmarks from the Standard ML of New Jersey benchmark suite (including Knuth-Bendix completion, Simple, Life, and Mandelbrot) currently use less than 1Mb of resident memory after compilation with the ML Kit [Hallenberg 1997]. Running times for object programs produced by the ML Kit vary between four times as slow and ten times as fast as running times of the same programs compiled with Standard ML of New Jersey [Birkedal et al. 1996]. While such comparisons do not give any deep insight, they do demonstrate that region technology is competitive with other existing implementation technology.

Fourth, the memory allocation primitives, which the analysis inserts, are each constant time operations at runtime. The fact that there are no interruptions of unbounded duration makes regions attractive for real-time programming and makes it easier to reason about time usage.

Finally, the program transformations which are useful for region inference often turn out to be useful for garbage collection too. Thus one can use region inference as a development tool for decreasing memory consumption, even if one prefers using garbage collection.

The main limitation of region inference is that, in some cases, it estimates lifetimes conservatively, compared to the lifetimes a dynamic garbage collector yields. However, in our experience, this deficiency can usually be circumvented through minor changes to source programs and use of a region profiler [Tofte et al. 1997].

Another possible concern about region inference is separate compilation. In order to perform region inference of a module, one needs region type schemes for the identifiers imported into the module. However, it is possible to integrate region inference with separate compilation of full Standard ML Modules (including functors) in a manner which avoids recompilation of a module as long as the region type schemes of its free identifiers have not changed. (Martin Elsman has developed this scheme for the ML Kit.)

In order to use regions effectively, one has to understand the region inference rules. Here it is important to distinguish between the *specification* and the *implementation* of region inference. The specification of region inference is a set of formal inference rules [Tofte and Talpin 1994; 1997] which admit many different results of the analysis. Indeed, the rules permit putting all values in one global region which is never deallocated. The soundness of these rules with respect to a dynamic semantics is proved in Tofte and Talpin [1997].

An implementation of region inference is an algorithm which is sound with respect to the region inference rules. It is an open problem whether Damas and Milner's result about the existence of principal types [Milner 1978; Damas and Milner 1982] can be extended to the region inference rules of Tofte and Talpin [1994]. This article proves the soundness and termination of one particular region inference algorithm. The ML Kit with Regions uses this algorithm (extended to the whole of Core Standard ML). The algorithm does not always infer principal types. The incompleteness of the algorithm has to do with the handling of polymorphic recursion. While the incompleteness is aesthetically displeasing, it is not of crucial importance. First, the situations where incompleteness arise are somewhat pathological. Second, while principality in ML influences which programs are accepted by the compiler (and is therefore important), region analysis is merely an "internal" static analysis which makes no claim of being optimal in terms of runtime memory consumption. From a practical point of view, other program analyses that supplement region inference [Aiken et al. 1995; Birkedal et al. 1996] are at least as important as principality.

The input programs to our algorithm do not contain region annotations. This makes it possible to apply region inference to Standard ML, which is defined without any reference to regions. This is good for portability, but decreases user control over regions.

At the other extreme, one could demand that users provide all region information explicitly, so that region inference is reduced to region checking. There are several problems with this, however. First, there is a great deal of region and effect information in region-annotated programs, at least if one uses the present region inference rules. Providing all region and effect information explicitly would be tedious. Second, region and effect inference is sensitive to relatively minor program changes. Thus it would probably be much more work to keep an explicitly region-annotated program "region correct" than it is to keep ML programs type correct. Finally, the region inference algorithm seems to do a good job of inferring regions for the vast majority of allocation points in the program; it seems better, therefore, to let the region inference algorithm do most of the inference and then provide region profiling tools which allow the programmer to identify program points which cause memory leaks.

It would be useful if the programmer could write incomplete region information, for example, to ensure that any future program changes which invalidate carefully tuned region manipulations are caught automatically. This would probably be the best solution if one were to design a new programming language specifically for region-based memory management. However, our present concern is to develop region inference for a programming language which is already in use.

2. RELATED WORK

Our region inference algorithm is based on unification of types that contain effects. Effects are sets, not terms, so effect equality is a larger relation than equality of the terms that represent effects. Similar situations have been studied for other type systems. Typically, terms which represent finite sets or finite maps are paired with some kind of variable which identifies the set or map in question. This is the situation in the semantics of ML structure sharing [Harper et al. 1987; Tofte 1988; Aponte 1993] (where structure names are paired with environments), in record typing [Rémy 1989] (where row variables are paired with record types) and in type systems for polymorphic references [Leroy 1992] (where closure type variables are paired with sets of type schemes). In this article, effect variables are paired with effects. The systems vary somewhat, for example, in their treatment of types with "cycles" and in their notions of substitution, but in all cases, some notion of what we call consistency is important for the existence of most general unifiers.

Another possibility is to reduce the inference problem to finding a solution to a set of constraints extracted from the program. This has been done in work on region and effect inference [Talpin 1993; **?**; **?**; Tofte and Talpin 1992], and in work on partial type inference for ML extended with polymorphic recursion [Henglein 1993]. Henglein reduces type inference to the problem of semi-unification. Semi-unification is undecidable [Kfoury et al. 1990], but Henglein [1993] gives a semi-algorithm for semi-unification. The region inference rules allow a *limited* form of polymorphic recursion; we are currently working on a constraint-based region inference algorithm, which is based on Henglein's semi-algorithm but is extended to deal with effects. The restrictions on type schemes that we introduce here are used in essential ways both in the region algorithms contained in the present article and in the evolving constraint-based approach.

Nielson et al. [Nielson and Nielson 1994; Nielson et al. 1996] use constraint sets for inferring effects in CML programs. Their system has subtyping; the Tofte-Talpin rules do not. Conversely, the Tofte-Talpin rules have polymorphic recursion in regions and effects, which the CML rules do not. Still, there are similarities in the approaches to type inference; for example, in both systems it was found (independently) that the notion of (what we call) well-formed type schemes is important for dealing with polymorphism.

The basic ideas of region inference are described in Tofte and Talpin [1994]. A soundness proof for region inference is presented in Tofte and Talpin [1997]. Other analyses which can accompany region inference are described in Aiken et al. [1995] and Birkedal et al. [1996]. Sections 4 to 6 of the present article are essentially a summary of Tofte and Birkedal [1998], in which properties about the basic operations employed in this article (such as unification) are stated and proved. The contribution of this article is the region inference algorithm itself and a proof that it is correct and always terminates. The main technical difficulty in proving termination stems from the presence of a limited form of polymorphic recursion in the region inference rules. A region inference algorithm for an early version of region inference (without polymorphic recursion) may be found in Tofte and Talpin [1992].

3. REGION-ANNOTATED PROGRAMS

The input to the region inference algorithm is a source program which has already been type-checked. The source language is the explicitly typed call-by-value lambda calculus with ML style polymorphism and recursive functions. The restriction to call-by-value is used in essential ways in the formulation of the region inference rules. The rules could probably be modified to model call-by-name, but lazy evaluation seems difficult to model accurately, since it is not clear how to distinguish the first use of a variable from the rest. Assume a denumerably infinite set TyVar of type variables. We use α to range over TyVar. We now define types and type schemes as follows:

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

We use underlining for all constructs that have to do with the source language. The set of ML types is denoted MLType. The set of ML type schemes is denoted MLTypeScheme. Throughout, we require the bound variables of a type scheme to be distinct and we abbreviate $\forall\alpha_1.\cdots.\forall\alpha_n.\underline{\tau}$ to $\forall\alpha_1\cdots\alpha_n.\underline{\tau}$.

Next, assume a denumerably infinite set Var of *program variables*, ranged over by x and f. The language TypedExp of *explicitly typed source expressions* is defined by the following grammar:

$$\begin{array}{lll} \underline{e} ::= x\texttt{[}\underline{\tau}_1\texttt{,}\ldots\texttt{,}\underline{\tau}_n\texttt{]} & & (n \geq 0) \\ \quad \mid \lambda x : \underline{\tau}.\underline{e} \mid \underline{e}_1\,\underline{e}_2 & & \\ \quad \mid \texttt{letrec}\ f : \underline{\sigma}(x)\ \texttt{=}\ \underline{e}_1\ \texttt{in}\ \underline{e}_2\ \texttt{end} & & \end{array}$$

Notice that this language has a recursive `letrec`, rather than Milner's original `let` [Milner 1978]. It is possible to treat `let` as well (we have done so for value polymorphism), but it adds nothing new, compared to `letrec`. Since `letrec` is interesting in that it admits polymorphic recursion in regions (which `let` does not), we only consider `letrec` in the formal development, although we use `let` in examples.

The type system for TypedExp appears in Figure 1. A *substitution* is a map from type variables to types. A type $\underline{\tau}_0$ is an *instance* of a type scheme $\underline{\sigma} = \forall\alpha_1\cdots\alpha_n.\underline{\tau}$, written $\underline{\sigma} \geq \underline{\tau}_0$, if there exist $\underline{\tau}_1,\ldots,\underline{\tau}_n$ such that $S(\underline{\tau}) = \underline{\tau}_0$, where S

$$\frac{\underline{TE}(x) = \forall\alpha_1\cdots\alpha_n.\underline{\tau} \qquad S = \{\alpha_1 \mapsto \underline{\tau}_1, \ldots, \alpha_n \mapsto \underline{\tau}_n\}}{\underline{TE} \vdash x\,[\underline{\tau}_1,\ldots,\underline{\tau}_n] : S(\underline{\tau})} \tag{1}$$

$$\frac{\underline{TE} + \{x \mapsto \underline{\tau}\} \vdash \underline{e} : \underline{\tau}_1}{\underline{TE} \vdash \lambda x : \underline{\tau}.\underline{e} : \underline{\tau} \to \underline{\tau}_1}$$

$$\frac{\underline{TE} \vdash \underline{e}_1 : \underline{\tau}_2 \to \underline{\tau} \qquad \underline{TE} \vdash \underline{e}_2 : \underline{\tau}_2}{\underline{TE} \vdash \underline{e}_1\ \underline{e}_2 : \underline{\tau}}$$

$$\frac{\begin{array}{l}\underline{\sigma} = \forall\alpha_1\cdots\alpha_n.\underline{\tau}_f \qquad \{\alpha_1,\ldots,\alpha_n\} \subseteq \mathit{ftv}(\underline{\tau}_f) \setminus \mathit{ftv}(\underline{TE})\\ \underline{TE} + \{f \mapsto \underline{\tau}_f\} \vdash \lambda x : \underline{\tau}_x.\underline{e}_1 : \underline{\tau}_f \quad \underline{\tau}_f = \underline{\tau}_x \to \underline{\tau}_1\\ \underline{TE} + \{f \mapsto \underline{\sigma}\} \vdash \underline{e}_2 : \underline{\tau}\end{array}}{\underline{TE} \vdash \texttt{letrec}\ f : \underline{\sigma}(x)\ \texttt{=}\ \underline{e}_1\ \texttt{in}\ \underline{e}_2\ \texttt{end} : \underline{\tau}} \tag{2}$$

Fig. 1. Type system for TypedExp. $\mathit{ftv}(\underline{TE})$ means the set of type variables that occur free in $\underline{TE}$.

is the substitution $\{\alpha_1 \mapsto \underline{\tau}_1, \ldots, \alpha_n \mapsto \underline{\tau}_n\}$. An *ML type environment*, $\underline{TE}$, is a finite map from program variables to ML type schemes. In the `letrec` construct (rule 2), the bound variables of $\underline{\sigma}$ bind over *both* $\underline{\tau}_f$ and $\underline{e}_1$. Also note that non-binding occurrences of program variables are annotated with types (rule 1). These explicitly typed terms can be thought of as the result of static elaboration of Milner's implicitly typed terms.

The output from the region inference algorithm is a *typed region-annotated term*, which is identical to the input term, except that every node in the term has been decorated with region and effect information. Typed region-annotated terms contain more information than is needed at runtime. This extra information is used by the region inference algorithm.

If one erases this extra information from typed region-annotated terms one obtains *untyped region-annotated terms*, defined as follows. We assume a denumerably infinite set

$$\mathrm{RegVar} = \{\rho_1, \rho_2, \ldots\}$$

of *region variables*; we use ρ to range over region variables. The grammar for the language of untyped region-annotated terms, call it RegionExp, is

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

Region variables stand for runtime regions. The "$\texttt{at}\,\rho$" annotation is found wherever a value is produced and indicates into which region the value should be put. For example, in $\lambda x.e$ `at` ρ, the ρ indicates where the closure representing the function should be put. The phrase `letregion` ρ `in` e `end` introduces ρ and binds it in e. The meaning in the dynamic semantics is: allocate a new region on the region stack, bind it to ρ, evaluate e (presumably using ρ), then (upon reaching `end`) pop the region stack, thereby removing a whole region in one operation.

```
P ≡ λh :(int → int).
        letrec g : int → (int → int)(a : int)=
          if a > 0 then
            let b : int= a − 1
            in λx : int.g(b − 1)x
            end
          else h
        in (g 5) 10
        end
```

Fig. 2. The source program P.

```
(λh. letrec g[ρ5] at ρ6(a)=
       if a > 0 then
         let b = (a − 1) at ρ4
         in
           (λx.letregion ρ7, ρ8
                in g[ρ7] at ρ8 (b − 1) at ρ7
                end x) at ρ3
         end
       else h
     in
       letregion ρ9, ρ10
       in g[ρ9] at ρ10 5 at ρ9
       end 10 at ρ1
     end
) at ρ0
```

Fig. 3. A region-annotated version of P.

The `letrec`-construct binds $\rho_1, \ldots, \rho_k$ in e_1. The $\rho_1, \ldots, \rho_k$ must be distinct; at runtime they serve as formal parameters to f. Function f resides in ρ. Region inference turns f into a "curried" function: applying f to actual regions $\rho'_1, \ldots, \rho'_k$ yields a function which may then be applied to an ordinary value argument. This is expressed by the phrase form f `[`ρ'_1`, ⋯, `ρ'_k`] at` ρ', where ρ' is the place where the closure created by the partial application is put.

3.1 An Example

Consider the source program P defined in Figure 2. An application $P\,h'$ will call g recursively three times (using values 5, 3, 1, and -1 for a) upon which h' will be applied to 10.

The output of the region inference algorithm is shown in Figure 3. Here ρ_5 is the region of a. Notice that g calls itself using a different region, ρ_7, which is local to the body of g. (This is obtained through use of the polymorphic recursion in regions.) Some region variables are free in the above expression (ρ_0, ρ_1, ρ_3, ρ_4, and ρ_6). These denote regions which must be introduced by the context of the expression. (At top level, free region variables are interpreted as global regions.)

Normally, some of the formal region parameters of a region-polymorphic function indicate where the function should put its result. In the above example, however, the conditional forces g to place $\lambda x.g(b-1)x$ in the same region as h resides in, i.e., ρ_3. When applied, $\lambda x.g(b-1)x$ will access b. Since the analysis treats h and $\lambda x.g(b-1)x$ as the same and since h is fixed (formally, h is lambda-bound) at the point where g is declared, the region of $a-1$ cannot become a formal parameter of g. Thus $a-1$ is stored in yet another free region, ρ_4. As a consequence, an application of the example program to some particular function, f, will write two values (one in ρ_4 and one in ρ_3) for each iteration of g, before finally applying f which puts its result in some completely different region, ρ_2, say. Thus the allocations into ρ_3 and ρ_4 are potential memory leaks. The ML Kit prints the following warning

> g has a type scheme with escaping put effects on region(s):
> ρ_3, which is also free in the type (schemes) of : h
> ρ_4, which is also free in the type (schemes) of : h

but compiles the program. In this example, the analysis discovered a "side-effect" at the region level for a purely functional program, but the Kit uses the same principle (and the same warnings) for effects involving ML references. For example, a function which creates a record, stores it in a nonlocal reference, and then returns an integer, will result in a warning. In both cases, warnings are useful for controlling space usage.

4. EFFECTS AND REGION-ANNOTATED TYPES

Region inference manipulates types in which every type constructor has been paired with a region variable. For example, a legal type of h in the example (Section 3.1) is

$$((\texttt{int}, \rho_1) \xrightarrow{\epsilon_1.\varphi_1} (\texttt{int}, \rho_2), \rho_3)$$

where ρ_1 is the region of the argument to h; ρ_2 is the region of the result of h; and ρ_3 is the region where h itself resides. (The annotation on the arrow will be described below.)

Also inferred for each expression is an *effect*. In the original work on effect systems, the term effect relates to references and side-effects [Lucassen and Gifford 1988; Jouvelot and Gifford 1991; Talpin and Jouvelot 1992b]. We use the word effect in a different sense, namely, any read or write in a region (irrespective of whether the region contains references). The effects make it possible to find out where `letregion` can safely be introduced. Putting a value into region ρ is represented by the atomic effect $\mathbf{put}(\rho)$. Accessing a value in region ρ is represented by the atomic effect $\mathbf{get}(\rho)$. An effect (of a region-annotated expression) is a finite set of atomic effects. Somewhat simplified, the rule for introducing `letregion` is

$$\frac{TE \vdash e' : \mu, \varphi \qquad \rho \text{ not free in } TE \text{ or } \mu}{TE \vdash \texttt{letregion}\ \rho\ \texttt{in}\ e'\ \texttt{end} : \mu, \varphi \setminus \{\mathbf{get}\,\rho, \mathbf{put}\,\rho\}}. \tag{3}$$

In other words, if region variable ρ occurs free neither in type environment TE nor in the region-annotated type (μ) of e' (although ρ may well occur in the effect φ of e'), then ρ is purely local to the evaluation of e'.

Function types take the form $\mu \xrightarrow{\epsilon.\varphi} \mu'$ where the object $\epsilon.\varphi$ is called an *arrow effect*. Here φ is an effect and ϵ is an *effect variable*. Formally, $\epsilon.\varphi$ is just a shorthand for the pair (ϵ, φ). We arrange that effect variables label effects in such a way that no label labels two different sets (Section 5.1). In an arrow effect $\epsilon.\varphi$, one can then think of ϵ as labeling φ. If a function f has type $\mu \xrightarrow{\epsilon.\varphi} \mu'$, then φ is referred to as the *latent* effect of f; it is the effect of evaluating the body of f. In a higher-order language, like the one we are considering, an expression of functional type can evaluate to different functions at runtime. We cannot distinguish functions statically, but we can use a common effect variable and a common latent effect to represent all the functions to which the expression can evaluate.

In Example 3.1, we can compute the latent effect of h as follows. Function h and $\lambda x.g\,(b-1)\,x$ are treated as equal, since they are both results of the same conditional. The function $\lambda x.g\,(b-1)\,x$ is evaluated as follows, if applied. First get b from ρ_4, get g from ρ_6, then call g, which either returns h or puts b in ρ_4 and a closure in ρ_3; next, get the closure (produced by $g(b-1)$) from ρ_3 and apply it to x; this application has the same effect as the latent effect of h, which is the effect we are trying to find. Thus the latent effect h is

$$\varphi_1 = \{\mathbf{get}(\rho_4), \mathbf{get}(\rho_6), \mathbf{put}(\rho_4), \mathbf{put}(\rho_3), \mathbf{get}(\rho_3)\}.$$

Unlike region variables, effect variables play no role at runtime. However, they play an important role in our region inference algorithm. P illustrates this. P, if applied, puts a closure for g into ρ_6, gets the same closure from ρ_6, and calls g (which either returns h or puts a b into ρ_4 and a closure into ρ_3), returning a function in ρ_3. Then 10 is put into ρ_1 and the function in ρ_3 applied. This application we represent by effect variable ϵ_1 (the handle of the effect variable of h) unioned with the latent effect of h, i.e., φ_1. To sum up, P has type

$$(((\mathtt{int}, \rho_1) \xrightarrow{\epsilon_1.\varphi_1} (\mathtt{int}, \rho_2), \rho_3) \xrightarrow{\epsilon_2.\varphi_2} (\mathtt{int}, \rho_2), \rho_0) \tag{4}$$

where $\varphi_2 = \{\mathbf{put}(\rho_6), \mathbf{get}(\rho_6), \mathbf{put}(\rho_4), \mathbf{put}(\rho_3), \mathbf{put}(\rho_1), \epsilon_1, \mathbf{get}(\rho_3), \mathbf{get}(\rho_4)\}$. Note that ϵ_1 is a member of φ_2. It stands for the fixed, but unknown, latent effect of h.

Now assume that P is part of the larger program

$$\mathtt{val}\ n = P(\lambda x.x + 1) \tag{5}$$

and that the effects computed above are really only the results of the analysis of the subexpression P. How are the effects we computed for P affected by the fact that we now know what h is?

Note that the computation of $P(\lambda x.x + 1)$ eventually applies the successor function, so the effect of $P(\lambda x.x + 1)$ should contain (as a subset) the latent effect of the successor function.

Assume that the result of analyzing $\lambda x.x + 1$ is

$$\begin{aligned}&(\lambda x : (\mathtt{int}, \rho_{12}).(x+1)\ \mathtt{at}\ \rho_{13})\ \mathtt{at}\ \rho_{14} : \\ &\quad ((\mathtt{int}, \rho_{12}) \xrightarrow{\epsilon_{14}.\{\mathbf{get}(\rho_{12}),\mathbf{put}(\rho_{13})\}} (\mathtt{int}, \rho_{13}), \rho_{14})\ .\end{aligned}$$

This type is now unified with the argument type of P. Thus ρ_{12} is mapped to ρ_1; ρ_{13} is mapped to ρ_2; and ρ_{14} is mapped to ρ_3. At this stage the latent effect of the successor function is $\{\mathbf{get}(\rho_1), \mathbf{put}(\rho_2)\}$. Continuing unification, ϵ_1 and ϵ_{14} are unified and the effects they denote unioned. Intuitively, the effect which ϵ_1 denotes has to be increased from φ_1 to $\varphi_1 \cup \{\mathbf{get}(\rho_1), \mathbf{put}(\rho_2)\}$ now that we know that the latent effect of h has to be at least as big as the latent effect of the successor function. Formally, we represent this effect increase by the "effect substitution" which maps

ϵ_1 to the arrow effect $\epsilon_1.\{\mathbf{get}(\rho_1), \mathbf{put}(\rho_2)\}$, read: map ϵ_1 to itself and increase the effect it denotes by $\{\mathbf{get}(\rho_1), \mathbf{put}(\rho_2)\}$. The unifying effect substitution maps ϵ_{14} to $\epsilon_1.\varphi_1$: it maps ϵ_{14} to ϵ_1 and unions the effect which ϵ_{14} stood for with φ_1. The unifying substitution, call it S, maps every other variable (e.g., ϵ_2) to itself. But ϵ_2 stands for an effect which contains ϵ_1, so if the effect that ϵ_1 has to increase, so does the effect that ϵ_2 stands for. Using the definition of substitution given below one has

$$\begin{aligned} S(\epsilon_2.\varphi_2) &= S(\epsilon_2.(\{\epsilon_1\} \cup (\varphi_2 \setminus \{\epsilon_1\}))) \\ &= \epsilon_2.(\{\epsilon_1, \mathbf{get}(\rho_1), \mathbf{put}(\rho_2)\} \cup S(\varphi_2 \setminus \{\epsilon_1\})) \\ &= \epsilon_2.(\{\mathbf{get}(\rho_1), \mathbf{put}(\rho_2)\} \cup \varphi_2)\,. \end{aligned}$$

Thus unification has taken care of propagating the latent effect of the successor function to the latent effect of P. We can compute the effect φ of $P(\lambda x.x+1)$: store P in ρ_0; store the successor function in ρ_3; access P in ρ_0; and then do everything recorded in the latent effect of P:

$$\varphi = \{\mathbf{put}(\rho_0), \mathbf{put}(\rho_3), \mathbf{get}(\rho_0), \epsilon_1, \epsilon_2, \mathbf{get}(\rho_1), \mathbf{put}(\rho_2)\} \cup \varphi_2. \tag{6}$$

We note that the effect mentions both ρ_1 and ρ_2, as it should.

To sum up, effect variables and effect substitutions are used in the algorithm in a manner which strongly resembles the way type variables and type substitutions are used in Milner's algorithm W. The main difference is that we have to deal with types that contain sets (the effects) and that unification therefore is not ordinary first-order term unification.

4.1 Definitions

We assume a denumerably infinite set

$$\text{EffVar} = \{\epsilon_1, \epsilon_2, \ldots\}$$

of *effect variables*, ranged over by ϵ. The sets TyVar, RegVar, and EffVar are assumed to be pairwise disjoint.

Although the distinction between **put** and **get** allows optimizations in later stages of compilation [Birkedal et al. 1996], it is not of great consequence for the region inference algorithm. Henceforth, an *atomic effect* is simply a region variable or an effect variable; an *effect* is a finite set of atomic effects:

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

The *handle* of an arrow effect $\epsilon.\varphi$ is ϵ. Annotated types are given by

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

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. 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.

Erasure of region and effect information from annotated types yields ML types; it is defined recursively by

$$\mathrm{ML}(\alpha) = \alpha; \quad \mathrm{ML}(\mathtt{int}) = \mathtt{int}; \quad \mathrm{ML}(\tau, \rho) = \mathrm{ML}(\tau)$$

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

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

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

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

Types, effects, and other objects constructed out of these are collectively referred to as *semantic objects*. Later on we introduce type schemes, which are semantic objects with binding occurrences of type, region, and effect variables. Throughout the paper, the set of type variables, effect variables, and region variables that occur free in a semantic object A are denoted $\mathit{ftv}(A)$, $\mathit{fev}(A)$, and $\mathit{frv}(A)$, respectively. For example, we define

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

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

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 $\mathit{Dom}(f)$ 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 $\{\}$.

5. UNIFICATION IN CONSISTENT BASES

We have already outlined the role of unification in our region inference algorithm (Section 4). We now look more closely at unification, with particular emphasis on effect variables.

To motivate our treatment of effect sets, Figure 4 shows the unification example from the previous section, using directed graphs (called *effect graphs*), to represent effects. An effect graph is finite. Each node is labeled by an effect variable or a region variable. No variable labels two distinct nodes. Region nodes are always terminal nodes; there can be arrows from effect variable nodes to other effect variables and to region variables.

Recall that an atomic effect is just a region or an effect variable (since we collapsed $\mathbf{put}(\rho)$ and $\mathbf{get}(\rho)$ to ρ). The idea is that there is an arrow from effect variable ϵ to atomic effect η, only if η is a member of the effect which ϵ denotes. The effect which an effect variable ϵ denotes is the set of nodes that can be reached from ϵ. Thus, in the top half of Figure 4, ϵ_2 denotes the set $\{\rho_1, \epsilon_1, \rho_3, \rho_4, \rho_6\}$. In the formal semantics, this is expressed by the "arrow effect"

$$\epsilon_2.\{\rho_1, \epsilon_1, \rho_3, \rho_4, \rho_6\}.$$

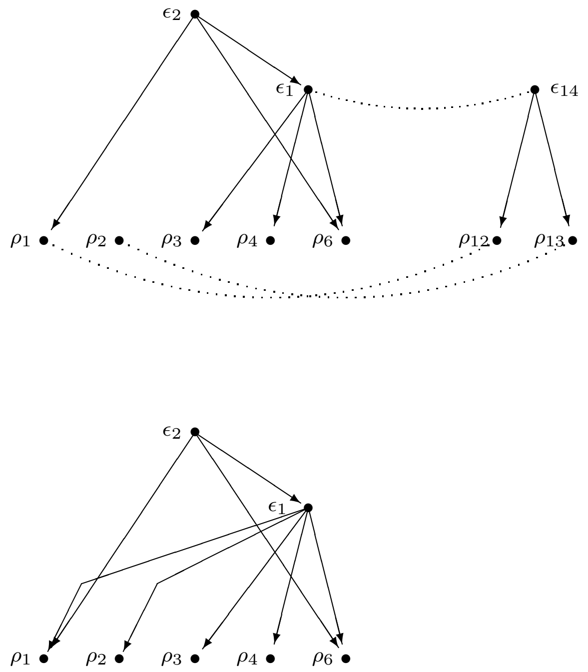

Fig. 4. An effect graph before unification (top) and after (bottom).

Still referring to the example in Section 4, the dotted lines connect nodes that are about to be unified.

Unification of two region variables ρ and ρ' is simple: just choose one of them, say ρ, and redirect all arrows that lead to ρ' so that they lead to ρ instead.

Unification of two effect variables is similar, except that the effect variable which is chosen to be the result of the unification must take over the arrows that emanated from the effect variable which is eliminated. In Figure 4, we chose to eliminate ϵ_{14}, so we had to add arrows from ϵ_1 to ρ_1 and ρ_2.

The unification procedure outlined above can be implemented using the UNION-FIND data structure (every dotted line corresponds to a UNION operation) and this is how it is done in the ML Kit. However, for proving properties of region inference, an algebraic formulation is convenient. That is why we operate with arrow effects rather than graphs. In order to ensure that a set of arrow effects really can be interpreted as an effect graph, we have to impose constraints on sets of arrow effects. In the terminology introduced below, they have to be *consistent*.

5.1 Consistency

Let Φ be a set of arrow effects. We say that Φ is *effect consistent*, written $\vdash \Phi$, if

(1) Φ is *functional*: $\forall(\epsilon_1.\varphi_1) \in \Phi.\forall(\epsilon_2.\varphi_2) \in \Phi.(\epsilon_1 = \epsilon_2 \Rightarrow \varphi_1 = \varphi_2)$;

(2) Φ is *closed*: $\forall(\epsilon.\varphi) \in \Phi.\forall\epsilon' \in \varphi.\exists\varphi'.((\epsilon'.\varphi') \in \Phi)$;

(3) Φ is *transitive*: $\forall(\epsilon_1.\varphi_1) \in \Phi.\forall(\epsilon_2.\varphi_2) \in \Phi.(\epsilon_2 \in \varphi_1 \Rightarrow \varphi_2 \subseteq \varphi_1)$.

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

The algebraic notion which corresponds to effect graphs is that of a *basis*. Formally, a *basis* is a pair $B = (Q, \Phi)$, where Q is a finite set of region variables and Φ is a finite set of arrow effects.

We say that $B = (Q, \Phi)$ is *consistent*, written $\vdash B$, if Φ is effect consistent and $Q \supseteq frv(\Phi)$. A consistent basis can be represented by an effect graph which has precisely one region variable node for each element of Q and precisely one effect variable node for every effect variable in the domain of $\hat{\Phi}$. The set of bases is called Basis. We use A and B to range over bases.

We emphasize that consistency is introduced for algorithmic reasons only; the soundness of the region inference rules has been proved without assuming consistency [Tofte and Talpin 1997]. The main advantages of consistency are (1) consistent bases can be represented efficiently as effect graphs; (2) in consistent bases, the unification algorithm outlined above yields most general unifiers (in a sense which we shall make precise).

When B is a basis we write $Q\,of\,B$ and $\Phi\,of\,B$ for the first and second component of B, respectively. We define $frv(B) = Q\;of\;B \cup frv(\Phi\;of\;B)$ and $fev(B) = fev(\Phi\;of\;B)$. The empty basis $(\emptyset, \emptyset)$ is denoted $B_\emptyset$. The *domain* of $B = (Q, \Phi)$, written $Dom(B)$, is the set $Q \cup Dom(\hat{\Phi})$.

We now describe a binary operation ($\uplus$) on bases which is important for understanding how binding of region and effect variables works. Recall that we said that the `letregion` rule (3) was "somewhat simplified". The point is that `letregion` is able to discharge both region variables and effect variables. Program (5) is an example. We found the effect of the translated version of $P(\lambda x.x + 1)$ at (6). But the region-annotated type of the translated version of the application $P(\lambda x.x + 1)$ is just $(\texttt{int}, \rho_2)$. Therefore, the `letregion` rule should be able to discharge ϵ_1 and ϵ_2 as well as all the region variables of (6), except ρ_2. In terms of effect graphs, `letregion` partitions the effect graph in two and discharges the "upper half"; see Figure 5. Thus `letregion` really binds a basis:

$$\begin{array}{l}
\texttt{val}\ n\ \texttt{=}\ \texttt{letregion}\ (\{\rho_0, \rho_1, \rho_3, \rho_4, \rho_6\},\\
\qquad\qquad\qquad\qquad \{\epsilon_1.\{\rho_1, \rho_2, \rho_3, \rho_4, \rho_6\}, \epsilon_2.\{\epsilon_1, \rho_1, \rho_2, \rho_3, \rho_4, \rho_6\}\})\\
\qquad\quad \texttt{in}\ (\lambda h.\quad \ldots\\
\qquad\qquad)\ \texttt{at}\ \rho_0 \quad (\lambda x.(x+1)\ \texttt{at}\ \rho_2)\ \texttt{at}\ \rho_3\\
\qquad\quad \texttt{end}
\end{array}$$

In this example, all effect variables were discharged; however, it is quite common that the separation of the basis discharges some, but not all, effect variables.

To make the above precise, let $B_1 = (Q_1, \Phi_1)$ and $B_2 = (Q_2, \Phi_2)$ be bases (they need not be consistent). 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, $Q_1 \cap Q_2 = \emptyset$, and $Dom(\hat{\Phi_1}) \cap Dom(\hat{\Phi_2}) = \emptyset$. (Otherwise, $B_1 \uplus B_2$ is undefined.) Often, B_1 will be consistent and B_2 will be functional and transitive, but not closed. Then the requirements for the existence of $B_1 \uplus B_2$ express that $B_1 \uplus B_2$ is a consistent extension of B_1 with region and

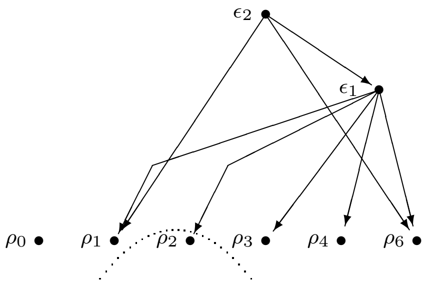

Fig. 5. Use of the `letregion`-rule for the program (5). The basis is split in two (separated by the dotted line) and the upper half bound by `letregion`.

effect variables that are not in B_1. This definition can be exploited by making bases explicit in the region inference rules and giving the following rule for `letregion`:

$$\frac{B \vdash TE \quad B \vdash \mu \quad B \uplus B_1 \text{ exists} \quad (B \uplus B_1), TE \vdash e_1 : \mu, \varphi_1}{B, TE \vdash \texttt{letregion}\ B_1\ \texttt{in}\ e_1\ \texttt{end} : \mu, \mathit{Observe}(B, \varphi_1)} \tag{7}$$

where the *observable part of* φ_1 *with respect to* B, written $\mathit{Observe}(B, \varphi_1)$, is the effect defined by

$$\mathit{Observe}(B, \varphi_1) = \varphi_1 \cap (\mathit{frv}(B) \cup \mathit{fev}(B)). \tag{8}$$

The idea is that the premises $B \vdash TE$ and $B \vdash \mu$ (defined formally later) ensure that B is consistent and that the type environment TE and the type with place μ are annotated using region variables and arrow effects from B only. It turns out that $B \vdash TE$ implies $(\mathit{frv}(TE) \cup \mathit{fev}(TE)) \subseteq \mathit{Dom}(B)$ and $B \vdash \mu$ implies $(\mathit{frv}(\mu) \cup \mathit{fev}(\mu)) \subseteq \mathit{Dom}(B)$. In particular, the premise "$B \uplus B_1$ exists" ensures that $\mathit{Dom}(B_1) \cap (\mathit{frv}(TE, \mu) \cup \mathit{fev}(TE, \mu)) = \emptyset$, and is thus a natural generalization of the side-condition of (3) to deal with effects. In general, B and $B \uplus B_1$ will be consistent, but B_1 will not necessarily be consistent. For example, in Figure 5 the upper basis refers to ρ_2 in the lower basis.

We now define the relation $B \vdash \mu$. Let $B = (Q, \Phi)$ be a basis. We say that a semantic object o is *consistent in* B, if the sentence $B \vdash o$ can be inferred from the following rules:

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

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

Note that in the rule for function types, φ is uniquely determined by B and ϵ: $\varphi = \hat{\Phi}(\epsilon)$, where $\Phi = \Phi \mathrel{\mathit{of}} B$.

For effects, we define consistency as follows. An effect φ is *consistent* in $B = (Q, \Phi)$, written $B \vdash \varphi$, if B is consistent, $\mathit{frv}(\varphi) \subseteq Q$, $\mathit{fev}(\varphi) \subseteq \mathit{Dom}(\hat{\Phi})$, and, for all $\epsilon'.\varphi' \in \Phi$, $\epsilon' \in \varphi$ implies $\varphi' \subseteq \varphi$.

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, that

$frv(o) \subseteq (Q \text{ of } B)$, and that $fev(o) \subseteq Dom(\hat{\Phi})$, where $\Phi = \Phi \text{ of } B$.

Inclusion $B \subseteq B'$ is defined to be componentwise inclusion and difference $B \setminus B'$ is componentwise set difference. For brevity, empty Q or Φ components are sometimes omitted; for example, $B \uplus \{\rho\}$ abbreviates $B \uplus (\{\rho\}, \emptyset)$.

5.2 Substitution

In terms of effect graphs, we need to be able to (1) collapse two region variable nodes; (2) collapse two effect variable nodes; and (3) add an edge from an effect variable node to an effect variable node or a region variable node. We now define the equivalent operations as substitutions on consistent bases.

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 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

Types and Region Variables

$$S(\texttt{int}) = \texttt{int} \quad S(\alpha) = S^t(\alpha) \quad S(\rho) = S^r(\rho) \quad S(\tau, \rho) = (S(\tau), S(\rho))$$

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

Arrow Effects

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

Effects

$$\begin{aligned} S(\varphi) \;=\; & \{S^r(\rho) \mid \rho \in \varphi\} \cup \\ & \{\eta \mid \exists \epsilon, \epsilon', \varphi'.\ \epsilon \in \varphi \wedge \epsilon'.\varphi' = S^e(\epsilon) \wedge \eta \in \{\epsilon'\} \cup \varphi'\}. \end{aligned}$$

Substitutions compose; Id is the identity substitution $(\mathrm{Id}^t, \mathrm{Id}^r, \mathrm{Id}^e)$, 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 $\{\alpha \in \mathrm{TyVar} \mid S^t(\alpha) \neq \alpha\}$. The *support* of a region substitution S^r, written $Supp(S^r)$, is the set $\{\rho \in \mathrm{RegVar} \mid S^r(\rho) \neq \rho\}$. 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.

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

5.3 Contraction

A key idea in the proof of termination of the region inference algorithm is that all three operations mentioned at the beginning of Section 5.2 in a sense make the basis smaller; in the terminology of this section, they are *contractions*. We show that any sequence of contractions starting from a consistent basis converges.

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 $\{\epsilon_1, \epsilon_2\} \subseteq Dom(\hat{\Phi})$ and $\hat{\Phi}(\epsilon_1) = \hat{\Phi}(\epsilon_2)$; or
(4) $S = (\{\}, \{\}, \{\epsilon \mapsto \epsilon.\varphi\})$, for some ϵ and φ with $\epsilon \in Dom(\hat{\Phi})$, $B \vdash \varphi$, and $\varphi \supseteq \hat{\Phi}(\epsilon)$.

Notice that—despite its name—an elementary contraction of the fourth kind can actually increase the effect associated with some effect variable, but only with variables which already occur in the basis. 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 $B_1 = S(B_2)$. 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 $B_1 \neq B_2$. Note that a contraction S can satisfy $S(B) = B$ and yet not be the identity substitution (e.g., $S = \{\epsilon \mapsto \epsilon.\varphi \mid \epsilon.\varphi \in \Phi \text{ } of \text{ } B\}$). Our algorithm will not produce such "idle" substitutions: whenever we produce an S which is not the identity, it will make the basis strictly smaller.

Lemma 5.3.1. *Let B_0 be a consistent basis. There exists no infinite descending chain:*

$$B_0 > B_1 > B_2 > \cdots \quad .$$

Proof. For the duration of this proof, we write $|X|$ to mean the number of elements of X, whenever X is a finite set. Let $A = (Q, \Phi)$ be a basis. Let $A' = (Q', \Phi')$ be a basis with $A' < A$. Let S be a contraction of A with $S(A) = A'$. Without loss of generality we can assume that S is elementary. It follows from the definition of elementary contraction that

(1) $|Q| > |Q'| \wedge |\,Dom(\hat{\Phi})| \geq |\,Dom(\hat{\Phi}')|$; or
(2) $|Q| \geq |Q'| \wedge |\,Dom(\hat{\Phi})| > |\,Dom(\hat{\Phi}')|$; or
(3) $|Q| = |Q'| \wedge |\,Dom(\hat{\Phi})| = |\,Dom(\hat{\Phi}')| \wedge (\forall \epsilon \in Dom(\hat{\Phi}).\hat{\Phi}(\epsilon) \subseteq \hat{\Phi}'(\epsilon)) \wedge (\exists \epsilon \in Dom(\hat{\Phi}).\hat{\Phi}(\epsilon) \subset \hat{\Phi}'(\epsilon))$.

In other words, since $S(A) \neq A$, S either decreases the number of variables occurring in A or, failing that, maintains the set of variables and increases the size of at least one effect within this finite number of variables (without decreasing the size of any effect). Starting from a (finite) basis A_0, any chain $A_0 \geq A_1 \geq A_2 \geq \cdots$ therefore is constant from some point onwards. □

Lemma 5.3.2. *Let o be a semantic object and let S be a contraction of a basis B. If B is consistent, then $S(B)$ is consistent. If $B \vdash o$, then $S(B) \vdash S(o)$.*

PROOF. By induction on the depth of $B \vdash o$. □

5.4 Unification

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 $\epsilon.\varphi \in \Phi$.

For example, consider the basis $B = (\{\rho_1, \rho_2\}, \{\epsilon_1.\{\rho_1\}, \epsilon_2.\{\rho_2\}\})$. Here are two most general unifiers for the two arrow effects $\epsilon_1.\{\rho_1\}$ and $\epsilon_2.\{\rho_2\}$ in B:

$$\begin{aligned} S_1 &= \{\epsilon_1 \mapsto \epsilon_1.\{\rho_1, \rho_2\}, \epsilon_2 \mapsto \epsilon_1.\{\rho_1, \rho_2\}\} \\ S_2 &= \{\epsilon_1 \mapsto \epsilon_1.\{\rho_2\}, \epsilon_2 \mapsto \epsilon_1.\{\rho_1\}\} \end{aligned}$$

Although S_1 and S_2 are quite different in general, one has $S_1(\mu) = S_2(\mu)$, for all μ satisfying $B \vdash \mu$. The substitution

$$S = (\{\rho_2 \mapsto \rho_1\}, \{\epsilon_1 \mapsto \epsilon_2.\emptyset\})$$

is a unifier for $\epsilon_1.\{\rho_1\}$ and $\epsilon_2.\{\rho_2\}$ in B, but it is not most general, for there exists no substitution S_1' satisfying both

$$\rho_1 = S_1(\rho_1) = S_1'(S(\rho_1)) = S_1'(\rho_1)$$

and

$$\rho_2 = S_1(\rho_2) = S_1'(S(\rho_2)) = S_1'(\rho_1).$$

The most general way to unify arrow effects involves taking the union of the sets the handles denote, without performing substitution on the region variables in those sets [Tofte and Birkedal 1996].

LEMMA 5.4.1. *There exists an algorithm unifyArrEff with the following properties. 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* $S^e = \mathrm{Id}^e$.

The algorithm with a proof of the above lemma may be found in Tofte and Birkedal [1998]. The basic idea is the one outlined in the example above: unification of arrow effects is done by unioning the sets the effect variables denote and then eliminating one of the effect variables.

Next, we consider unification of region-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 (ML type inference is over by the time we do region inference). 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 $\epsilon.\varphi \in \Phi$.

LEMMA 5.4.2. *There exist algorithms unifyRho, unifyTy, and unifyMu with the following properties. Let B be a consistent basis and assume $B \vdash \tau_1$, $B \vdash \tau_2$, and* $\mathrm{ML}(\tau_1) = \mathrm{ML}(\tau_2)$. *Then $S^* = unifyTy(\tau_1, \tau_2)$ 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 unifyRho(ρ_1, ρ_2) and unifyMu(μ_1, μ_2).

A proof of the above lemma may be found in Tofte and Birkedal [1998]; it uses *unifyArrEff* as a subroutine. An example of unification was given in Section 4.

6. TYPE SCHEMES

Type schemes resemble the type schemes of Damas and Milner [1982] 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 *body*(σ). The *bound variables of* σ, written $bv(\sigma)$, 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 $btv(\sigma)$, to be the set $\{\alpha_1, \ldots, \alpha_n\}$. The *free variables of* σ, written $fv(\sigma)$, is the set $fv(\tau) \setminus bv(\sigma)$. The bound variables of a type scheme must be distinct. The *arity* of σ is the triple (n, k, m). The set of type schemes is denoted TypeScheme.

The erase function of Section 4 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 and permutation 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 σ' is an *instance of* σ *(via S)*, written $\sigma \geq \sigma'$ (via S), if $\sigma \geq body(\sigma')$ and $bv(\sigma') \cap fv(\sigma) = \emptyset$. As in Milner's type system one has:

LEMMA 6.1. *Let σ and σ' be type schemes. Then $\sigma \geq \sigma'$ iff for every τ, $\sigma' \geq \tau$ implies $\sigma \geq \tau$.*

Proof. See Tofte and Birkedal [1998]. □

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 bound variables. For example, letting $\mu_0 = (\mathtt{int}, \rho_0)$ we have

$$\forall \epsilon\rho.\mu_0 \xrightarrow{\epsilon.\{\rho\}} \mu_0 \equiv \forall \epsilon\rho\rho'.\mu_0 \xrightarrow{\epsilon.\{\rho,\rho'\}} \mu_0. \tag{9}$$

(To see this, note that $\forall \epsilon\rho.\mu_0 \xrightarrow{\epsilon.\{\rho\}} \mu_0 \geq \forall \epsilon\rho\rho'.\mu_0 \xrightarrow{\epsilon.\{\rho,\rho'\}} \mu_0$ via $\{\epsilon \mapsto \epsilon.\{\rho'\}\}$ and that $\forall \epsilon\rho\rho'.\mu_0 \xrightarrow{\epsilon.\{\rho,\rho'\}} \mu_0 \geq \forall \epsilon\rho.\mu_0 \xrightarrow{\epsilon.\{\rho\}} \mu_0$ via $\{\rho' \mapsto \rho\}$.) 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.)

For an example of a type scheme, consider function g in Figure 3. It has type scheme

$$\forall \rho_5\epsilon_3.(\mathtt{int}, \rho_5) \xrightarrow{\epsilon_3.\{\rho_5,\rho_4,\rho_3\}} ((\mathtt{int}, \rho_1) \xrightarrow{\epsilon_1.\varphi_1} (\mathtt{int}, \rho_2), \rho_3) \tag{10}$$

where φ_1 is as in Section 4.

6.1 Well-Formed Type Schemes

There are certain type schemes we want to reject as ill-formed, for reasons explained in the following.

Definition 6.1.1. 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 ϵ of an arrow effect $\epsilon.\varphi$ in τ is free, then the entire arrow effect is free.

We later impose a restriction which bans ill-formed type schemes. If one allowed type schemes to be ill-formed, consistency of the basis could be compromised by instantiation of type schemes. For example, consider the type scheme $\sigma = \forall \rho.(\mathtt{int}, \rho_0) \xrightarrow{\epsilon.\{\rho\}} (\mathtt{int}, \rho_0)$ 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 $(\mathtt{int}, \rho_0) \xrightarrow{\epsilon.\{\rho_1\}} (\mathtt{int}, \rho_0)$ and $(\mathtt{int}, \rho_0) \xrightarrow{\epsilon.\{\rho_2\}} (\mathtt{int}, \rho_0)$ are not consistent in any basis. This clearly does not agree well with basing region inference on unification of consistent types. This example also illustrates that if one wants region polymorphism and consistency, then one must also allow effect polymorphism.

Conversely, by restricting attention to well-formed type schemes, we have the advantage that the same device which underlies binding at `letregion`, namely, disjoint union of bases (Section 5.1), can account for binding in type schemes as well. Figure 6 shows a basis before and after the formation of the type scheme (10) for function g from Figure 3.

6.2 Polymorphic Recursion

The Tofte-Talpin region inference rules allow polymorphic region recursion [Tofte and Talpin 1997]. In other words, let $\sigma = \forall \alpha_1 \cdots \alpha_n \rho_1 \cdots \rho_k \epsilon_1 \cdots \epsilon_m.\tau$ be a type scheme

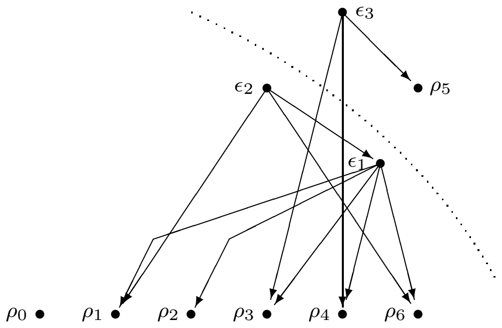

Fig. 6. Forming the type scheme for g in Figure 3 splits the basis in two (separated by the dotted line) and binds the upper half by $\forall$.

and consider a typed region-annotated expression of the form

$$\texttt{letrec}\ f : (\sigma, \rho)(x)\ \texttt{=}\ t_1\ \texttt{in}\ t_2\ \texttt{end.} \tag{11}$$

Then *both* t_1 and t_2 may contain instances of f of the form

$$f_{([\tau_1,\ldots,\tau_n],[\rho'_1,\ldots,\rho'_k],[\epsilon'_1.\varphi'_1,\ldots,\epsilon'_m.\varphi'_m])}$$

where we do not necessarily have $[\rho'_1, \ldots, \rho'_k] = [\rho_1, \ldots, \rho_k]$ and $[\epsilon'_1, \ldots, \epsilon'_m] = [\epsilon_1, \ldots, \epsilon_m]$. In t_1, however, it is required that $[\tau_1, \ldots, \tau_n] = [\alpha_1, \ldots, \alpha_n]$, i.e., we do not admit polymorphic recursion in types. (Polymorphic recursion in types would make the type inference problem undecidable.) For an example of polymorphic region recursion, note that g in Figure 3 calls itself with an actual region (ρ_7) which is different from the formal region (ρ_5).

The reason for allowing polymorphic recursion in regions and effects is that it can (and often does) give dramatic improvements of the results of the region inference. For example, the space usage of a program can drop from linear in running time to logarithmic in running time by admitting polymorphic region recursion; for an example, see Tofte and Talpin [1997, Figure 2]. Polymorphic recursion in types is known to make type inference undecidable; polymorphic recursion can lead to an infinite growth in the size of types. If one is not careful, similar phenomena appear when doing region inference. The problem is that even though region inference only annotates ML types (which are known when region inference begins), types contain effects, and effect sets can grow large. To obtain termination, some bound on the number of region and effect variables used is required.

To see why that is so, let us show an example of how *not* to do region inference. Milner's algorithm W [Milner 1978] generates fresh type variables each time it takes an instance of a type scheme. Mycroft [1984] generalizes this idea to polymorphic recursion. Consider an expression of the form

$$\texttt{letrec}\ f\texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end.}$$

Mycroft's algorithm iteratively analyzes e_1, as follows. First e_1 is analyzed under the assumption that f has type scheme $\sigma_0 = \forall\alpha.\alpha$, the most general of all type schemes. Let τ_1 be the type inferred for e_1 and let σ_1 be the result of quantifying all type variables in τ_1 that are not free in the type environment. If $\sigma_0 = \sigma_1$, the iteration stops. Otherwise, e_1 is analyzed again, this time assuming that f has type σ_1. This iteration continues until two consecutive type schemes become equal. However, types may grow unboundedly and the process may loop forever.

Since the ML types are given at the outset of region inference, and since region inference simply annotates these types, it is natural to try to generalize Mycroft iteration to region inference in the hope that it would always terminate in what appears to be a simpler form of polymorphic recursion. Here the idea is to start the iteration from the most general type scheme whose erasure is the known ML type scheme. For example, we start the analysis of g from our example (Section 3.1) from the type scheme

$$\sigma_0 = \forall \rho_1\rho_2\rho_3\rho_4\epsilon_1\epsilon_2.(\texttt{int}, \rho_1) \xrightarrow{\epsilon_1.\emptyset} ((\texttt{int}, \rho_2) \xrightarrow{\epsilon_2.\emptyset} (\texttt{int}, \rho_3), \rho_4).$$

Then we analyze the body of the recursive function. Each time we need a region variable for an `at` annotation, we pick a fresh one. Each time we take an instance of a type scheme, we pick fresh region and effect variables. So in the first iteration we pick a fresh region variable, say ρ_7, for $(a-1)$. At the end of the first iteration, it will be clear that the range type of g has to be the type of h and that the latent effect of h must contain ρ_7. The next type scheme for g will take the form

$$\sigma_1 = \forall \rho_1\epsilon_1.(\texttt{int}, \rho_1) \xrightarrow{\epsilon_1.\{\rho_7\}} ((\texttt{int}, \rho_2) \xrightarrow{\epsilon_2.\{\rho_6,\rho_7\}} (\texttt{int}, \rho_3), \rho_4).$$

The second iteration will now pick a fresh region variable, say ρ_8, for $(a-1)$ and will give a type scheme of the form

$$\sigma_2 = \forall \rho_1\epsilon_1.(\texttt{int}, \rho_1) \xrightarrow{\epsilon_1.\{\rho_7,\rho_8,\ldots\}} ((\texttt{int}, \rho_2) \xrightarrow{\epsilon_2.\{\rho_6,\rho_7,\rho_8,\ldots\}} (\texttt{int}, \rho_3), \rho_4).$$

Upon each iteration, one more region variable will be generated and added to the latent effect of h, so the process will never terminate!

To understand the problem more deeply, let us first define:

Definition 6.2.1. Let σ_0 and σ_1 be type schemes. We say that σ_1 *matches* σ_0 *(via S)*, written $\sigma_0 \sqsupseteq \sigma_1$ (via S), if $\mathrm{ML}(\sigma_0) = \mathrm{ML}(\sigma_1)$ and $\sigma_0 \geq \sigma_1$ (via S).

Since $\geq$ is a partial order on equivalence classes of type schemes, so is $\sqsupseteq$. What we observed in the above example was an instance of the following lemma.

LEMMA 6.2.2. *Matching is a non-well-founded relation.*

PROOF. Let $\mu_0 = (\texttt{int}, \rho_0)$. We have

$$\forall\epsilon.\mu_0 \xrightarrow{\epsilon.\emptyset} \mu_0 \sqsupset \forall\epsilon.\mu_0 \xrightarrow{\epsilon.\{\rho_1\}} \mu_0 \sqsupset \forall\epsilon.\mu_0 \xrightarrow{\epsilon.\{\rho_1,\rho_2\}} \mu_0 \sqsupset \cdots .$$

Since the set of region variables is infinite, this chain can be continued ad infinitum. □

So we need to find ways of limiting the number of region and effect variables that are generated. In the above example, it was clearly unnecessary to pick fresh region variables for $(a-1)$ upon each iteration. In the program that is output from region inference, there will be precisely one region variable for every value-producing expression, so it must be sufficient to generate only one region variable for that subterm. Similarly, for every occurrence of a lambda abstraction, one effect variable is needed. The only other places where region and effect variables have to be generated are when taking instances of type schemes. This raises the question: how many bound variables do there have to be in a type scheme?

To motivate the answer to this question, let us consider how many bound region and effect variables may be needed for region type schemes whose erasure is the ML type `int` $\to$ (`int` $\to$ `int`). We could need two effect variables (one for each function arrow) and four region variables (one for each occurrence of `int` or $\to$, except the outermost constructor):

$$\sigma_0 = \forall\rho_1\rho_2\rho_3\rho_4\epsilon_1\epsilon_2.(\mathtt{int},\rho_1) \xrightarrow{\epsilon_1.\emptyset} ((\mathtt{int},\rho_2) \xrightarrow{\epsilon_2.\emptyset} (\mathtt{int},\rho_3),\rho_4).$$

But it is also possible to bind region and effect variables that occur in latent effects without occurring paired with a type constructor! For example, consider the type schemes

$$\sigma_1 = \forall\rho_1\rho_2\rho_3\rho_4\rho_5\epsilon_1\epsilon_2.(\mathtt{int},\rho_1) \xrightarrow{\epsilon_1.\emptyset} ((\mathtt{int},\rho_2) \xrightarrow{\epsilon_2.\{\rho_5\}} (\mathtt{int},\rho_3),\rho_4)$$

$$\sigma_2 = \forall\rho_1\rho_2\rho_3\rho_4\rho_5\epsilon_1\epsilon_2.(\mathtt{int},\rho_1) \xrightarrow{\epsilon_1.\{\rho_5\}} ((\mathtt{int},\rho_2) \xrightarrow{\epsilon_2.\emptyset} (\mathtt{int},\rho_3),\rho_4)$$

$$\sigma_3 = \forall\rho_1\rho_2\rho_3\rho_4\rho_5\rho_6\epsilon_1\epsilon_2.(\mathtt{int},\rho_1) \xrightarrow{\epsilon_1.\{\rho_5\}} ((\mathtt{int},\rho_2) \xrightarrow{\epsilon_2.\{\rho_6\}} (\mathtt{int},\rho_3),\rho_4).$$

None of these are equal; indeed $\sigma_0 > \sigma_1 > \sigma_3$ and $\sigma_0 > \sigma_2 > \sigma_3$, while σ_1 and σ_2 are unrelated with respect to instantiation. Let us refer to the region variables that occur paired with a type constructor as *primary* in the type and let us refer to the rest, i.e., those that occur in a latent effect only, as *secondary.* In σ_2, for example, ρ_2 is primary and ρ_5 is secondary. Similarly, let us say that an effect variable is *primary* in τ if it is the handle of an arrow effect in τ and let us call it *secondary* if it occurs in latent effects only. Concerning region variables, we see that we might need 2^n distinct bound secondary region variables, where n is the number of function arrows in τ, namely, one secondary region variable for each subset of the n arrows. Interestingly, however, we do not need *more* than 2^n secondary region variables. For example, consider

$$\sigma_4 = \forall\rho_1\rho_2\rho_3\rho_4\rho_5\epsilon_1\epsilon_2.(\mathtt{int},\rho_1) \xrightarrow{\epsilon_1.\{\rho_5\}} ((\mathtt{int},\rho_2) \xrightarrow{\epsilon_2.\{\rho_5\}} (\mathtt{int},\rho_3),\rho_4)$$

and

$$\sigma_4' = \forall\rho_1\rho_2\rho_3\rho_4\rho_5\rho_5'\epsilon_1\epsilon_2.(\mathtt{int},\rho_1) \xrightarrow{\epsilon_1.\{\rho_5,\rho_5'\}} ((\mathtt{int},\rho_2) \xrightarrow{\epsilon_2.\{\rho_5,\rho_5'\}} (\mathtt{int},\rho_3),\rho_4).$$

Here we have $\sigma_4 \geq \sigma_4'$ via $\{\epsilon_1 \mapsto \epsilon_1.\{\rho_5'\}, \epsilon_2 \mapsto \epsilon_2.\{\rho_5'\}\}$ and $\sigma_4' \geq \sigma_4$ via $\{\rho_5' \mapsto \rho_5\}$. The idea is that whenever two bound secondary region variables ρ and ρ' occur on the same subset of arrows in a type scheme σ', they can be collapsed, yielding a type scheme σ satisfying $\sigma \equiv \sigma'$. This construction works for type schemes that are well-formed and have consistent bodies. One has $\sigma' \geq \sigma$ via the substitution $\{\rho' \mapsto \rho\}$. The other direction, $\sigma \geq \sigma'$, relies on the well-formedness of σ' to ensure that the effect variables $\epsilon_1, \ldots, \epsilon_m$ that are handles of arrow effects in which ρ occurs are bound too; the consistency of the body is needed to ensure that the substitution $\{\epsilon_i \mapsto \epsilon_i.\{\rho'\}\,; 1 \leq i \leq m\}$ is the identity on all arrow effects in τ that do not contain ρ.

So no consistent region type scheme needs more than 2^n bound secondary region variables, where n is the number of function arrows in the underlying ML type scheme. Similarly, one needs at most 2^n bound secondary effect variables. Thus 2^{n+1} secondary bound region and effect variables suffice.

While there is thus a bound on the number of variables required for expressing the *output* of region inference, it is not clear that this bound is sufficient *during* region inference. The difficulty is that the set of arrows on which a given region or effect variable occurs may vary during region inference. That two region variables occur on the same set of function arrows during one point of region inference does not imply that they would have to occur on the same set of arrows in a final solution. For example, while the above σ'_4 can be simplified to σ_4 in a final solution, that simplification might sacrifice principality, if done during region inference. (It could be that ρ_5 and ρ'_5 just happened to be on the same set of function arrows at the time the simplification was made.) A similar problem exists concerning instantiation of secondary variables which do not stay on the same set of function arrows throughout the inference process. At the time of writing, we do not see any way of avoiding making arbitrary identifications of secondary region variables while maintaining a bound on the number of variables in use.

Because of the bound on the number of variables, if an expression has a principal type scheme, then it can be found by enumerating all possible type schemes and region annotations and selecting the most general type scheme which gives a legal derivation. But it is not known whether enumeration can lead to type schemes that are in-equivalent, but maximal (with respect to $\sqsupseteq$) amongst the type schemes that give legal derivations. The existence of (unique) principal type schemes remains an open problem.

Our algorithm circumvents these problems by omitting generalization of secondary variables altogether. The price we pay is that we lose principality (example included). The gains are (a) during the iterative analysis of recursive functions, we shall never have to generate fresh region and effect variables; (b) our algorithm becomes polynomial, namely, $O(n^4)$, where n is the size of the ML term including type annotations.

We now give an example of a function which has a secondary region variable in its type and for which our algorithm fails to find the most general type scheme.

Example 6.2.3. The following expression has a type which contains a secondary region variable:

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

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 would like to give f the type scheme $\sigma=\forall\rho_0\rho_1\rho_2\rho_3\rho_4\epsilon_1\epsilon_2.\tau_0$. However, since we do not generalize secondary variables, we construct the less general type scheme $\sigma = \forall\rho_0\rho_2\rho_3\rho_4\epsilon_1\epsilon_2.\tau_0$.

We now define the notions of primary and secondary variables formally. Let A be a type or a type and a place. The set of *primary free effect variables of* A, written $pfev(A)$, and the set of *primary free region variables of* A, written $pfrv(A)$, are defined by

$$\begin{aligned} pfev(\texttt{int}) &= \emptyset \\ pfev(\alpha) &= \emptyset \\ pfev(\mu \xrightarrow{\epsilon.\varphi} \mu') &= pfev(\mu) \cup pfev(\mu') \cup \{\epsilon\} \\ pfev(\tau,\rho) &= pfev(\tau) \end{aligned} \qquad \begin{aligned} pfrv(\texttt{int}) &= \emptyset \\ pfrv(\alpha) &= \emptyset \\ pfrv(\mu \xrightarrow{\epsilon.\varphi} \mu') &= pfrv(\mu) \cup pfrv(\mu') \\ pfrv(\tau,\rho) &= pfrv(\tau) \cup \{\rho\}. \end{aligned}$$

Here the equation $pfev(\mu \xrightarrow{\epsilon.\varphi} \mu') = pfev(\mu) \cup pfev(\mu') \cup \{\epsilon\}$ expresses that effect variables which occur in latent effects only are not primary; similarly, the equation $pfrv(\mu \xrightarrow{\epsilon.\varphi} \mu') = pfrv(\mu) \cup pfrv(\mu')$ expresses that region variables that occur in latent effects only are not primary.

Next, we say that an effect variable ϵ is a *secondary effect variable* in τ if $\epsilon \in fev(\tau) \setminus pfev(\tau)$; similarly, we say that a region variable ρ is *secondary* in τ if $\rho \in frv(\tau) \setminus pfrv(\tau)$.

Definition 6.2.4. 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 $\{\rho_1,\ldots,\rho_k\} \subseteq pfrv(\tau)$, $\{\epsilon_1,\ldots,\epsilon_m\} \subseteq pfev(\tau)$, and $\{\alpha_1,\ldots,\alpha_n\} \subseteq ftv(\tau)$.

6.3 Overview of the Algorithm

Our region inference algorithm is actually two algorithms, $\mathcal{S}$ and $\mathcal{R}$. Algorithm $\mathcal{S}$ takes as input an explicitly typed source term (Section 3). $\mathcal{S}$ is in charge of all generation of fresh region and effect variables; the entire term (including definitions of recursive functions) is traversed once. Algorithm $\mathcal{S}$ does not attempt to find the right type schemes of polymorphic functions. Algorithm $\mathcal{R}$, on the other hand, generates no new variables, but uses Mycroft iteration to resolve the type schemes of recursive functions. We refer to the job done by $\mathcal{S}$ as "spreading" (since it sprinkles fresh variables over the term) and we refer to the job done by $\mathcal{R}$ as "fixed-point resolution."

Algorithm $\mathcal{S}$ builds a basis, B_0, containing the variables it generated.

> *The crux of the termination argument is that $\mathcal{R}$ only contracts B_0 and we know (from Lemma 5.3.1) that iterated contraction of a basis must terminate.*

The terms produced by $\mathcal{S}$ and iteratively transformed by $\mathcal{R}$ contain a region variable for each `at` annotation. Furthermore, every nonbinding variable occurrence is decorated by an instantiation list which describes the instantiation of the type scheme of that variable at that occurrence.

Formally, 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. 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.

The following lemma guarantees that the instantiation lists of the $(i+1)$st iteration of a function body can be obtained by "thinning" the instantiation lists of the ith iteration, provided the type schemes have structural quantification. This is where the concept of structural quantification plays a crucial role.

LEMMA 6.3.1. *Assume that* σ *and* σ' *both have structural quantification and that* $\sigma \sqsupseteq \sigma'$ *via* S. *Write* σ *in the form* $\forall\alpha_1\cdots\alpha_n\rho_1\cdots\rho_k\epsilon_1\cdots\epsilon_m.\tau$ *and write* σ' *in the form* $\forall\alpha_1\cdots\alpha_n\rho'_1\cdots\rho'_{k'}\epsilon'_1\cdots\epsilon'_{m'}.\tau'$. *Then* $k' \leq k$, $m' \leq m$, *and*

(*1*) *for every* $i' \in \{1,\ldots,k'\}$ *there exists an* $i \in \{1,\ldots,k\}$ *such that* $S(\rho_i) = \rho'_{i'}$;

(*2*) *for every* $j' \in \{1,\ldots,m'\}$ *there exists a* $j \in \{1,\ldots,m\}$ *and a* φ *such that* $S(\epsilon_j) = \epsilon'_{j'}.\varphi$.

PROOF. Since σ' has structural quantification, every bound region or effect variable in σ' occurs in a principal position in τ'. Since $\sigma \sqsupseteq \sigma'$, we have $\mathrm{ML}(\sigma) = \mathrm{ML}(\sigma')$. Therefore, the principal position in question exists in σ as well. By the definition of instantiation, the variable at that position in σ must be bound. Thus (1) and (2) follow. Finally $k' \leq k$ and $m' \leq m$ follow from the fact that S is a map. □

Now the idea is that actual region argument number i' in the thinned instance list is going to be argument number i of the unthinned list (the conclusion $k' \leq k$ ensures that this makes sense), similarly for arrow effects. To be precise, let il and il' be instantiation lists.

$$il = ([\tau''_1,\ldots,\tau''_n],[\rho''_1,\ldots,\rho''_k],[\epsilon''_1.\varphi''_1,\ldots,\epsilon''_m.\varphi''_m]) \tag{12}$$

$$il' = ([\tau'''_1,\ldots,\tau'''_{n'}],[\rho'''_1,\ldots,\rho'''_{k'}],[\epsilon'''_1.\varphi'''_1,\ldots,\epsilon'''_{m'}.\varphi'''_{m'}]), \tag{13}$$

and assume $\sigma = \forall\alpha_1\cdots\alpha_n\rho_1\cdots\rho_k\epsilon_1\cdots\epsilon_m.\tau$ and $\sigma' = \forall\alpha_1\cdots\alpha_n\rho'_1\cdots\rho'_{k'}\epsilon'_1\cdots\epsilon'_{m'}.\tau'$. We say that *$il'$ is a thinning of il (since σ' matches σ)*, written $il\, R^{\sigma \sqsupseteq \sigma'}\, il'$, if $arity(il) = arity(\sigma)$, $arity(il') = arity(\sigma')$, and there exists a substitution S such that $\sigma \geq \sigma'$ *via* S and

(1) $n = n'$ and $\forall i \in \{1,\ldots,n\}.\tau''_i = \tau'''_i$,

(2) $\forall j' \in \{1,\ldots,k'\}.\exists j \in \{1,\ldots,k\}.S(\rho_j) = \rho'_j$ and $\rho'''_{j'} = \rho''_j$,

(3) $\forall l' \in \{1,\ldots,m'\}.\exists l \in \{1,\ldots,m\}.\exists\varphi'.S(\epsilon_l) = \epsilon'_{l'}.\varphi'$ and $\epsilon'''_{l'}.\varphi'''_{l'} = \epsilon''_l.\varphi''_l$.

Given that $\sigma \sqsupseteq \sigma'$ and both of these type schemes have structural quantification, the substitution S is easy to find, since all region and effect variables in its domain occur in primary positions in the body of σ. Consequently, when σ, σ', and il are given, it is easy to compute an il' such that $il\, R^{\sigma \sqsupseteq \sigma'}\, il'$. Note that there may be more than one such il', since there may be more than one j and l satisfying the conditions in (2) and (3), respectively. However, every choice gives the same result, due to a unification that takes place in line 4 of algorithm $\mathcal{R}$ (Section 9.2).

LEMMA 6.3.2. *If* $il_1\, R^{\sigma_1 \sqsupseteq \sigma_2}\, il_2$ *and* $il_2\, R^{\sigma_2 \sqsupseteq \sigma_3}\, il_3$*, then* $il_1\, R^{\sigma_1 \sqsupseteq \sigma_3}\, il_3$.

PROOF. Simple verification. □

6.4 Consistent Type Schemes

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 $bound(\sigma)$, is the basis $(\{\rho_1,\ldots,\rho_k\},\{\epsilon.\varphi \in arreff(\tau) \mid \epsilon \in bv(\sigma)\})$.

Definition 6.4.1. 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 $B' = bound(\sigma)$, the basis $B \uplus B'$ exists 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 $Dom(\hat{\Phi}) = \{\epsilon_1,\ldots,\epsilon_m\}$, where $\Phi = \Phi\ of\ bound(\sigma)$. The use of disjoint union in the formation of type schemes was illustrated in Figure 6.

LEMMA 6.4.2. *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 $B' = (Q',\Phi') = bound(\sigma)$. By $B \vdash \sigma$ we have that the basis $B \uplus B'$ exists, is consistent, and $B \uplus B' \vdash \tau$. Take $\epsilon.\varphi \in arreff(\tau)$ 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 $\epsilon \notin bv(\sigma)$ we have $\epsilon.\varphi \in B$. But then, since B itself is consistent we have $fv(\epsilon.\varphi) \subseteq fv(B)$ and thus $fv(\epsilon.\varphi) \cap Dom(B') = \emptyset$. □

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.

6.5 Instantiation of Type Schemes

In Tofte and Birkedal [1998] an algorithm *inst* for instantiating type schemes is defined. We do not repeat it here, but simply state the property of the algorithm which we need in the following.

THEOREM 6.5.1. *Assume* $B \vdash \sigma$, $B \vdash il$*, and that* σ *and* il *have the same arity. Then* $(S_1^e, \tau') = inst(\sigma, il)$ *succeeds and*

(1) S_1^e *is a contraction of* B;
(2) $S_1^e(\sigma) \geq \tau'$ *via* $S_1^e(il)$ *and* $S_1^e(B) \vdash \tau'$;
(3) $S_1^e(B) = B$ *implies* $S_1^e = \mathrm{Id}^e$.

PROOF. See Tofte and Birkedal [1998]. □

The following example illustrates why *inst* returns a substitution.

Example 6.5.2. Consider $B = (\{\rho_0\}, \{\epsilon_0.\emptyset\})$, $\sigma = \forall \epsilon\rho.(\texttt{int}, \rho_0) \xrightarrow{\epsilon.\{\rho\}} (\texttt{int}, \rho)$, and $S = (\{\rho \mapsto \rho_0\}, \{\epsilon \mapsto \epsilon_0.\emptyset\})$. The instance of σ via S is the type $\tau' = (\texttt{int}, \rho_0) \xrightarrow{\epsilon_0.\{\rho_0\}} (\texttt{int}, \rho_0)$, which is not consistent in B, since the arrow effect $\epsilon_0.\emptyset$ has "grown" to become $\epsilon_0.\{\rho_0\}$. Therefore, *inst* returns an effect substitution $S^e = \{\epsilon_0 \mapsto \epsilon_0.\{\rho_0\}\}$.

6.6 Generalization of Region and Effect Variables

In this section, we describe how to form type schemes from types. The general idea was already described (end of Section 6.1). The partitioning of the basis (see Figure 6) is achieved by using an operation called *Below*. It is similar to the notion of closure used by Nielson et al. [1996] and is defined as follows.

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 (Section 5.1). Also, we do not necessarily have $Q_0 \supseteq \mathit{frv}(\Phi_0)$. Thus B_0 is not necessarily consistent. However, let $\mathit{Below}(B, B_0) = (Q_1, \Phi_1)$, where Φ_1 is the smallest closed subset of Φ satisfying $\Phi_1 \supseteq \Phi_0$ and Q_1 is $Q_0 \cup \mathit{frv}(\Phi_1)$. Then $\mathit{Below}(B, B_0)$ is obviously a consistent basis; indeed it is the smallest consistent basis B' satisfying $B_0 \subseteq B' \subseteq B$.

In terms of effect graphs, $\mathit{Below}(B, B_0)$ is the subgraph of B which can be reached starting from nodes in B_0.

The operation which forms type schemes from types is called *RegEffGen*. Let τ be a type and let φ be an effect. Furthermore, let B_0 be a basis and let $B \supseteq B_0$ be a basis with $B \vdash \tau$. Intuitively, B_0 and φ represent those region and effect variables that must not be quantified, typically because they may occur free in the type environment. We now define the operation $(B_1, \sigma) = \mathit{RegEffGen}(B, B_0, \varphi)(\tau)$ by the following:

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

Here (Q_0', Φ_0') represent variables that must not be generalized (line 4). These include region and effect variables in φ (lines 2 and 3). More interestingly, Q_0' contains secondary region variables $(\mathit{frv}(\tau) \setminus \mathit{pfrv}(\tau))$ and secondary effect variables $(\mathit{fev}(\tau) \setminus \mathit{pfev}(\tau))$ in τ. Moreover, if an effect variable ϵ occurring in τ cannot be generalized, for example, because it is secondary, we ensure that no region or effect variable which occurs in the effect which ϵ denotes is generalized (line 4). This ensures that the resulting type scheme is well-formed.

For every type τ and basis B with $B \vdash \tau$, we write $Below(B, (B_0, \tau))$ as a shorthand for $Below(B, (B_0 \cup (frv(\tau), arreff(\tau))))$. Similarly for $Below(B, (B_0, \mu))$. Note that if $B \vdash \tau$, $B \supseteq B_0$, and $B = Below(B, (B_0, \tau))$, then B is the smallest basis which satisfies $B \supseteq B_0$ and $B \vdash \tau$, and one need not have $B_0 \vdash \tau$.

LEMMA 6.6.1. *Assume $B \vdash \tau$, $B \vdash \varphi$, $B_0 \subseteq B$, and $B = Below(B, (B_0, \tau))$. Then $(B_1, \sigma) = RegEffGen(B, B_0, \varphi)(\tau)$ succeeds and we have $B_1 \vdash \sigma$, $B_1 \vdash \varphi$, and $B \supseteq B_1 \supseteq B_0$.*

PROOF. The call $(B_1, \sigma) = RegEffGen(B, B_0, \varphi)(\tau)$ clearly succeeds and $B \supseteq B_1 \supseteq B_0$ follows from the definition of *Below*. From $B = Below(B, (B_0, \tau))$ and the definition of Q'_0 and Φ'_0 we get $\{\rho_1, \dots, \rho_k\} \subseteq pfrv(\tau)$ and $\{\epsilon_1, \dots, \epsilon_m\} \subseteq pfev(\tau)$. Let $B' = (Q \setminus Q_1, \Phi \setminus \Phi_1)$. Then $B' = bound(\sigma)$. 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 6.4.1. Also, $B_1 \vdash \varphi$ holds by the definition of Q'_0 and Φ'_0. □

The use of (Q'_0, Φ'_0) and not (Q_0, Φ_0) as an argument to *Below* in line 4 causes incompleteness, as described earlier. We are not aware of any other source of incompleteness in the algorithms.

7. THE TARGET LANGUAGE

We first define the fully annotated target language of region inference. We then impose a special implementation-oriented type system on this language. By implementation-oriented we mean that the inference system formalizes that target expressions are typable using the usual region inference rules [Tofte and Talpin 1994] and in addition satisfy invariants that are used in the proof of correctness of $\mathcal{S}$ and $\mathcal{R}$.

7.1 Typed, Region-Annotated Terms

The typed target language has two mutually recursive phrase classes, RTypedExp and RTrip:

Target triple, $t \in$ RTrip:

$$t \ ::= e : \mu : \varphi$$

Typed, region-annotated expression, $e \in$ RTypedExp:

$$\begin{aligned} e \ ::= \ & x \mid f_{il}\ \texttt{at}\ \rho \mid \lambda x : \mu\,.\,t \mid t\,t \\ \mid\ & \texttt{letrec}\ f : (\sigma, \rho)(x)\ \texttt{=}\ t_1\ \texttt{in}\ t_2\ \texttt{end} \\ \mid\ & \texttt{letregion}\ B\ \texttt{in}\ t\ \texttt{end} \end{aligned}$$

The members of RTypedExp and RTrip are referred to as *typed, region-annotated expressions*, and *target triples*, respectively.

Target triples may be regarded as untyped region-annotated terms (RegionExp, see Section 3) annotated with additional information. For example, f_{il} `at` ρ is the fully annotated version of f `[`$\rho_1, \dots, \rho_k$`] at` ρ, assuming that il takes the form $([\tau_1, \dots, \tau_n], [\rho_1, \dots, \rho_k], [\epsilon_1.\varphi_1, \dots, \epsilon_m.\varphi_m])$. In the `letrec` expression, the bound variables of σ are precisely the formal parameters of f in the corresponding untyped region-annotated term. The scope of the bound variables of σ includes t_1. Finally, $\texttt{letregion}\ B\ \texttt{in}\ \ldots\ \texttt{end}$ is the fully explicit form of $\texttt{letregion}\ \rho_1,\ldots,\rho_k\ \texttt{in}\ \ldots\ \texttt{end}$, where $\{\rho_1,\ldots,\rho_k\} = Q\,of\,B$. The fully explicit form allows `letregion` to introduce arrow effects as well as region variables. The arrow effects have no significance at runtime, but they arise naturally during region inference. Some "$\texttt{at}\rho$" annotations in region-annotated terms are omitted from typed region annotated terms, since they can be read from the type and place of the expression and would hence be redundant. For example, $(\lambda x : \mu_x.t)\,\texttt{at}\,\rho : \mu, \varphi$ would be redundant, since ρ must be equal to the second component of μ.

Furthermore, we define a function ML : RTrip $\rightarrow$ TypedExp and a function ML : RTypedExp $\rightarrow$ TypedExp for erasing region and effect information (TypedExp was defined in Section 3):

$$
\begin{array}{l}
\mathrm{ML}(e : \mu : \varphi) = \mathrm{ML}(e)\\
\mathrm{ML}(x) = x\,\texttt{[]}\\
\mathrm{ML}(f_{([\tau_1,\ldots,\tau_n],\_,\_)}\ \texttt{at}\ \rho) = f\,\texttt{[}\mathrm{ML}(\tau_1)\texttt{,}\ldots\texttt{,}\mathrm{ML}(\tau_n)\texttt{]}\\
\mathrm{ML}(\lambda x : \mu.t) = \lambda x : \mathrm{ML}(\mu).\,\mathrm{ML}(t)\\
\mathrm{ML}(t_1\ t_2) = \mathrm{ML}(t_1)\ \mathrm{ML}(t_2)\\
\mathrm{ML}(\texttt{letrec}\ f : (\sigma,\rho)\texttt{(}x\texttt{)}\ \texttt{=}\ t_1\ \texttt{in}\ t_2\ \texttt{end}) =\\
\quad \texttt{letrec}\ f : \mathrm{ML}(\sigma)\texttt{(}x\texttt{)}\ \texttt{=}\ \mathrm{ML}(t_1)\ \texttt{in}\ \mathrm{ML}(t_2)\ \texttt{end}\\
\mathrm{ML}(\texttt{letregion}\ B\ \texttt{in}\ t\ \texttt{end}) = \mathrm{ML}(t).
\end{array}
$$

7.2 Cones

During fixed point resolution, binding occurrences of region and effect variables can migrate outward in the target triple or disappear altogether. This phenomenon is crucial for the understanding of the correctness proof of the algorithm.

Here is an example which illustrates the issue. Assume that g is a `letrec`-bound identifier which $\mathcal{S}$ has ascribed the type scheme $\sigma = \forall\rho_1\rho_2\epsilon.(\texttt{int},\rho_1) \xrightarrow{\epsilon.\{\rho_2\}} (\texttt{int},\rho_2)$. Algorithm $\mathcal{S}$ will transform the source expression[1]

$$\texttt{let}\ z : \texttt{int}\ \texttt{=}\ \texttt{5}\ \texttt{in}\ \lambda y : \alpha.(g(z); y)\ \texttt{end}$$

into

$$
\begin{array}{rl}
e_1 = & \texttt{letregion}\ \rho_5\\
& \texttt{in let}\ z : (\texttt{int},\rho_5)\ \texttt{= 5 at}\ \rho_5\\
& \quad \texttt{in}\ (\lambda y : (\alpha,\rho_{11}).\texttt{(letregion}\ \rho_8,\rho_9\quad \texttt{(**)}\\
& \qquad\qquad\qquad\quad \texttt{in}\ g\texttt{[}\rho_5\texttt{,}\rho_8\texttt{] at}\ \rho_9\ z\\
& \qquad\qquad\qquad\quad \texttt{end;}\\
& \qquad\qquad\qquad\quad y\texttt{)}\\
& \qquad \texttt{) at}\ \rho_{10}\\
& \quad \texttt{end}\\
& \texttt{end}
\end{array}
$$

which has type $(\alpha,\rho_{11}) \xrightarrow{\epsilon_{10}.\emptyset} (\alpha,\rho_{11})$. (For brevity, we show only the underlying region-annotated terms, not the fully decorated target triples.) But suppose $\mathcal{R}$ later discovers that g has the less general type scheme $\sigma' = \forall\rho\epsilon.(\texttt{int},\rho) \xrightarrow{\epsilon.\{\rho\}} (\texttt{int},\rho)$.

[1] We write $e_1; e_2$ as a shorthand for $(\lambda\, v.e_2)e_1$, where v is not free in e_2.

$e_2 =$
```
      let z : (int, ρ5) = 5 at ρ5
      in (λy : (α, ρ11).(letregion ρ9
                        in g[ρ5,ρ5] at ρ9 z
                        end;
                        y)
         ) at ρ10
      end
```

Fig. 7. After the second iteration, ρ_5 is no longer `letregion`-bound.

Then the call of g must put its result in ρ_5. Note that ρ_5 cannot be bound by `letregion` at (**), since ρ_5 occurs free in the type of z. The next annotation of the expression is shown in Figure 7. Here e_2 has type $(\alpha, \rho_{11}) \xrightarrow{\epsilon_{10}.\{\rho_5\}} (\alpha, \rho_{11})$. So we see that the `letregion`-bindings of ρ_5 and ρ_8 "disappear." More importantly, while ρ_5 did not appear in the type of e_1, it must occur in any basis in which the type of e_2 is consistent. Informally, ρ_5 has migrated outward in the term.

To sum up, as regions migrate outward, the basis in which a given subexpression is considered appears to grow! At first, this looks like a demonstration that our idea of ensuring termination by restricting $\mathcal{R}$ to perform contractions of a basis (discussed in Section 6.3), will not work. However, the saving fact is that if we compute the union of the basis, B, in which e_1 is considered, and all the bases that are discharged by subexpressions of e_1, call the union A, then A is contracted even if B is not necessarily contracted. In the example, we have Q *of* $A = \{\rho_5, \rho_8, \rho_9\}$ which is contracted to $\{\rho_5, \rho_9\}$.

To allow us to distinguish between the basis, B, in which the type of a target triple t is consistent and the larger basis, A, which also includes all bases discharged locally within t, we introduce the notion of a *cone*. Intuitively, a cone shows the layering of bindings in a proof tree; see Figure 8. The term "cone" comes from the shape of proof trees.

Formally, a *cone* is a pair (A, B) of bases such that $\vdash B$, $\vdash A$, and $A \supseteq B$. We usually write cones in the form $\binom{A}{B}$. Here A represents the union of all binding layers in the proof tree and B is the basis currently under consideration. To make the statement of the theorems easier, we put cones into the inference rules. In a sentence $\binom{A}{B}, TE \vdash e : \mu : \varphi$, B is a basis in which TE, μ, and φ are consistent, while A is the basis obtained from B and e by forming the union of B and all the bases that are discharged locally within e.

A central idea in the termination argument is to use A (not B) to bound the computation. To do this, we need to be sure that bases that are discharged by different subexpressions are kept distinct. Formally, let $\binom{B_1'}{B_1}$ and $\binom{B_2'}{B_2}$ be cones. Write B_i in the form (Q_i, Φ_i) and write B_i' in the form (Q_i', Φ_i'), $i = 1, 2$. The *sum* of $\binom{B_1'}{B_1}$ and $\binom{B_2'}{B_2}$, written $\binom{B_1'}{B_1} \uplus \binom{B_2'}{B_2}$, is defined by

$$\begin{pmatrix} B_1' \\ B_1 \end{pmatrix} \uplus \begin{pmatrix} B_2' \\ B_2 \end{pmatrix} = \begin{cases} \binom{B_1' \cup B_2'}{B_1} & \text{if } B_1 = B_2 \text{ and } Dom(B_1') \cap Dom(B_2') \subseteq Dom(B_1) \\ \text{undefined} & \text{otherwise.} \end{cases}$$

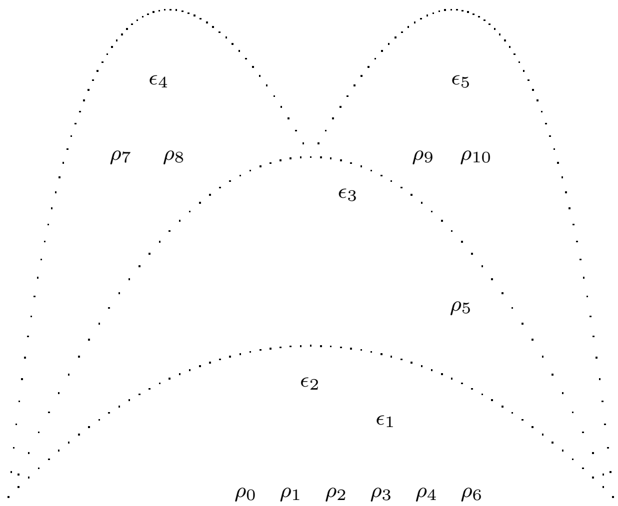

Fig. 8. A cone for the effect graph for the program in Figure 3. Levels of binding are separated by dotted lines. There can be arrows from nodes in higher levels to nodes in lower levels, but not the opposite way.

7.3 The Inference Rules

We distinguish between *pre-typing* and *well-typing*. Spreading an expression gives a pre-typed expression, while fixed point resolution transforms pre-typed expressions into well-typed expressions, through a sequence of pre-typed expressions. Every well-typed expression is also pre-typed. A pre-typed expression is well-typed except that for annotated subexpressions of the form f_{il} $\mathtt{at}\,\rho : \mu, \varphi$, the μ may differ from the result of instantiating the type scheme for f via il.

In the inference rules we use the following auxiliary functions. 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

$$il = ([\tau_1, \dots, \tau_n], [\rho_1, \dots, \rho_k], [\epsilon_1.\varphi_1, \dots, \epsilon_m.\varphi_m])$$

be an instantiation list of the same arity as σ. The $zip(\sigma, il)$ is the substitution $(\{\alpha_1 \mapsto \tau_1, \dots, \alpha_n \mapsto \tau_n\}, \{\rho'_1 \mapsto \rho_1, \dots, \rho'_k \mapsto \rho_k\}, \{\epsilon'_1 \mapsto \epsilon_1.\varphi_1, \dots, \epsilon'_m \mapsto \epsilon_m.\varphi_m\})$; $zip(\sigma, il)$ is undefined if $arity(\sigma) \neq arity(il)$. Second, we define $dropTyvars(\sigma)$ to be $\forall\rho_1\cdots\rho_k\epsilon_1\cdots\epsilon_m.\tau$ where $\forall\alpha_1\cdots\alpha_n\rho_1\cdots\rho_k\epsilon_1\cdots\epsilon_m.\tau = \sigma$.

A *type environment* is a finite map TE : Var $\rightarrow$ TypeScheme $\times$ RegVar. We say that TE is *consistent in basis* B, written $B \vdash TE$, if for all variables $x \in Dom(TE)$ we have $B \vdash TE(x)$, where $B \vdash (\sigma, \rho)$ means $B \vdash \sigma \wedge B \vdash \rho$. For arbitrary semantic objects we write $B \vdash (o_1, \dots, o_n)$ to mean $B \vdash o_1 \wedge \cdots \wedge B \vdash o_n$.

The rules for pretyping are

$$\frac{B \vdash TE \qquad TE(x) = \mu}{\binom{B}{B}, TE \vdash x : \mu : \emptyset} \tag{14}$$

$$\frac{\begin{array}{c} B \vdash (TE, il, \mu) \qquad TE(f) = (\sigma, \rho) \qquad \mu = (\tau, \rho') \\ S = zip(\sigma, il) \qquad \tau_0 = body(\sigma) \qquad \mathrm{ML}(\tau) = \mathrm{ML}(S(\tau_0)) \end{array}}{\binom{B}{B}, TE \vdash f_{il}\ \texttt{at}\ \rho' : \mu : \{\rho, \rho'\}} \tag{15}$$

$$\frac{\begin{array}{c} \binom{A}{B}, TE + \{x \mapsto \mu_x\} \vdash t_1 \qquad t_1 = e_1 : \mu_1 : \varphi_1 \\ \varphi_0 \supseteq \varphi_1 \qquad B \vdash TE \qquad (\{\rho\}, \{\epsilon.\varphi_0\}) \subseteq B \end{array}}{\binom{A}{B}, TE \vdash \lambda x : \mu_x.t_1 : (\mu_x \xrightarrow{\epsilon.\varphi_0} \mu_1, \rho) : \{\rho\}} \tag{16}$$

$$\frac{\begin{array}{c} \binom{A_1}{B}, TE \vdash t_1 \qquad t_1 = e_1 : (\mu' \xrightarrow{\epsilon_0.\varphi_0} \mu, \rho_0) : \varphi_1 \\ \binom{A_2}{B}, TE \vdash t_2 \qquad t_2 = e_2 : \mu' : \varphi_2 \qquad \binom{A_3}{B} = \binom{A_1}{B} \uplus \binom{A_2}{B} \end{array}}{\binom{A_3}{B}, TE \vdash t_1\ t_2 : \mu : \varphi_1 \cup \varphi_2 \cup \{\epsilon_0, \rho_0\} \cup \varphi_0} \tag{17}$$

$$\frac{\begin{array}{c} B \vdash (TE, \varphi_1) \qquad \hat{\sigma} = dropTyvars(\sigma) \qquad btv(\sigma) \cap ftv(TE) = \emptyset \\ B_1 = bound(\sigma) \qquad B \uplus B_1 \text{ exists} \qquad \tau_0 = body(\sigma) = \mu_x \xrightarrow{\epsilon_0.\varphi_0} \mu_1 \\ \binom{A_1}{B \uplus B_1}, TE + \{f \mapsto (\hat{\sigma}, \rho_1)\} \vdash \lambda\, x : \mu_x.t_1 : (\tau_0, \rho_1) : \varphi_1 \\ \binom{A_2}{B}, TE + \{f \mapsto (\sigma, \rho_1)\} \vdash e_2 : \mu : \varphi_2 \qquad \binom{A_3}{B} = \binom{A_1}{B} \uplus \binom{A_2}{B} \end{array}}{\begin{array}{l} \binom{A_3}{B}, TE \vdash \texttt{letrec}\ f : (\sigma, \rho_1)(x) \texttt{=}\ t_1 \\ \qquad\qquad\quad \texttt{in}\ e_2 : \mu : \varphi_2\ \texttt{end} : \mu : \varphi_1 \cup \varphi_2 \end{array}} \tag{18}$$

$$\frac{\begin{array}{c} B \uplus B_1 \text{ exists} \qquad \binom{A}{B \uplus B_1}, TE \vdash e_1 : \mu : \varphi_1 \\ B \vdash (TE, \mu) \qquad \varphi = Observe(B, \varphi_1) \end{array}}{\binom{A}{B}, TE \vdash \texttt{letregion}\ B_1\ \texttt{in}\ e_1 : \mu : \varphi_1\ \texttt{end} : \mu : \varphi}. \tag{19}$$

In Rule 14, the resulting effect is empty, since, in a call-by-value language, simply referring to a λ-bound variable x accesses the environment, but not the store. For a `letrec`-bound variable f, however, we access the region ρ in which the region-polymorphic function resides and then produce a closure in a perhaps different region ρ'; see Rule 15. In Rule 16, the effect of t_1 becomes the latent effect of the lambda abstraction. The rule allows for an increase of effects, just before they are put onto function arrows.

The side-condition $\binom{A_3}{B} = \binom{A_1}{B} \uplus \binom{A_2}{B}$, which appears in some of the rules, expresses that the region and effect variables bound locally within two subphrases must be distinct. In Rule 18 notice that t_1 is analyzed in an environment where f may be polymorphic in regions and effects, but not in types. The condition that $B \uplus B_1$ exists (Section 5.1) forces the bound region and effect variables of σ to be chosen distinct from the ones in B; since $B \vdash (TE, \varphi_1)$ implies that B is consistent, the side-conditions force σ to be consistent (Section 6.4). The mention of B_1 in $\binom{A_1}{B \uplus B_1}$ introduces the bound variables of $\hat{\sigma}$ which appear free in $\lambda x : \mu_x.t_1 : (\tau_0, \rho_1) : \varphi_1$.

Similarly, in Rule 19 we introduce B_1 into the cone $\binom{A}{B \uplus B_1}$, since the variables in $Dom(B_1)$ (let us call them the `letregion`-bound variables) can appear free in e_1.

The two conditions that $B \uplus B_1$ exist and that $B \vdash (TE, \mu)$ prevent the `letregion`-bound variables from occurring free in TE or μ. They may occur in φ_1, although this is not required. By the definition of *Observe* (8), every variable in φ is free in B and, since B is consistent and $B \uplus B_1$ exists, we have that no `letregion`-bound variable is a member of φ or occurs free in B.

The rules for well-typed expressions are obtained by replacing Rule 15 by the slightly more demanding rule

$$\frac{\begin{array}{ccc} B \vdash (TE, il, \mu) & TE(f) = (\sigma, \rho) & \mu = (\tau, \rho') \\ S = zip(\sigma, il) & \tau_0 = body(\sigma) & \tau = S(\tau_0) \end{array}}{\binom{B}{B}, TE \vdash f_{il}\ \mathtt{at}\ \rho' : \mu : \{\rho, \rho'\}}. \tag{20}$$

We say that $e : \mu : \varphi$ is *pre-typed in* $\binom{A}{B}$ *and* TE, written $\binom{A}{B}, TE \vdash_{\mathrm{P}} e : \mu : \varphi$, if $\binom{A}{B}, TE \vdash e : \mu : \varphi$ can be inferred using Rules 14–19; we say that $e : \mu : \varphi$ is *well-typed in* $\binom{A}{B}$ *and* TE, written $\binom{A}{B}, TE \vdash_{\mathrm{W}} e : \mu : \varphi$, if $\binom{A}{B}, TE \vdash e : \mu : \varphi$ can be inferred using Rules 14 and 16–20.

In Rule 18, notice that the body of the type scheme for f is the same as the type which annotates $\lambda x : \mu_x.t_1$. This does not imply that the recursive declaration is well-typed, however, because the free occurrences of f in t_1 and t_2 may be only pre-typed as opposed to well-typed.

LEMMA 7.3.1. *If* $\binom{A}{B}, TE \vdash_P e : \mu : \varphi$ *or* $\binom{A}{B}, TE \vdash_W e : \mu : \varphi$, *then* $B \vdash (TE, \mu, \varphi)$.

PROOF. By induction on the depth of inference. □

LEMMA 7.3.2. *Let* $\binom{A}{B}$ *be a cone,* $B \vdash \varphi$, *and let* S *be a contraction of* A. *Then* $Observe(S(B), S(\varphi)) \supseteq S(Observe(B, \varphi))$.

PROOF. Immediate from the definition of *Observe*. □

7.4 Thinning Instantiation Lists

The relation $il\, R^{\sigma \sqsupseteq \sigma'}\, il'$ defined in Section 6.2 can be extended to a relation on expressions as follows. Let f be a program variable and let t be a target triple in which every free occurrence of f is annotated by an instantiation list which has the same arity as σ. We say that t' is obtained from t by *lowering the polymorphism of* f *from* σ *to* σ', written $t\, R_f^{\sigma \sqsupseteq \sigma'}\, t'$, if t' can be obtained from t by replacing every free occurrence f_{il} in t by $f_{il'}$ where il' satisfies $il\, R^{\sigma \sqsupseteq \sigma'}\, il'$. It is straightforward to compute such a t' from t and $R^{\sigma \sqsupseteq \sigma'}$.

LEMMA 7.4.1. *If* ∇ *proves* $\binom{A}{B}, TE + \{f \mapsto (\sigma, \rho)\} \vdash_P t$ *and* $\sigma \sqsupseteq \sigma'$, $B \vdash \sigma'$, *and* $t\, R_f^{\sigma \sqsupseteq \sigma'}\, t'$, *then there exists a proof* ∇' *of* $\binom{A}{B}, TE + \{f \mapsto (\sigma', \rho)\} \vdash_P t'$ *with* $depth(\nabla) = depth(\nabla')$.

PROOF. The proof is by induction on the depth of inference of $\binom{A}{B}, TE + \{f \mapsto (\sigma, \rho)\} \vdash_{\mathrm{P}} t$. There is one case for each of the rules (14)–(19). We show only the case for Rule 15; the remaining cases are straightforward. Assume $\binom{A}{B}, TE +$

$\{f \mapsto (\sigma, \rho)\} \vdash_{\mathrm{P}} t$ has been inferred using Rule 15. Then $A = B$ and there exist f', il ρ', φ, and μ such that $t = f'_{il}$ `at` $\rho' : \mu : \varphi$. Only the case $f' = f$ is interesting. By the assumptions of Rule 15 there exist τ_0, S, and τ such that $B \vdash (TE + \{f \mapsto (\sigma, \rho)\}, il, \mu)$, $\mu = (\tau, \rho')$, $S = zip(\sigma, il)$, $\tau_0 = body(\sigma)$, $\varphi = \{\rho, \rho'\}$, and $\mathrm{ML}(\tau) = \mathrm{ML}(S(\tau_0))$.

Since $B \vdash \sigma'$ we have $B \vdash TE + \{f \mapsto (\sigma', \rho)\}$. Moreover, since $t\, R_f^{\sigma \sqsupseteq \sigma'}\, t'$ there exists an il' such that $t' = f_{il'}$ `at` $\rho' : \mu : \varphi$ and $il\, R^{\sigma \sqsupseteq \sigma'}\, il'$. Now $B \vdash il$ and $il\, R^{\sigma \sqsupseteq \sigma'}\, il'$ implies $B \vdash il'$. Moreover, let $\tau'_0 = body(\sigma')$ and let $S' = zip(\sigma', il')$ (which exists). Then we have $\mathrm{ML}(\tau) = \mathrm{ML}(S(\tau_0)) = \mathrm{ML}(S'(\tau'_0))$, since $\sigma \sqsupseteq \sigma'$. Thus Rule 15 gives the desired pre-typing $\binom{B}{B}, TE + \{f \mapsto (\sigma', \rho)\} \vdash_{\mathrm{P}} t'$. $\square$

We remark that the corresponding statement with $\vdash_{\mathrm{W}}$ for $\vdash_{\mathrm{P}}$ is false. When the type scheme associated with f is made less polymorphic, subsequent unification between τ and the instance of σ' given by il' may be required to obtain well-typing. (This unification is performed in line 4 of Algorithm $\mathcal{R}$ in Section 9.2).

8. SPREADING EXPRESSIONS

We first introduce some auxiliary functions which $\mathcal{S}$ uses. Then we present $\mathcal{S}$ and prove that it produces pre-typed target triples.

8.1 Auxiliary Functions

The types of the auxiliary functions are

$$\begin{aligned}
\mathit{freshRegVar} &: \mathrm{Basis} \to \mathrm{RegVar}\\
\mathit{freshEffVar} &: \mathrm{Basis} \to \mathrm{EffVar}\\
\mathit{freshType} &: \mathrm{Basis} \times \mathrm{MLType} \to \mathrm{Basis} \times \mathrm{Type}\\
\mathit{freshType}^{(n)} &: \mathrm{Basis} \times \mathrm{MLType}^n \to \mathrm{Basis} \times \mathrm{Type}^n\\
\mathit{freshTypeWithPlace} &: \mathrm{Basis} \times \mathrm{MLType} \to \mathrm{Basis} \times \mathrm{TypeWithPlace}\\
\mathit{newInstance} &: \mathrm{Basis} \times \mathrm{MLTypeScheme} \times \mathrm{MLType}^n \to\\
&\qquad \mathrm{Subst} \times \mathrm{Basis} \times \mathrm{Type} \times \mathrm{InstList}.
\end{aligned}$$

All the functions, except *newInstance*, are total. $\mathit{freshRegVar}(B)$ returns a region variable ρ satisfying $\rho \notin \mathit{frv}\, B$, and $\mathit{freshEffVar}(B)$ returns an effect variable ϵ satisfying $\epsilon \notin \mathit{fev}\, B$. *freshType* can be any function that satisfies that if $(B', \tau) = \mathit{freshType}(B, \underline{\tau})$ and $B = (Q, \Phi)$ is consistent, then $B \subseteq B'$, $B' \vdash \mu$, $\mathrm{ML}(\mu) = \underline{\tau}$, no region or effect variable occurs twice in μ, and, writing B' in the form (Q', Φ'), we have $Q' \setminus Q = \mathit{frv}(\mu)$ and $\Phi' \setminus \Phi = \{\epsilon.\emptyset \mid \epsilon \in \mathit{fev}(\mu)\}$. $\mathit{freshType}^{(n)}$ extends *freshType* to n-tuples. *freshTypeWithPlace* is similar to *freshType* but returns a type with (a fresh) place instead of just a type. *newInstance* is defined in Figure 9, together with the function *Retract*, which introduces `letregion`-expressions. To simplify the correctness proofs for $\mathcal{R}$, *Retract* produces a `letregion` expression even when $B_1 \setminus B_{keep}$ is empty. We have omitted such empty bindings in examples.

8.2 Algorithm $\mathcal{S}$

Algorithm $\mathcal{S}$ appears in Figures 10 and 11. In a call $(S, A', B', t) = \mathcal{S}(A, B, TE, \underline{e})$, $\underline{e}$ is an explicitly typed source expression (Section 3), A, B, A', and B' are bases

$$
\begin{array}{l}
\mathit{newInstance}(A \text{ as } (Q,\Phi), \sigma, (\underline{\tau}_1,\ldots,\underline{\tau}_n)) = \\
\quad \text{let } \forall\alpha_1\cdots\alpha_{n'}\rho_1\cdots\rho_k\epsilon_1\cdots\epsilon_m.\tau_0 = \sigma \quad \text{FAIL, if } n' \neq n \\
\qquad \rho'_1,\ldots,\rho'_k \text{ be distinct region variables not in } Q \\
\qquad \epsilon'_1,\ldots,\epsilon'_m \text{ be distinct effect variables not in } \mathit{Dom}(\hat{\Phi}) \\
\qquad A' = (Q \cup \{\rho'_1,\ldots,\rho'_k\},\ \Phi \cup \{\epsilon'_1.\emptyset,\ldots,\epsilon'_m.\emptyset\}) \\
\qquad (A_1, (\tau_1,\ldots,\tau_n)) = \mathit{freshType}^{(n)}(A', (\underline{\tau}_1,\ldots,\underline{\tau}_n)) \\
\qquad \mathit{il} = ([\tau_1,\ldots,\tau_n], [\rho'_1,\ldots,\rho'_k], [\epsilon'_1.\emptyset,\ldots,\epsilon'_k.\emptyset]) \\
\qquad (S^e_1, \tau) = \mathit{inst}(\sigma, \mathit{il}) \\
\quad \text{in} \\
\qquad (S^e_1, A_1, \tau, S^e_1(\mathit{il})) \\
\quad \text{end}
\end{array}
$$

$$
\begin{array}{l}
\mathit{Retract}(S, A, B_1, B_0, e : \mu : \varphi) = \\
\quad \text{let} \\
\qquad B_{keep} = \mathit{Below}(B_1, (B_0, \mu)) \\
\qquad \varphi' = \mathit{Observe}(B_{keep}, \varphi) \\
\quad \text{in} \\
\qquad (S, A, B_{keep}, \texttt{letregion } B_1 \setminus B_{keep} \texttt{ in } e : \mu : \varphi \texttt{ end} : \mu : \varphi') \\
\quad \text{end}
\end{array}
$$

Fig. 9. Algorithms *newInstance* and *Retract*.

(Section 5.1), and TE is a type environment (Section 7.3). In the soundness theorem we assume that TE is consistent in B (Section 7.3), $\underline{e}$ is well-typed in ML(TE) (Figure 1), and that A is a particularly simple kind of basis in which arrow effects are handles of empty effects only. (Then one can think of A as representing the variables generated thus far; this is useful for formalizing what it means to pick "fresh" variables.) The soundness theorem states that (under suitable conditions) $\mathcal{S}$ succeeds, and moreover

(1) $A' \supseteq A$: more variables may have been generated;
(2) $S(TE)$ is consistent in B', t is pretyped in B', and $B' \supseteq S(B)$: the basis B may have "grown" to contain fresh region and effect variables occurring in t;
(3) S is a contraction of $B'' = (A' \setminus A) \cup B$, i.e., a contraction of the brand new variables $(A' \setminus A)$ and the old ones in B.

Note that $\mathcal{S}$ returns a substitution. It arises, for example, from instantiation of type schemes (line 2 in $\mathcal{S}$). Since $\mathcal{S}$ has to return a substitution, we make it do as much region inference as possible right away, e.g., the unification in line 13 and the introduction of `letregion`-expressions wherever there is a possibility for it (lines 15 and 28). Thus the output term t becomes pre-typed, i.e., well-typed apart from fixed-point resolution. Note that $\mathcal{S}$ does not perform any iteration when processing `letrec`.

Algorithm $\mathcal{S}$ assumes that every nonbinding occurrence of a variable in the input term $\underline{e}$ has been classified as either *letrec-bound* or *otherwise bound*. Free variables in $\underline{e}$ may be classified as either, as long as all free occurrences of the same variable are given the same classification.[2]

[2]In practice, the classification of free variables is determined by the initial basis or by previously compiled program units. Also, the classification can be kept in TE, so that $TE(x)$ takes the form $((\sigma,\rho), \mathit{classification})$, but we prefer not to clutter the formal presentation by this extra component.

$$
\begin{array}{l}
\mathcal{S}(A, B, TE, \underline{e}) = \text{case } \underline{e} \text{ of} \\
\quad x[\underline{\tau}_1, \ldots, \underline{\tau}_n] \Rightarrow \\
\qquad \text{let} \\
\qquad\quad (\sigma, \rho) = TE(x) \\
\qquad \text{in if } x \text{ is letrec-bound then} \\
\qquad\qquad \text{let } (S_1^e, A_1, \tau, il_1) = \mathit{newInstance}(A, \sigma, [\underline{\tau}_1, \ldots, \underline{\tau}_n]) \\
\qquad\qquad\qquad \rho' = \mathit{freshRegVar}(A_1) \\
\qquad\qquad \text{in } (S_1^e, A_1 \uplus \{\rho'\}, S_1^e(B \uplus (A_1 \setminus A)) \uplus \{\rho'\}, \\
\qquad\qquad\qquad x_{il_1} \; \texttt{at} \; \rho' : (\tau, \rho') : \{\rho, \rho'\}) \\
\qquad\qquad \text{end} \\
\qquad\quad \text{else } (\mathrm{Id}, A, B, x : TE(x) : \emptyset) \\
\qquad \text{end} \\
\mid \; \lambda x : \underline{\tau}_0 . \underline{e}_1 \Rightarrow \\
\qquad \text{let} \\
\qquad\quad (A_0, \mu_0) = \mathit{freshTypeWithPlace}(A, \underline{\tau}_0) \\
\qquad\quad (S_1, A_1, B_1, t_1 \text{ as } (e_1 : \mu_1 : \varphi_1)) = \\
\qquad\qquad\quad \mathcal{S}(A_0, B \uplus (A_0 \setminus A), TE + \{x \mapsto \mu_0\}, \underline{e}_1) \\
\qquad\quad \epsilon = \mathit{freshEffVar}(A_1) \\
\qquad\quad \rho = \mathit{freshRegVar}(A_1) \\
\qquad\quad A_2 = (\{\rho\}, \{\epsilon.\varphi_1\}) \\
\qquad\quad S = \{\epsilon \mapsto \epsilon.\varphi_1\} \circ S_1 \\
\qquad \text{in} \\
\qquad\quad (S, A_1 \uplus (\{\rho\}, \{\epsilon.\emptyset\}), B_1 \uplus A_2, (\lambda x : S_1(\mu_0).t_1) : (S_1(\mu_0) \xrightarrow{\epsilon.\varphi_1} \mu_1, \rho) : \{\rho\}) \\
\qquad \text{end} \\
\mid \; \underline{e}_1 \;\; \underline{e}_2 \Rightarrow \\
\qquad \text{let} \\
\qquad\quad (S_1, A_1, B_1, t_1 \text{ as } (e_1 : (\mu_2 \xrightarrow{\epsilon.\varphi_0} \mu_1, \rho_0) : \varphi_1)) = \mathcal{S}(A, B, TE, \underline{e}_1) \\
\qquad\quad (S_2, A_2, B_2, t_2 \text{ as } (e_2 : \mu_2' : \varphi_2)) = \mathcal{S}(A_1, B_1, S_1(TE), \underline{e}_2) \\
\qquad\quad S_3 = \mathit{unifyMu}(S_2(\mu_2), \mu_2') \\
\qquad\quad S = S_3 \circ S_2 \circ S_1 \\
\qquad \text{in } \mathit{Retract}(S, A_2, S_3 B_2, SB, S_3(S_2(t_1) \quad t_2) \\
\qquad\qquad\qquad : S_3(S_2(\mu_1)) : S_3(S_2(\{\epsilon, \rho_0\} \cup \varphi_0 \cup \varphi_1) \cup \varphi_2)) \\
\qquad \text{end}
\end{array}
$$

Fig. 10. Algorithm $\mathcal{S}$ (first half).

For closed expressions we have the following main result.

THEOREM 8.2.1. *Assume* $\{\} \vdash \underline{e} : \underline{\tau}$. *Then* $(S, A', B', e : \mu : \varphi) = \mathcal{S}(B_\emptyset, B_\emptyset, \{\}, \underline{e})$ *succeeds,* $\mathrm{ML}(e) = \underline{e}$, *and* $\binom{A'}{B'}, \{\} \vdash_P e : \mu : \varphi$.

This theorem is a consequence of a stronger theorem, which is stated and proved in the electronic appendix.

9. FIXED-POINT RESOLUTION

9.1 Saturated Expressions

We say that a target expression is strongly saturated if it has a `letregion` immediately enclosing every application and `letrec`-expression. (Those are the forms for which there is a possibility for discharging region variables and arrow effects by `letregion`.) Precisely, we define the *saturated expressions* (ranged over by s) and *strongly saturated expressions* (ranged over by ss) to be the target triples generated by the grammar:

$\mid$ `letrec` $f : \forall\alpha_1 \cdots \alpha_n.\underline{\tau}_0$`(`$x$`) =` $\underline{e}_1$ `in` $\underline{e}_2$ `end` $\Rightarrow$
let
18 $\underline{\tau}_x \to \underline{\tau}_1 = \underline{\tau}_0$
19 $(A_0, \tau_0) = \mathit{freshType}(A, \underline{\tau}_0)$
20 $\sigma = \forall\alpha_1 \cdots \alpha_n \rho_1 \cdots \rho_k \epsilon_1 \cdots \epsilon_m.\tau_0$
where $\{\rho_1, \ldots, \rho_k\} = \mathit{frv}(\tau_0)$ and $\{\epsilon_1, \ldots, \epsilon_m\} = \mathit{fev}(\tau_0)$
21 $\hat{\sigma} = \forall\rho_1 \cdots \rho_k \epsilon_1 \cdots \epsilon_m.\tau_0$
22 $\rho = \mathit{freshRegVar}(A_0)$
23 $(S_1, A_1, B_1, t_1 \text{ as } ((\lambda x : \mu_x.t'_1) : (\tau_1, \rho_1) : \varphi_1)) =$
$\mathcal{S}(A \uplus \{\rho\}, B \uplus \{\rho\}, TE + \{f \mapsto (\hat{\sigma}, \rho)\}, \lambda x : \underline{\tau}_x.\underline{e}_1)$
24 $S_2 = \mathit{unifyRho}(S_1\rho, \rho_1)$
25 $(B_2, \sigma_1) = \mathit{RegEffGen}(S_2(B_1), S_2(S_1(B \uplus \{\rho\})), S_2(\varphi_1))(S_2(\tau_1))$
26 t''_1 be such that $S_2(t'_1)\ R^{\hat{\sigma} \sqsupseteq \sigma_1}\ t''_1$
27 $(S_3, A_3, B_3, t_2 \text{ as } (e_2 : \mu : \varphi_2)) =$
$\mathcal{S}(A_1, B_2, S_2(S_1(TE)) + \{f \mapsto (\forall\alpha_1 \cdots \alpha_n.\sigma_1, S_2\rho_1)\}, \underline{e}_2)$
$S = S_3 \circ S_2 \circ S_1$
28 in $\mathit{Retract}(S, A_3, B_3, SB,$
`letrec` $f : (S_3(\forall\alpha_1 \cdots \alpha_n.\sigma_1), S_3(S_2(\rho_1)))$`(`$x$`) =` $S_3(t''_1)$
`in` t_2 `end` $: \mu : S_3(S_2(\varphi_1)) \cup \varphi_2)$
end

Fig. 11. Algorithm $\mathcal{S}$ (second half)

$$
\begin{aligned}
ss &::= x : \mu : \varphi \mid f_{il}\ \texttt{at}\ \rho : \mu : \varphi \\
&\quad\mid (\lambda x : \mu_x.ss) : \mu : \varphi \mid \texttt{letregion}\ B\ \texttt{in}\ s\ \texttt{end} : \mu : \varphi
\end{aligned}
$$

$$
\begin{aligned}
s &::= (ss\ ss) : \mu : \varphi \\
&\quad\mid \texttt{letrec}\ f : (\sigma, \rho)(x)\ \texttt{=}\ ss\ \texttt{in}\ ss\ \texttt{end} : \mu : \varphi \\
&\quad\mid ss
\end{aligned}
$$

It is easy to see that all target expressions produced by $\mathcal{S}$ are strongly saturated. As a consequence, $\mathcal{R}$ does not have to introduce new `letregion` expressions.

9.2 Algorithm $\mathcal{R}$

The fixed point resolution algorithm $\mathcal{R}$ is defined in Figures 12 and 13. The most interesting case of algorithm $\mathcal{R}$ is the case for `letrec` (Figure 13). The recursive function $\mathcal{R}^{rec}$ implements the Mycroft iteration. Every change in the arity of the type scheme of the recursive function is accompanied by a change in the instantiation lists (lines 31 and 34), as outlined in Section 7.4.

Algorithm $\mathcal{R}$ maps target expressions to target expressions; indeed, it preserves the structure of target expressions, mapping every application to an application, every `letregion` to a `letregion`, and so on. The only changes are in the region and effect annotations. Thus we have

LEMMA 9.2.1. *Assume $(S, B', t') = \mathcal{R}(B, TE, t)$ succeeds. If t is saturated, then t' is also saturated. If t is strongly saturated, then t' is also strongly saturated.*

Here is the theorem that states the correctness of $\mathcal{R}$.

THEOREM 9.2.2. *Let $t = e : \mu : \varphi$ be saturated. If $\binom{A}{B}, TE \vdash_P t$ then $(S, B', t'$ as $(e' : \mu' : \varphi')) = \mathcal{R}(B, TE, t)$ succeeds, S is a contraction of A, $\binom{S(A)}{B'}, S(TE) \vdash_W t'$, $\mathrm{ML}(t') = \mathrm{ML}(t)$, $B' \supseteq S(B)$, $\mu' = S(\mu)$, and $\varphi' \supseteq S(\varphi)$. Furthermore, if*

$$
\begin{array}{lll}
 & \multicolumn{2}{l}{\mathcal{R}(B, TE, t \text{ as } (e : \mu : \varphi)) = \text{case } e \text{ of}} \\
 & \quad f_{il}\ \mathtt{at}\ \rho \Rightarrow & \\
 & \qquad \text{let} & \\
1 & \qquad\quad (\sigma, \_) = TE(f) & \\
2 & \qquad\quad (S_1^e, \tau_1) = inst(\sigma, il) & \\
3 & \qquad\quad (\tau, \_) = \mu & \\
4 & \qquad\quad S_2 = unifyTy(\tau_1, S_1^e(\tau)) & \text{(* See Section 7.4 *)} \\
5 & \qquad\quad S = S_2 \circ S_1^e & \\
 & \qquad \text{in } (S, S(B), S(t)) & \\
 & \qquad \text{end} & \\
 & \mid\ x \Rightarrow\ (\mathrm{Id}, B, t) & \\
 & \mid\ \lambda x : \mu_x . t_1 \Rightarrow & \\
 & \qquad \text{let} & \\
6 & \qquad\quad (\mu_2 \xrightarrow{\epsilon.\varphi_0} \mu_1, \_) = \mu & \\
7 & \qquad\quad (S_1, B_1, t_1' \text{ as } (e_1' : \mu_1' : \varphi_1')) = \mathcal{R}(B, TE + \{x \mapsto \mu_2\}, t_1) & \\
8 & \qquad\quad \epsilon_1'.\varphi_1'' = S_1(\epsilon.\varphi_0) & \\
9 & \qquad\quad S_2 = \text{if } \varphi_1'' \supseteq \varphi_1' \text{ then Id else } \{\epsilon_1' \mapsto \epsilon_1'.(\varphi_1' \cup \varphi_1'')\} & \\
10 & \qquad\quad S = S_2 \circ S_1 & \\
 & \qquad \text{in} & \\
11 & \qquad\quad (S, S_2(B_1), (\lambda x : S(\mu_x).S_2 t_1') : S(\mu) : S(\varphi)) & \\
 & \qquad \text{end} & \\
 & \mid\ t_1\ \ t_2 \Rightarrow & \\
 & \qquad \text{let} & \\
12 & \qquad\quad (S_1, B_1, t_1' \text{ as } (e_1' : (\mu_1' \xrightarrow{\epsilon_0'.\varphi_0'} \mu_1'', \rho') : \varphi_1')) = \mathcal{R}(B, TE, t_1) & \\
13 & \qquad\quad (S_2, B', t_2' \text{ as } (e_2' : \mu_2' : \varphi_2')) = \mathcal{R}(B_1, S_1(TE), S_1(t_2)) & \\
14 & \qquad\quad \varphi' = S_2(\varphi_1' \cup \{\epsilon_0', \rho'\} \cup \varphi_0') \cup \varphi_2' & \\
15 & \qquad\quad S = S_2 \circ S_1 & \\
16 & \qquad \text{in } (S, B', ((S_2 t_1')\ t_2') : S_2(\mu) : \varphi') & \\
 & \qquad \text{end} &
\end{array}
$$

Fig. 12. Algorithm $\mathcal{R}$ (first half).

$S(A) = A$, then $S = \mathrm{Id}$, $t' = t$, and $B' = B$. Also, if t is strongly saturated, then $B' = Below(S(A), S(B))$.

PROOF. See the electronic appendix. □

Here we see the outward migration of region and effect variables: we have $B' \supseteq S(B)$; that is, the basis appears to grow, but B' is bounded above by $S(A)$, where S is a contraction of the A that was produced by $\mathcal{S}$. The conclusion that $S(A) = A$ implies $S = \mathrm{Id}$, $t' = t$, and $B' = B$ is used in the proof that $\mathcal{R}^{rec}$ terminates. Much simplified, that proof goes as follows. By Lemma 5.3.1 we get that eventually one will get $S_4(B_3 \uplus bound(\hat{\sigma_3})) = B_3 \uplus bound(\hat{\sigma_3})$. When that happens, the induction hypothesis gives $S = \mathrm{Id}$, $t_4 = t_3$, and $B_4 = B_3 \uplus bound(\hat{\sigma_3})$ so that the $(B_5, \hat{\sigma_5})$ computed in line 28 is in fact $(B_3, \hat{\sigma_3})$. But then, since $S_4 = \mathrm{Id}$, we have that the test in line 29 succeeds.

The test in line 9 is necessary in order to obtain that $S(A) = A$ implies $S = \mathrm{Id}$. The unification algorithms contain similar tests to ensure that whenever they produce a substitution which does anything at all, it strictly contracts the basis; cf. the discussion in Section 5.3.

The conclusion $\mu' = S(\mu)$ in the theorem tells us that there is no need to re-

$\mid$ `letregion` B_1 `in` t_1 `end` $\Rightarrow$
  let
    $(S, B_2, t_2 \text{ as } (e_2 : \mu_2 : \varphi_2)) = \mathcal{R}(B \uplus B_1, TE, t_1)$
    $B' = Below(B_2, S(B))$
    $\varphi' = Observe(B', \varphi_2)$
  in $(S, B', \texttt{letregion } B_2 \setminus B' \texttt{ in } t_2 \texttt{ end} : \mu_2 : \varphi')$
  end
$\mid$ `letrec` $f : (\sigma_0, \rho_0)(x)$ `=` t_1 `in` t_2 `end` $\Rightarrow$
  let
    $\forall\vec{\alpha}\vec{\rho}\vec{\epsilon}.\tau_0 = \sigma_0;\ (\mu_x \xrightarrow{\ \ } \_, \_) = \tau_0$
    $\hat{\sigma_0} = \forall\vec{\rho}\vec{\epsilon}.\tau_0$
    fun $\mathcal{R}^{rec}(B_3, \hat{\sigma_3}, t_3, S_{acc}) =$
      let
        $(S_4, B_4, t_4 \text{ as } (\_ : (\tau_4, \rho_4) : \varphi_4)) =$
          $\mathcal{R}(B_3 \uplus bound(\hat{\sigma_3}), S_{acc}(TE) + \{f \mapsto (\hat{\sigma_3}, S_{acc}\rho_0)\}, t_3)$
        $(B_5, \hat{\sigma_5}) = RegEffGen(B_4, S_4B_3, \varphi_4)(\tau_4)$
      in
        if $S_4(\hat{\sigma_3}) = \hat{\sigma_5}$ then (*done*)
          $(S_4 \circ S_{acc}, B_5, t_4 : \hat{\sigma_5} : \varphi_4)$
        else (*loop*)
          let t_5 be such that $t_4\ R_f^{S_4\hat{\sigma_3} \sqsupseteq \hat{\sigma_5}}\ t_5$
          in $\mathcal{R}^{rec}(B_5, \hat{\sigma_5}, t_5, S_4 \circ S_{acc})$ end
      end
    $(S_1, B_1, t_1' : \hat{\sigma_1'} : \varphi_1') = \mathcal{R}^{rec}(B, \hat{\sigma_0}, \lambda x : \mu_x.t_1, \mathrm{Id})$
    t_2'' be such that $S_1t_2\ R_f^{S_1(\sigma_0) \sqsupseteq \forall\vec{\alpha}.\hat{\sigma_1'}}\ t_2''$
    $(S_2, B', t_2' \text{ as } (\_ : \mu' : \varphi_2')) = \mathcal{R}(B_1, S_1(TE) + \{f \mapsto (\forall\vec{\alpha}.\hat{\sigma_1'}, S_1\rho_0)\}, t_2'')$
  in
    $(S_2 \circ S_1, B', \texttt{letrec } f : S_2(\forall\vec{\alpha}.\hat{\sigma_1'}, S_1\rho_0) \texttt{ = } S_2t_1' \texttt{ in } t_2' \texttt{ end} : \mu' : S_2\varphi_1' \cup \varphi_2')$
  end

Fig. 13. Algorithm $\mathcal{R}$ (second half).

construct region-annotated types during fixed-point resolution; it suffices to compute S. This is particularly useful if substitutions are implemented by destructively updating term graphs. The conclusion $\varphi' \supseteq S(\varphi)$ says that the effect of an expression can only increase during fixed-point resolution. This is also useful in the implementation if φ is represented as an effect graph: fixed-point resolution can simply add edges to existing nodes in the graph. A very effective optimization is for $\mathcal{R}$ to compute only the increment $\varphi' \setminus S(\varphi)$ rather than the full effect φ'. This is done in the ML Kit implementation; details are omitted.

10. IMPLEMENTATION

We have implemented the algorithms in the ML Kit with Regions [Tofte et al. 1997]. The ML Kit with Regions compiles Standard ML to C or to HP PA-RISC assembly language.

The implementation requires care, since effect sets can grow large. For example, if a program consists of a long sequence of top-level function declarations culminating in a single function which, directly or indirectly, calls all the other functions, then the regions of those functions will appear free in the latent effect of the function.

Below we give a rough algorithmic analysis of the algorithms and data structures employed and report practical results obtained with the implementation.

We use effect graphs to represent effects, as outlined in Section 5.1. A node is labeled by `UNION`, a region variable, or an effect variable. `UNION` nodes give constant-time union of effects. The number of `UNION` nodes created is proportional to the size of the input expression.

Let n be the size of the input term to $\mathcal{S}$, counting both nodes in the abstract syntax tree and in the type annotations it contains. (Note that with ML type inference, putting types into terms may lead to an exponential growth in the term.) Let N be the number of region and effect variables generated by $\mathcal{S}$. By inspection of the $\mathcal{S}$ one sees that, since we require all type schemes to have structural quantification, we have that $N < k \times n$, where k is a program-independent constant. Furthermore, let T be the size of the largest type annotation occurring in the input.

Every node in the effect graph is an element of the union-find data structure, so that unification of two region variables or two arrow effects can be done using the union operation (not to be confused with the `UNION` node constructor). In what follows, we make the simplifying assumption that a union-find operation can be done in constant time. Then unification of two region variables can be done in constant time.

A cone is implemented as a stack of sets of nodes in the effect graph, indexed by nonnegative integer levels. The nodes at level i can only point to nodes at level at most i. A new set is pushed onto the stack each time one goes inside the scope of region and effect variables. More precisely, the new level corresponds to the bases called B_1 in Rules 18 and 19.

Every region variable and effect variable contains an updatable integer level. Unification of two region variables sets the level of the chosen representative to be the minimum of the levels of the two variables. Unification of two arrow effects $\epsilon_1.\varphi_1$ and $\epsilon_2.\varphi_2$ is similar, but if the level of ϵ_1 is greater than the level of ϵ_2, say, then levels of nodes reachable from ϵ_1 must be lowered to be at most the level of ϵ_2. Thus unification of two arrow effects can take at most time $O(N)$ and unification of two types can be done in time $O(N \times T)$.

Next, $\varphi' = \mathit{Observe}(B, \varphi)$ can be computed in time $O(N)$. But with the chosen representation of cones, *Observe* need not traverse the entire basis: it simply pops the topmost level of the cone and computes the intersection of these nodes and φ.

Instantiation of a type scheme $\sigma = \forall \alpha_1 \cdots \alpha_n \rho_1 \cdots \rho_k \epsilon_1 \cdots \epsilon_m.\tau$ is done in reverse depth-first order: if one has $\epsilon.\{\ldots, \epsilon', \ldots\} \in \mathit{arreff}(\tau)$ and both ϵ and ϵ' are among the bound variables of σ, then ϵ' is instantiated before ϵ. To make this possible, $\mathit{RegEffGen}(B, \tau)$ performs a linear-time contraction of strongly connected components in $\mathit{arreff}(\tau)$ before forming the type scheme. This has the additional benefit of sometimes reducing the number of secondary effect variables. Instantiation and generalization each take at most time $O(T \times N)$ each.

Matching two type schemes can be done in $O(T)$ time. Thinning one instantiation list can be done in $O(T)$ time. The number of occurrences of a program variable is at most n, so the operations in line 26 in $\mathcal{S}$ (and 31 in $\mathcal{R}$) take at most time $O(T \times n)$ each. In the Kit, every `letrec` expression contains a list of the (updatable) instantiation lists of all the free occurrences of the `letrec`-bound variable, so thinning is done without traversing the term.

Table I. Performance of Algorithms $\mathcal{S}$ and $\mathcal{R}$ in the ML Kit with Regions

Benchmark	**Lines**	**Nodes**	**Visited**	n	**Time,** $\mathcal{S}$	**Time,** $\mathcal{R}$
fib	6	24	35	30	0.01	0.01
loop2	42	98	142	158	0.01	0.03
dangle	16	96	286	182	0.05	0.08
dangle3	19	121	319	226	0.06	0.08
mandelbrot	170	623	956	1,031	0.14	0.13
tmergesort	169	658	956	1,077	0.28	0.24
reynolds2	155	762	1,185	1,369	0.25	0.30
reynolds3	156	833	1,288	1,495	0.31	0.31
life_eq	281	2,113	3,029	3,818	1.11	1.43
life	363	2,662	3,951	4,973	1.21	1.50
knuth-bendix-eq	781	3,972	11,752	7,431	3.68	9.15
knuth-bendix	837	5,433	9,533	10,478	4.16	10.35
simple	1,148	9,168	13,766	23,974	11.73	14.21

Collecting all of the above, we see that for each subexpression of the input term, the operations which $\mathcal{S}$ performs are in $O(n^2)$. Thus $\mathcal{S}$ is in $O(n^3)$.

Concerning $\mathcal{R}$, one might fear that the number of iterations could be exponential in the nesting depth of recursive functions, but that is not the case. By Theorem 9.2.2, every node in the abstract syntax tree is considered at most $O(N^2)$ times, for whenever a node is considered more than once, it is because a contraction different from the identity has taken place and this can at most happen $O(N^2)$ times. Thus $\mathcal{R}$ is $O(n^4)$ in the worst case.

Table I shows results obtained by running the Kit implementation of $\mathcal{S}$ and $\mathcal{R}$ on a range of benchmarks. For each benchmark we tabulate: **Lines**, the number of lines of SML source code, including the relevant parts of the Kit prelude which defines the initial basis; **Nodes**, the size of the explicitly typed input term to $\mathcal{S}$, counted as the number of subexpressions of the abstract syntax tree (not including type annotations); **Visited**, the number of times a subexpression of the abstract syntax tree was visited during the execution of $\mathcal{R}$ (note that $\mathcal{S}$ visits every node in the input tree precisely once); n the size of the input term, including type annotations; **Time**; the time taken by $\mathcal{S}$ and $\mathcal{R}$, respectively, in seconds, running on an HP 9000s700 computer. Times are wall clock time, measured using the SML/NJ timer.

The ratio between Visited and Nodes is an indication of the speed with which the fixed-point resolution converges. The Knuth-Bendix benchmarks have a deep nesting level of recursive functions, leading to more fixed-point iteration. For recursive functions that do not contain declarations of other recursive functions, analyzing the body just twice usually leads to a fixed point. The `simple` benchmark operates with large tuple types, as indicated by a high ratio between n and number of program lines. As one would expect, $\mathcal{S}$ is sensitive to programs with large types and $\mathcal{R}$ is sensitive to programs with deeply nested recursive function declarations. These factors seem to be more important than the number of lines in the source program.

11. SUMMARY

We have presented a region inference algorithm and proved that it is sound with respect to the Tofte-Talpin region inference rules. The key ideas in the algorithm are

(1) region inference is based on unification in consistent bases.

(2) in order to ensure termination, region inference is divided into two algorithms, separating the generation of region and effect variables from the iteration which deals with polymorphic recursion in regions and effects.

(3) in order to ensure that fixed-point resolution does not need to generate fresh region and effect variables, we impose a syntactic constraint on type schemes which prevents generalization of secondary region and effect variables.

(4) during fixed-point resolution, region and effect variables appear to migrate outward in the target term and the set of free region and effect variables under consideration appears to grow. Termination is proved by bounding the set of variables from above by a single basis, which is shown to contract during fixed-point iteration.

The algorithm has served as a specification for the ML Kit implementation of region inference for Standard ML. Runs of the ML Kit on a (small) benchmark suite suggest that, although our implementation gives somewhat slower compilation than ML users are accustomed to, the algorithm is fast enough for many practical purposes.

ONLINE-ONLY APPENDIX

A. FORMAL CORRECTNESS PROOFS

Appendix A is available only online. You should be able to get the online-only appendix from the citation page for this article:

http://www.acm.org/pubs/citations/journals/toplas/1998-20-4/p724-tofte/

Alternative instructions on how to obtain online-only appendices are given on the back inside cover of current issues of ACM TOPLAS or on the ACM TOPLAS web page:

http://www.acm.org/toplas

ACKNOWLEDGMENTS

We wish to thank Fritz Henglein, Mikkel Thorup, and Robert Paige for suggesting the use of graphs for representing effects. Also thanks to Alex Aiken for discussions concerning the upper bounds on the number of variables mentioned in Section 6.2. Finally, we wish to thank the three referees, whose perseverance and constructive suggestions we greatly appreciate.

REFERENCES

Aiken, A., Fähndrich, M., and Levien, R. 1995. Better static memory management: Improving region-based analysis of higher-order languages. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Languages and Implementation (PLDI)*. ACM Press, New York, 174–185.

Aponte, M. V. 1993. Extending record typing to type parametric modules with sharing. In *Proceedings of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL).* ACM Press, New York, 465–478.

Birkedal, L., Tofte, M., and Vejlstrup, M. 1996. 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.* ACM Press, New York, 171–183.

Damas, L. and Milner, R. 1982. Principal type schemes for functional programs. In *Proceedings of the 9th Annual ACM Symposium on Principles of Programming Languages.* ACM, New York, 207–212.

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

Hallenberg, N. 1997. ML Kit test report. (Automatically generated test report; available at http://www.diku.dk/research-groups/topps/activities/kit2/test.ps).

Harper, R., Milner, R., and Tofte, M. 1987. A type discipline for program modules. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT).* Lecture Notes in Computer Science, vol. 250. Springer-Verlag, Berlin, 308–319.

Henglein, F. 1993. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Sys. 15,* 2 (April), 253.

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

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

Leroy, X. 1992. Typage polymophe d'un language algorithmique. Ph.D. thesis, University Paris VII. English version: Polymorphic Typing of an Algorithmic Language, INRIA Research Report no. 1778, October 1992.

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

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

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

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

Naur, P. 1963. Revised report on the algorithmic language Algol 60. *Comm. ACM 1*, 1–17.

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

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

Rémy, D. 1989. Typechecking records and variants in a natural extension of ML. In *Proceedings 16th Annual ACM Symposium on Principles of Programming Languages.* ACM, 77–88.

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

Talpin, J.-P. and Jouvelot, P. 1992a. Polymorphic type, region and effect inference. *Journal of Functional Programming 2,* 3.

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

TOFTE, M. 1988. Operational semantics and polymorphic type inference. Ph.D. thesis, Edinburgh University, Department of Computer Science, Edinburgh University, Mayfield Rd., EH9 3JZ Edinburgh. Available as Technical Report CST-52-88.

TOFTE, M. AND BIRKEDAL, L. 1996. Unification and polymorphism in region inference. Submitted to the Milner Festschrift. `http://www.diku.dk/users/tofte/publ/milner.ps`.

TOFTE, M., BIRKEDAL, L., ELSMAN, M., , HALLENBERG, N., OLESEN, T. H., SESTOFT, P., AND BERTELSEN, P. 1997. Programming with regions in the ML Kit. Tech. Rep. DIKU-TR-97/12, Dept. of Computer Science, University of Copenhagen. (http://www.diku.dk/research-groups/topps/activities/kit2).

TOFTE, M. AND TALPIN, J.-P. 1992. Data region inference for polymorphic functional languages (technical summary). Tech. Rep. EMP/CRI/A-229, Ecole des Mines de Paris.

TOFTE, M. AND TALPIN, J.-P. 1994. 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.* ACM Press, 188–201.

TOFTE, M. AND TALPIN, J.-P. 1997. Region-based memory management. *Information and Computation 132,* 2, 109–176.

Received November 1996; revised August 1997; accepted December 1997

THIS DOCUMENT IS THE ONLINE-ONLY APPENDIX TO:

A Region Inference Algorithm

MADS TOFTE
University of Copenhagen
and
LARS BIRKEDAL
Carnegie Mellon University

0164-0925/98/0500-0000

In Appendix A we state and prove a substitution lemma. Then we proceed to the proof of the correctness of $\mathcal{S}$ (Appendix B) and the proof of correctness of $\mathcal{R}$ (Appendix C).

A. SUBSTITUTION LEMMAS

The following statement is false. If $\binom{A}{B}, TE \vdash e : \mu : \varphi$ and S is a contraction of A then $\binom{S(A)}{S(B)}, S(TE) \vdash S(e : \mu : \varphi)$. The reason is that applying a substitution to e may require renaming of region and effect variables that have binding occurrences in e, but such variables are members of A and we stated no condition to ensure that S acts as a renaming on these variables. The following lemma gives conditions under which pre-typing and well-typing are preserved under substitution.

LEMMA A.1. *If $\binom{A}{B}, TE \vdash_P t$ and $\binom{A}{B} \uplus \binom{A'}{B}$ exists, and S is a contraction of A', $S(A') \supseteq B' \supseteq S(B)$, and $\vdash B'$ then $\binom{S(A) \cup B'}{B'}, S(TE) \vdash_P S(t)$, and $\binom{S(A) \cup B'}{B'} \uplus \binom{S(A')}{B'}$ exists. Similarly for $\vdash_W$.*

In order to motivate the form of the lemma, let us consider a typical use of it. Consider the task of spreading an application $e = e_1\ e_2$. Let us assume that $\mathcal{S}$ has already spread e_1, yielding a pretyped t_1:

$$\begin{pmatrix} A \\ B \end{pmatrix}, TE \vdash_{\mathrm{P}} t_1. \tag{21}$$

When e_2 is subsequently spread, fresh region and effect variables are picked from a consistent basis A' satisfying that $\binom{A}{B} \uplus \binom{A'}{B}$ exists, i.e., that the fresh region and effect variables picked during the spreading of e_2 are disjoint from the variables used in the pretyping of e_1. Spreading e_2 also yields a contraction S of A', for spreading an expression involves unification much as in Milner's algorithms W [Milner 1978; Damas and Milner 1982]. Although S is a contraction of A' and $S(B) \subseteq S(A')$ we do not necessarily have that $S(B)$ is consistent: it need not be closed, due to region or effect variables which migrate into B during the spreading of e_2. Therefore, t_2 is pretyped in a perhaps larger consistent basis B', so that $S(A') \supseteq B' \supseteq S(B)$ and

$$\binom{S(A')}{B'}, S(TE) \vdash_{\mathrm{P}} t_2. \tag{22}$$

In order to combine (21) and (22) to obtain a pretyping for the application, we now wish to "apply" S to (21). The substitution lemma allows us to do this, yielding

$$\binom{S(A) \cup B'}{B'}, S(TE) \vdash_{\mathrm{P}} S(t_1) \tag{23}$$

$$\binom{S(A) \cup B'}{B'} \uplus \binom{S(A')}{B'} \qquad \text{exists.} \tag{24}$$

Using Rule 17 on (23), (22), and (24) one now constructs the desired pretyping

$$\binom{S(A) \cup S(A')}{B'}, S(TE) \vdash_{\mathrm{P}} S(t_1)\, t_2 : \mu : \varphi$$

provided the types of the operator and the operand match. (The actual construction in $\mathcal{S}$ unifies the types of t_1 and t_2, yielding yet another substitution, which has to be applied to the two premises, using the substitution lemma twice, before the desired pre-typing can be constructed.)

A.1 Definitions

The *yield* of a substitution S, written $\mathit{Yield}(S)$, is $\mathit{fv}(\mathit{Rng}(S^t \downarrow (\mathit{Supp}(S^t))) \cup \mathit{Rng}(S^r \downarrow (\mathit{Supp}(S^r))) \cup \mathit{Rng}(S^e \downarrow (\mathit{Supp}(S^e))))$. The set of variables *involved in* S, written $\mathit{Inv}(S)$, is defined to be $\mathit{Supp}(S) \cup \mathit{Yield}(S)$.

In what follows, 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 $(S' \bullet S^t, S' \bullet S^r, S' \bullet S^e)$. Every finite substitution can be extended to a (total) substitution on basic semantic objects.

Proof (of Lemma A.1). The conclusion "$\binom{S(A) \cup B'}{B'} \uplus \binom{S(A')}{B'}$ exists" follows directly independently of t. Since A' is consistent and S is a contraction of A', we have that $S(A')$ is consistent. Since also B' is assumed to be consistent and $B' \subseteq S(A')$ we have that $\binom{S(A')}{B'}$ is a cone. Since $\binom{A}{B} \uplus \binom{A'}{B}$ exists and S is a contraction of A' we have

$$\forall \rho \in Q \text{ of } (A \setminus B). S(\rho) = \rho$$

$$\forall \epsilon.\varphi \in \Phi \text{ of } (A \setminus B). S(\epsilon.\varphi) = \epsilon.S(\varphi).$$

Thus $S(A) \cup B'$ is consistent, $\binom{S(A) \cup B'}{B'}$ is therefore a cone, and $\binom{S(A) \cup B'}{B'} \uplus \binom{S(A')}{B'}$ exists.

The other conclusion, $\binom{S(A) \cup B'}{B'}, S(TE) \vdash_{\mathrm{P}} S(t)$, is shown by induction on the depth of inference of $\binom{A}{B}, TE \vdash_{\mathrm{P}} t$. We first prove the statement for $\vdash_{\mathrm{P}}$ and then for $\vdash_{\mathrm{W}}$. Write t in the form $e : \mu : \varphi$. There are the following cases.

Case $e \equiv f_{il}$ `at` ρ'. By Rule 15, there exist σ, S_0, τ, ρ, and τ_0 such that $B \vdash (TE, il, \mu)$, $TE(f) = (\sigma, \rho)$, $\mu = (\tau, \rho')$, $S_0 = zip(\sigma, il)$, $\tau_0 = body(\sigma)$, and

$$\mathrm{ML}(\tau) = \mathrm{ML}(S_0(\tau_0)). \tag{25}$$

Also, $A = B$ and $\varphi = \{\rho, \rho'\}$. By renaming the bound variables of σ, if necessary, we can achieve that $bv(\sigma) \cap Inv(S) = \emptyset$. Since $S(A') \supseteq B' \supseteq S(B)$ and B' is consistent we have $B' \vdash (S(TE), S(il), S(\mu))$. Furthermore, $(S(TE))(f) = (S(\sigma), S(\rho'))$, $S(\mu) = (S(\tau), S(\rho))$, and $S(\sigma) = \forall\, bv(\sigma).S(\tau_0)$. Also, $zip(S(\sigma), S(il))$ exists and equals $S \bullet S_0$. Finally,

$$\begin{array}{rcll} \mathrm{ML}(S(\tau)) & = & \mathrm{ML}(\tau) & \text{since } S \text{ is a contraction} \\ & = & \mathrm{ML}(S_0(\tau_0)) & \text{by (25)} \\ & = & \mathrm{ML}(S(S_0(\tau_0))) & \text{since } S \text{ is a contraction} \\ & = & \mathrm{ML}((S \bullet S_0)(S(\tau_0))) & bv(\sigma) \cap Inv(S) = \emptyset. \end{array}$$

Since $S(A) = S(B) \subseteq B'$ we have $\binom{S(A) \cup B'}{B'} = \binom{B'}{B'}$, so $\binom{S(A) \cup B'}{B'}, S(TE) \vdash_{\mathrm{P}} S(t)$ holds as desired.

Case $e \equiv x$. Straightforward.

Case $e \equiv \lambda x : \mu_x.t_1$. By Rule 16 there exist e_1, ϵ, φ_0, φ_1, μ_1, and ρ such that $\mu = (\mu_x \xrightarrow{\epsilon.\varphi_0} \mu_1, \rho)$, $\varphi = \{\rho\}$, $B \vdash TE$, $\varphi_0 \supseteq \varphi_1$, $(\{\rho\}, \{\epsilon.\varphi_0\}) \subseteq B$, and

$$\binom{A}{B}, TE + \{x \mapsto \mu_x\} \vdash_{\mathrm{P}} t_1 \qquad t_1 = e_1 : \mu_1 : \varphi_1. \tag{26}$$

Since $S(B) \subseteq B'$ and B' is consistent we have $B' \vdash S(TE)$. Next, by induction on (26) we have

$$\binom{S(A) \cup B'}{B'}, S(TE) + \{x \mapsto S(\mu_x)\} \vdash_{\mathrm{P}} S(t_1). \tag{27}$$

We have $S(\{\rho\}, \{\epsilon.\varphi_0\}) \subseteq S(B) \subseteq B'$, as required in Rule 16. Write $S(\epsilon.\varphi_0)$ in the form $\epsilon'.\varphi'$. By the definition of substitution (Section 5.2) we have $\varphi' \supseteq S(\varphi_0)$. Thus $\varphi' \supseteq S(\varphi_0) \supseteq S(\varphi_1)$. We have therefore satisfied the side-conditions of Rule 16, which we therefore apply to (27) to get the desired $\binom{S(A) \cup B'}{B'}, S(TE) \vdash_{\mathrm{P}} S(t)$.

Case $e \equiv t_1\, t_2$. Referring to Rule 17, since $\binom{A_3}{B} \uplus \binom{A'}{B}$ exists, each of $\binom{A_1}{B} \uplus \binom{A'}{B}$ and $\binom{A_2}{B} \uplus \binom{A'}{B}$ exist. Thus by induction on the two premises,

$$\binom{S(A_1) \cup B'}{B'}, S(TE) \vdash_{\mathrm{P}} S(t_1) \tag{28}$$

$$\binom{S(A_2) \cup B'}{B'}, S(TE) \vdash_{\mathrm{P}} S(t_2). \tag{29}$$

Since $\binom{A_3}{B} = \binom{A_1}{B} \uplus \binom{A_2}{B}$ exists, S is a contraction of A', $\binom{A_3}{B} \uplus \binom{A'}{B}$ exists, $S(A') \supseteq B' \supseteq S(B)$, and B' is consistent we have that

$$\binom{S(A_1) \cup B'}{B'} \uplus \binom{S(A_2) \cup B'}{B'} \qquad \text{exists.} \tag{30}$$

Let $\epsilon'.\varphi' = S(\epsilon_0)$. Then $S(\epsilon_0.\varphi_0) = \epsilon'.(\varphi' \cup S(\varphi_0))$, so $S(\varphi) = S(\varphi_1 \cup \varphi_2 \cup \{\epsilon_0, \rho_0\} \cup \varphi_0) = S(\varphi_1) \cup S(\varphi_2) \cup \{\epsilon', S(\rho_0)\} \cup (\varphi' \cup S(\varphi_0))$. By Rule 17 on this, (28), (29), and (30) we get the desired $\binom{S(A_3) \cup B'}{B'}, S(TE) \vdash_{\mathrm{P}} S(t)$. (Notice that the inclusion of ϵ_0 in the conclusion of Rule 17 is essential for this case to go through.)

Case $e \equiv$ `letrec` $f : (\sigma, \rho_1)(x)$ `=` t_1 `in` $e_2 : \mu : \varphi_2$ `end`. Here $\binom{A}{B}, TE \vdash_{\mathrm{P}} t$ was inferred by Rule 18 on premises $B \vdash (TE, \varphi_1)$, $\sigma = \forall \vec{\alpha}\, \vec{\rho}\, \vec{\epsilon}.\tau_0$, $\hat{\sigma} = \forall \vec{\rho}\, \vec{\epsilon}.\tau_0$, $btv(\sigma) \cap ftv(TE) = \emptyset$, $B_1 = bound(\sigma)$, $B \uplus B_1$ exists, $\tau_0 = \mu_x \xrightarrow{\epsilon_0.\varphi_0} \mu_1$, and

$$\binom{A_1}{B \uplus B_1}, TE + \{f \mapsto (\hat{\sigma}, \rho_1)\} \vdash \lambda\, x : \mu_x.t_1 : (\tau_0, \rho_1) : \varphi_1 \tag{31}$$

$$\binom{A_2}{B}, TE + \{f \mapsto (\sigma, \rho_1)\} \vdash e_2 : \mu : \varphi_2 \tag{32}$$

$$\binom{A}{B} = \binom{A_1}{B} \uplus \binom{A_2}{B}. \tag{33}$$

Since $\binom{A_1}{B \uplus B_1}$ is a cone and $\binom{A}{B} \uplus \binom{A'}{B}$ exists we have $Dom(B_1) \cap Dom(A') = \emptyset$. Since $B \vdash (TE, \varphi_1)$, $S(B) \subseteq B'$, and B' is consistent, we get

$$B' \vdash (S(TE), S(\varphi_1)). \tag{34}$$

Since $Dom(B_1) \cap Dom(A') = \emptyset$ and S is a contraction of A' we have

$$S(\sigma) = \forall \vec{\alpha}\, \vec{\rho}\, \vec{\epsilon}.S(\tau_0) \qquad S(\hat{\sigma}) = \forall \vec{\rho}\, \vec{\epsilon}.S(\tau_0) \tag{35}$$

$$btv(S(\sigma)) \cap ftv(S(TE)) = \emptyset \tag{36}$$

$$S(B_1) = bound(S\sigma). \tag{37}$$

Since furthermore $\binom{A}{B} \uplus \binom{A'}{B}$ exists and $B_1 \subseteq A_1 \setminus B$ we have that

$$S(B) \uplus S(B_1) \qquad \text{exists.} \tag{38}$$

Since $\binom{A}{B} \uplus \binom{A'}{B}$ exists and (33) and $\binom{A_1}{B \uplus B_1}$ is a cone we have that

$$\binom{A_1}{B \uplus B_1} \uplus \binom{A' \uplus B_1}{B \uplus B_1} \qquad \text{exists.} \tag{39}$$

Also, since B' is consistent and $S(A') \supseteq B' \supseteq S(B)$, $Inv(S) \cap Dom(B_1) = \emptyset$, and S is a contraction of A', we have that $B' \uplus S(B_1)$ exists and

$$S(A' \uplus B_1) \supseteq B' \uplus S(B_1) \supseteq S(B) \uplus S(B_1). \tag{40}$$

By induction on (31), (39), and (40) we get

$$\binom{S(A_1) \cup (B' \uplus S(B_1))}{B' \uplus S(B_1)}, S(TE) + \{f \mapsto S(\hat{\sigma}, \rho_1)\} \vdash_{\mathrm{P}} S(\lambda x : \mu_x.t_1 : (\tau_0, \rho_1) : \varphi_1). \tag{41}$$

Also, $\binom{A_2}{B} \uplus \binom{A'}{B}$ exists, so by induction on (32) we get

$$\binom{S(A_2) \cup B'}{B'}, S(TE) + \{f \mapsto S(\sigma, \rho_1)\} \vdash_{\mathrm{P}} S(e_2 : \mu : \varphi_2). \tag{42}$$

From (41), (42), and (33) and the assumption that S is a contraction of A' and $S(B_1) \subseteq S(A_1)$ we get

$$\binom{S(A_1) \cup (B' \uplus S(B_1))}{B'} \uplus \binom{S(A_2) \cup B'}{B'} \qquad \text{exists.} \tag{43}$$

Concerning the bases in (43) we have $S(A_1) \cup (B' \uplus S(B_1)) \cup S(A_2) \cup B' = S(A) \cup B'$. Thus from (34)–(38) and (41)–(43) we get the desired $\binom{S(A) \cup B'}{B'}, S(TE) \vdash_{\mathrm{P}} S(t)$.

Case $e \equiv \texttt{letregion}\ B_1\ \texttt{in}\ t_1\ \texttt{end}$. Here $\binom{A}{B}, TE \vdash_{\mathrm{P}} t$ has been inferred by Rule 19 from premises $B \uplus B_1$ exists, $B \vdash (TE, \mu)$, $\varphi = \mathit{Observe}(B, \varphi_1)$, and

$$\binom{A}{B \uplus B_1}, TE \vdash_{\mathrm{P}} t_1 \qquad t_1 = e_1 : \mu : \varphi_1. \tag{44}$$

Since $\binom{A}{B \uplus B_1}$ is a cone we have $B_1 \subseteq A$. Thus

$$\binom{A}{B \uplus B_1} \uplus \binom{A' \uplus B_1}{B \uplus B_1} \qquad \text{exists.} \tag{45}$$

Also, S is a contraction on $A' \uplus B_1$, and since $B_1 \subseteq A \setminus B$ we have $\mathit{Inv}(S) \cap \mathit{Dom}(B_1) = \emptyset$. Thus

$$\begin{aligned} S(A' \uplus B_1) &= S(A') \uplus S(B_1) \\ &\supseteq B' \uplus S(B_1) \\ &\supseteq S(B) \uplus S(B_1), \end{aligned}$$

and $B' \uplus S(B_1)$ is consistent. By induction on (44) and (45) we have

$$\binom{S(A) \cup (B' \uplus S(B_1))}{B' \uplus S(B_1)}, S(TE) \vdash_{\mathrm{P}} S(e_1 : \mu : \varphi_1).$$

Since $B_1 \subseteq A$ this reads

$$\binom{S(A) \cup B'}{B' \uplus S(B_1)}, S(TE) \vdash_{\mathrm{P}} S(e_1 : \mu : \varphi_1). \tag{46}$$

From $B \vdash (TE, \mu)$, $B' \supseteq S(B)$, and B' being consistent, we get $B' \vdash (S(TE), S(\mu))$ as required. By Lemma 7.3.1 on (44) we have $\varphi_1 \subseteq \mathit{Dom}(B \uplus B_1)$. Thus, since also $S(A') \supseteq B' \supseteq S(B)$, S is a contraction of A', and $\binom{A}{B} \uplus \binom{A'}{B}$ exists we have

$$\begin{aligned} \mathit{Observe}(B', S(\varphi_1)) &= \mathit{Observe}(S(B), S(\varphi_1)) \\ &\supseteq S(\mathit{Observe}(B, \varphi_1)) \qquad \text{by Lemma 7.3.2.} \end{aligned}$$

We also have $\mathit{Observe}(S(B), S(\varphi_1)) \subseteq S(\mathit{Observe}(B, \varphi_1))$, since the following: S is a contraction of A', $\binom{A}{B} \uplus \binom{A'}{B}$ exists, $\mathit{fv}(\varphi_1 \setminus \varphi) \subseteq \mathit{Dom}(B_1)$ and $\mathit{Inv}(S) \cap \mathit{Dom}(B_1) = \emptyset$. Thus by Rule 19 on (46) we get

$$\binom{S(A) \cup B'}{B'}, S(TE) \vdash_{\mathrm{P}} \texttt{letregion}\ S(B_1)\ \texttt{in}\ S(e_1 : \mu : \varphi_1)\ \texttt{end} : S(\mu) : S(\varphi),$$

i.e., since $\mathit{Inv}(S) \cap \mathit{Dom}(B_1) = \emptyset$,

$$\binom{S(A) \cup B'}{B'}, S(TE) \vdash_{\mathrm{P}} S(\texttt{letregion}\ B_1\ \texttt{in}\ e_1 : \mu : \varphi_1\ \texttt{end}) : S(\mu) : S(\varphi)$$

as desired.

This concludes the inductive proof of $\binom{S(A)\cup B'}{B'}, S(TE) \vdash_{\mathrm{P}} S(t)$.

Case for Well-Typing. The corresponding proof concerning $\vdash_{\mathrm{W}}$ is almost identical. The only difference (apart from using $\vdash_{\mathrm{W}}$ instead of $\vdash_{\mathrm{P}}$ everywhere) is in the case for Rule 20. Here one replaces (25) by

$$\tau = S_0(\tau_0) \tag{47}$$

and replaces the subsequent display of equations by

$$\begin{array}{rcll} S(\tau) & = & S(S_0(\tau_0)) & \text{by (47)} \\ & = & (S \bullet S_0)(S(\tau_0)) & bv(\sigma) \cap \mathit{Inv}(S) = \emptyset \end{array}$$

and the proof case is completed as before. □

As a corollary, we have the following lemma, in the case where S is the identity substitution.

LEMMA A.1.1. *If $\binom{A}{B}, TE \vdash_P t$, $\binom{B'}{B}$ is a cone, and $\binom{A}{B} \uplus \binom{B'}{B}$ exists, then $\binom{A\cup B'}{B'}, TE \vdash_P t$.*

Another important special case of Lemma A.1 is when A' equals B and B' equals $S(B)$:

LEMMA A.1.2. *If $\binom{A}{B}, TE \vdash_P t$, and S is a contraction of B then $\binom{S\,A}{S\,B}, S(TE) \vdash_P S(t)$, similarly for $\vdash_W$.*

A final remark on Lemma A.1: the requirement that S must be a contraction prevents S from having any type variables in its support. This reflects the fact that region inference is only performed on ML terms that are already correctly typed and region inference does not affect the underlying ML types, it merely refines them with region and effect information.

LEMMA A.1.3. *If ∇ is a proof of $\binom{A}{B}, TE \vdash_P t$ and ∇' is a proof of $\binom{A'}{B'}, TE' \vdash_P S(t)$ then ∇ and ∇' have the same depth, similarly for $\vdash_W$.*

PROOF. Straightforward inductive proof. Note that in the case $S = \mathrm{Id}$, the lemma says that all proofs that have t on the right-hand side of $\vdash_{\mathrm{P}}$ have the same depth; indeed this depth equals the depth of t. □

B. PROOF OF CORRECTNESS OF $\mathcal{S}$

A basis $A = (Q, \Phi)$ is *simple* if for all $\epsilon.\varphi \in \Phi$ one has $\varphi = \emptyset$. A cone $\binom{A'}{A}$ is *simple* if A and A' are both simple.

Here is the theorem that states the correctness of $\mathcal{S}$.

THEOREM B.1. *Assume $\underline{TE} \vdash \underline{e} : \underline{\tau}$, S_0 is a contraction of a simple consistent basis A, $\binom{S_0 A}{B}$ is a cone, $B \vdash TE$, and $\mathrm{ML}(TE) = \underline{TE}$. Then $(S, A', B', e : \mu : \varphi) = \mathcal{S}(A, B, TE, \underline{e})$ succeeds, $\mathrm{ML}(e) = \underline{e}$, $\mathrm{ML}(\mu) = \underline{\tau}$, $\binom{A'}{A}$ is a simple cone, $S(B) \subseteq B'$, $B' = Below(B', (S(B), \mu))$, and, letting $B'' = (A' \setminus A) \cup B$, B'' is consistent, S is a contraction of B'', and $\binom{SB''}{B'}, S(TE) \vdash_P e : \mu : \varphi$.*

Some explanation is in order. The expression $\underline{e}$ is a well-typed source expression which we want to "spread." The purpose of A is to record the set of region and effect variables that have been used up till the point just before $\underline{e}$ is spread. (Since A is simple, A is essentially just a set of region and effect variables.) Algorithm $\mathcal{S}$ performs unification. The substitution S_0 is the substitution computed up till the point just before $\underline{e}$ is spread; it is a contraction of A. B is a basis in which the region type environment TE is consistent; the assumption that $\binom{S_0 A}{B}$ is a cone implies that B is contained in the "current" contraction $S_0(A)$ of all the generated region and effect variables A. In the conclusion of the theorem, A' consists of A plus the variables generated during the spreading of $\underline{e}$. (Note that the statement "$\binom{A'}{A}$ is a cone" implies $A \subseteq A'$.) The statements "$S(B) \subseteq B'$ and $B' = \mathit{Below}(B', (S(B), \mu))$" express that B' contains the (the suitably contracted version of) B but, on the other hand, B' contains no more than is needed in order to prove that $S(B)$ and μ are consistent. The latter statement (about minimality of B') only holds because $\mathcal{S}$ calls *Retract* when necessary, thus encapsulating region and effect variables which occur free neither in $S(B)$ nor in μ inside `letregion` bindings. Finally, B'' is B extended with those region and effect variables which have been generated by the spreading of $\underline{e}$. Note that the theorem claims that S, the substitution returned by $\mathcal{S}$, is a contraction on B'', which is stronger than saying that it is a contraction on A': S does not involve (Section 5.2) variables in $\mathit{fv}(A) \setminus \mathit{fv}(B)$. (The variables in this set may be thought of as those variables which are not accessible via the type environment but have been generated during spreading of some other part of the program.)

Proof (of Theorem B.1). The proof is by structural induction on $\underline{e}$. In all the cases we assume the following:

$$\underline{TE} \vdash \underline{e} : \underline{\tau} \tag{48}$$

$$S_0 \text{ is a contraction of } A\text{, which is consistent} \tag{49}$$

$$\begin{pmatrix} S_0(A) \\ B \end{pmatrix} \quad \text{is a cone} \tag{50}$$

$$B \vdash TE \tag{51}$$

$$\underline{TE} = \mathrm{ML}(TE). \tag{52}$$

Case $\underline{e} \equiv x[\underline{\tau}_1, \ldots, \underline{\tau}_n]$. Assume that x is `letrec`-bound. (The case where x is otherwise bound is straightforward.) By (48) we have $x \in \mathit{Dom}(\underline{TE})$ and, letting $\underline{\sigma} = \underline{TE}(x)$, we have that $\underline{\sigma}$ must take the form $\forall \alpha_1 \cdots \alpha_m . \underline{\tau}_0$, with $m = n$ and $\underline{\tau}_0[\underline{\tau}_1/\alpha_1 \cdots \underline{\tau}_n/\alpha_n] = \underline{\tau}$. Since $\mathrm{ML}(TE) = \underline{TE}$ we therefore have $x \in \mathit{Dom}(TE)$ and, letting $(\sigma, \rho) = TE(x)$, we have $\mathrm{ML}(\sigma) = \underline{\sigma}$. In particular, the decomposition of σ at the beginning of *newInstance* succeeds. Just before the call of *inst* in *newInstance* we have $(A_1 \setminus A) \cup B \vdash \sigma$, $(A_1 \setminus A) \cup B \vdash \mathit{il}$, and σ and il have the same arity. Thus by Theorem 6.5.1 we have that the call to *inst* succeeds and that

$$S_1^e \text{ is a contraction of } (A_1 \setminus A) \cup B \tag{53}$$

$$S_1^e(\sigma) \geq \tau \; via \, S_1^e(il) \text{ and } S_1^e((A_1 \setminus A) \cup B) \vdash \tau. \tag{54}$$

Thus line 3 of $\mathcal{S}$ succeeds and

$$S_1^e(\sigma) \geq \tau \; via \, il_1 \; \text{ and } S_1^e((A_1 \setminus A) \cup B) \vdash \tau. \tag{55}$$

We see that $\mathcal{S}$ succeeds. We have to prove

$$\mathrm{ML}(x_{il_1} \, \texttt{at} \, \rho') = x[\underline{\tau}_1, \ldots, \underline{\tau}_n] \tag{56}$$

$$\mathrm{ML}(\tau, \rho') = \underline{\tau} \tag{57}$$

$$\begin{pmatrix} A_1 \uplus \{\rho'\} \\ A \end{pmatrix} \text{ is a simple cone} \tag{58}$$

$$S_1^e B \subseteq S_1^e(B \uplus (A_1 \setminus A)) \cup \{\rho'\} \tag{59}$$

$$B' = Below(B', (S_1^e(B), (\tau, \rho'))) \tag{60}$$

where $B' = S_1^e(B \uplus (A_1 \setminus A)) \uplus \{\rho'\}$. Moreover, letting $B'' = ((A_1 \uplus \{\rho'\}) \setminus A) \cup B$, we have to prove that

$$B'' \text{ is consistent} \tag{61}$$

$$S_1^e \text{ is a contraction of } B'' \tag{62}$$

$$\begin{pmatrix} S_1^e B'' \\ S_1^e(B \uplus (A_1 \setminus A)) \uplus \{\rho'\} \end{pmatrix}, S_1^e(TE) \vdash_{\mathrm{P}} x_{il_1} \, \texttt{at} \, \rho' : (\tau, \rho') : \{\rho, \rho'\}. \tag{63}$$

Now (56) follows from the definition of *newInstance* and the fact that S_1^e is an effect substitution. Next, (57) follows from (48), (52), (54), and the fact that S_1^e is an effect substitution. Next, (58) follows from the definition of *newInstance*; (59) is trivially true.

That (60) holds is due to the fact that σ is consistent in B (by 51), so that every quantified variable of σ occurs in the body of σ.

Next, (61) and (62) follow from (53). As for (63) we have

$$\begin{aligned} S_1^e B'' &= S_1^e(((A_1 \uplus \{\rho'\}) \setminus A) \cup B) \\ &= S_1^e(B \uplus (A_1 \setminus A)) \uplus \{\rho'\} \end{aligned}$$

as required. Also, we have $S_1^e B'' \vdash (S_1^e TE, il_1, (\tau, \rho'))$ by (51), (55), and $(A_1 \setminus A) \cup B \vdash il$. Let $S' = zip(S_1^e(\sigma), il_1)$. Now $body(S_1^e(\sigma)) = S_1^e(body(\sigma))$ and $\tau = S'(S_1^e(body(\sigma)))$, by (55), so in particular, $\mathrm{ML}(\tau) = \mathrm{ML}(S'(S_1^e(body(\sigma))))$. Thus Rule 15 gives (63).

Case $\underline{e} \equiv \lambda x : \underline{\tau}_0 . \underline{e}_1$. In this case, the assumption (48) is

$$\underline{TE} \vdash \lambda x : \underline{\tau}_0 . \underline{e}_1 : \underline{\tau}. \tag{64}$$

Thus there exists a $\underline{\tau}_1$ such that $\underline{\tau} = \underline{\tau}_0 \to \underline{\tau}_1$ and after line 6 in $\mathcal{S}$ we have

$$\underline{TE} + \{x \mapsto \underline{\tau}_0\} \vdash \underline{e}_1 : \underline{\tau}_1. \tag{65}$$

$$S_0 \text{ is a contraction of } A_0\text{, which is consistent and simple} \tag{66}$$

$$\begin{pmatrix} S_0 A_0 \\ B \cup (A_0 \setminus A) \end{pmatrix} \text{ is a cone} \tag{67}$$

$$B \cup (A_0 \setminus A) \vdash TE + \{x \mapsto \mu_0\} \tag{68}$$

$$\mathrm{ML}(TE + \{x \mapsto \mu_0\}) = \underline{TE} + \{x \mapsto \underline{\tau}_0\}. \tag{69}$$

By induction on (65)–(69) we have that line 7 of $\mathcal{S}$ succeeds and that

$$\mathrm{ML}(e_1) = \underline{e}_1 \tag{70}$$

$$\mathrm{ML}(\mu_1) = \underline{\tau}_1 \tag{71}$$

$$\begin{pmatrix} A_1 \\ A_0 \end{pmatrix} \text{ is a simple cone} \tag{72}$$

$$S_1(B \cup (A_0 \setminus A)) \subseteq B_1 \tag{73}$$

$$B_1 = \mathit{Below}(B_1, (S_1(B \cup (A_0 \setminus A)), \mu_1)). \tag{74}$$

Furthermore, letting $B_1'' = (A_1 \setminus A_0) \cup B \cup (A_0 \setminus A)$ we have

$$B_1'' \text{ is consistent} \tag{75}$$

$$S_1 \text{ is a contraction of } B_1'' \tag{76}$$

$$\begin{pmatrix} S_1 B_1'' \\ B_1 \end{pmatrix}, S_1(TE + \{x \mapsto \mu_0\}) \vdash_{\mathrm{P}} e_1 : \mu_1 : \varphi_1. \tag{77}$$

Consequently, $\mathcal{S}$ succeeds. Note that $B_1'' = (A_1 \setminus A) \cup B$. From (70) we get $\mathrm{ML}(\lambda x : S_1\mu_0.t_1) = \lambda x : \underline{\tau}_0.\underline{e}_1$. Also, from (76) and (71) we get $\mathrm{ML}(S_1\mu_0 \xrightarrow{\epsilon.\varphi_1} \mu_1, \rho) = \underline{\tau}_0 \to \underline{\tau}_1$, as required. From (72) we get that $\binom{A'}{A}$ is a simple cone. Also, by (73), $S(B) = S_1(B) \subseteq B_1 \subseteq B_1 \uplus A_2$, as required. Finally, let $B'' = A' \setminus A \cup B = B_1'' \uplus (\{\rho\}, \{\epsilon.\emptyset\})$. From (75) we get that B'' is consistent. By Lemma 7.3.1 on (77) we have $S_1 B_1'' \vdash \varphi_1$. Thus S is a contraction of B''. Next, from (77) and Lemma A.1.1 we get

$$\begin{pmatrix} S_1 B_1'' \uplus A_2 \\ B_1 \uplus A_2 \end{pmatrix}, S_1(TE + \{x \mapsto \mu_0\}) \vdash_{\mathrm{P}} e_1 : \mu_1 : \varphi_1; \tag{78}$$

i.e., since $S(\{\rho\}, \{\epsilon.\emptyset\}) = A_1$ and $S(TE) = S_1(TE)$,

$$\begin{pmatrix} SB'' \\ B' \end{pmatrix}, S(TE) + \{x \mapsto S_1\mu_0\} \vdash_{\mathrm{P}} e_1 : \mu_1 : \varphi_1. \tag{79}$$

Note that $B' \vdash S(TE)$, since $B \vdash TE$ and $S_1 B \subseteq B'$. Using Rule 16 on this and (79) we get

$$\begin{pmatrix} S(B'') \\ B' \end{pmatrix}, S(TE) \vdash_{\mathrm{P}} (\lambda x : S_1\mu_0.t_1) : (S_1\mu_0 \xrightarrow{\epsilon.\varphi_1} \mu_1, \rho) : \{\rho\}$$

as required. Finally, from (74) and $S(B) = S_1(B)$ we get the desired

$$B_1 \cup A_2 = \mathit{Below}(B_1 \cup A_2, (S(B), (S_1\mu_0 \xrightarrow{\epsilon.\varphi_1} \mu_1, \rho))).$$

Case $\underline{e} \equiv \underline{e}_1\,\underline{e}_2$. By (48) there exists a $\underline{\tau}_2$ such that

$$\underline{TE} \vdash \underline{e}_1 : \underline{\tau}_2 \rightarrow \underline{\tau} \tag{80}$$

$$\underline{TE} \vdash \underline{e}_2 : \underline{\tau}_2. \tag{81}$$

By induction on (80) and (49) through (52) we have that line 11 of $\mathcal{S}$ succeeds and that

$$\mathrm{ML}(e_1) = \underline{e}_1 \wedge \mathrm{ML}(\mu_2 \xrightarrow{\epsilon.\varphi_0} \mu_1, \rho_0) = \underline{\tau}_2 \rightarrow \underline{\tau} \tag{82}$$

$$\begin{pmatrix} A_1 \\ A \end{pmatrix} \text{ is a simple cone} \tag{83}$$

$$S_1(B) \subseteq B_1 \tag{84}$$

$$B_1'' \text{ is consistent} \tag{85}$$

$$S_1 \text{ is a contraction of } B_1'' \tag{86}$$

$$\begin{pmatrix} S_1 B_1'' \\ B_1 \end{pmatrix}, S_1(TE) \vdash_{\mathrm{P}} e_1 : (\mu_2 \xrightarrow{\epsilon.\varphi_0} \mu_1, \rho_0) : \varphi_1 \tag{87}$$

where $B_1'' = (A_1 \setminus A) \cup B$. By (49), (50), (83), (86), and the definition of B_1'' we have

$$S_1 \circ S_0 \text{ is a contraction of } A_1. \tag{88}$$

Next we wish to show

$$\begin{pmatrix} (S_1 \circ S_0) A_1 \\ B_1 \end{pmatrix} \text{ is a cone .} \tag{89}$$

But $\vdash B_1$, by (87), and $\vdash (S_1 \circ S_0)A_1$ by (88) and (83). Also $S_1 B_1'' \subseteq (S_1 \circ S_0)A_1$, so we have (89) from (87). Next, from (51) and (84) we get

$$B_1 \vdash S_1(TE) \tag{90}$$

and from (86) and (52),

$$\mathrm{ML}(S_1(TE)) = \underline{TE}. \tag{91}$$

By induction on (81) and (88)–(91) we have that line 12 of $\mathcal{S}$ succeeds and

$$\mathrm{ML}(e_2) = \underline{e}_2 \wedge \mathrm{ML}(\mu_2') = \underline{\tau}_2 \tag{92}$$

$$\begin{pmatrix} A_2 \\ A_1 \end{pmatrix} \text{ is a simple cone} \tag{93}$$

$$S_2(B_1) \subseteq B_2 \tag{94}$$

$$B_2'' \text{ is consistent} \tag{95}$$

$$S_2 \text{ is a contraction of } B_2'' \tag{96}$$

$$\begin{pmatrix} S_2 B_2'' \\ B_2 \end{pmatrix}, (S_2 \circ S_1)TE \vdash_{\mathrm{P}} e_2 : \mu_2' : \varphi_2 \tag{97}$$

where

$$B_2'' = (A_2 \setminus A_1) \cup B_1. \tag{98}$$

We now wish to apply the substitution lemma to (87). From (84), the definition of B_1'', (93), and (98) we get

$$\begin{pmatrix} S_1 B_1'' \\ B_1 \end{pmatrix} \uplus \begin{pmatrix} B_2'' \\ B_1 \end{pmatrix} \qquad \text{exists.} \tag{99}$$

Also, by (97) and (94),

$$S_2(B_1) \subseteq B_2 \subseteq S_2(B_2''). \tag{100}$$

Thus by Lemma A.1 on (87), (96), (99), and (100),

$$\begin{pmatrix} S_2(S_1 B_1'') \cup B_2 \\ B_2 \end{pmatrix}, S_2(S_1 TE) \vdash_{\mathrm{P}} S_2(e_1 : (\mu_2 \xrightarrow{\epsilon.\varphi_0} \mu_1, \rho_0) : \varphi_1) \tag{101}$$

$$\begin{pmatrix} S_2(S_1 B_1'') \cup B_2 \\ B_2 \end{pmatrix} \uplus \begin{pmatrix} S_2 B_2'' \\ B_2 \end{pmatrix} \qquad \text{exists.} \tag{102}$$

By (82) and (92) we have $\mathrm{ML}(S_2\mu_2) = \mathrm{ML}(\mu_2')$; by (97) and (101) we have $B_2 \vdash \mu_2'$ and $B_2 \vdash S_2\mu_2$. By Lemma 5.4.2 we thus get that line 13 of $\mathcal{S}$ succeeds and

$$S_3 \text{ is a most general unifier for } S_2(\mu_2) \text{ and } \mu_2' \text{ in } B_2 \tag{103}$$

$$S_3 \text{ is a contraction of } B_2 \tag{104}$$

$$S_3(B_2) = B_2 \Rightarrow S_3 = \mathrm{Id}. \tag{105}$$

As in $\mathcal{S}$, let $S = S_3 \circ S_2 \circ S_1$. By Lemma A.1.2 and (104) on (101) and (97) we get

$$\begin{pmatrix} S(B_1'') \cup S_3 B_2 \\ S_3 B_2 \end{pmatrix}, S(TE) \vdash_{\mathrm{P}} S_3(S_2(e_1 : (\mu_2 \xrightarrow{\epsilon.\varphi_0} \mu_1, \rho_0) : \varphi_1)) \tag{106}$$

$$\begin{pmatrix} S_3(S_2 B_2'') \\ S_3 B_2 \end{pmatrix}, S(TE) \vdash_{\mathrm{P}} S_3(e_2 : \mu_2' : \varphi_2). \tag{107}$$

From (102) and (104) we get

$$\begin{pmatrix} S(B_1'') \cup S_3 B_2 \\ S_3 B_2 \end{pmatrix} \uplus \begin{pmatrix} S_3(S_2(B_2'')) \\ S_3 B_2 \end{pmatrix} \qquad \text{exists.} \tag{108}$$

Let $B_3'' = (A_2 \setminus A) \cup B$, $e_3 = S_3(S_2(t_1)\, t_2)$, $\mu' = (S_3 \circ S_2)\mu_1$ and $\varphi' = S_3(S_2(\{\epsilon, \rho_0\} \cup \varphi_0 \cup \varphi_1) \cup \varphi_2)$. Since the two cones in (83) and (93) are simple, B_3'' is consistent. Since (86) and $S_1 B_1'' \supseteq B_1$ and $B_2'' = (A_2 \setminus A_1) \cup B_1$ and (96) we have that $S_2 \circ S_1$ is a contraction of B_3''. Also,

$$\begin{array}{rcll} B_2 & \subseteq & S_2(B_2'') & \text{by (100)} \\ & = & S_2(A_2 \setminus A_1 \cup B_1) & \text{by (98)} \\ & \subseteq & S_2(A_2 \setminus A_1 \cup S_1(A_1 \setminus A \cup B)) & \text{by (87)} \end{array}$$

$$= \quad (S_2 \circ S_1)(B_3'') \qquad \text{since } \mathit{Inv}(S_1) \cap \mathit{fv}(A_2 \setminus A_1) = \emptyset.$$

But then, by (104), S is a contraction of B_3''. Next, concerning the bases in (108) we get

$$\begin{array}{lll} S(B_1'') \cup S_3B_2 \cup S_3(S_2B_2'') & & \\ \quad = & SB_1'' \cup S_3(S_2B_2'') & \text{as } S_3B_2 \subseteq S_3(S_2B_2''), \text{ by (100)} \\ \quad = & SB_1'' \cup S_3(S_2(A_2 \setminus A_1)) \cup S_3(S_2(B_1)) & \text{by (98)} \\ \quad = & SB_1'' \cup S_3(S_2(A_2 \setminus A_1)) & \text{by (87)} \\ \quad = & S((A_1 \setminus A) \cup B) \cup S(A_2 \setminus A_1) & \text{since } \mathit{Inv}(S_1) \cap \mathit{fv}(A_2 \setminus A_1) = \emptyset \\ \quad = & S((A_2 \setminus A) \cup B). & \end{array}$$

Thus by Rule 17 on (106)–(108) and (103) we get

$$\begin{pmatrix} S((A_2 \setminus A) \cup B) \\ S_3B_2 \end{pmatrix}, S(\mathit{TE}) \vdash_{\mathrm{P}} e_3 : \mu' : \varphi'. \tag{109}$$

It follows from the definition of *Retract* that $\mathcal{S}$ succeeds. Moreover,

$$\begin{array}{rcll} \mathrm{ML}(e) & = & \mathrm{ML}(e_3) & \text{by the definition of ML} \\ & = & \mathrm{ML}(t_1)\ \mathrm{ML}(t_2) & \\ & = & e_1\, e_2. & \end{array}$$

Also, $\mathrm{ML}((S_3 \circ S_2)\mu_1) = \mathrm{ML}(\mu_1) = \underline{\tau}$, by (82). Since $\binom{A_1}{A}$ and $\binom{A_2}{A_1}$ are simple cones, $\binom{A_2}{A}$ is also a simple cone. Next, $S(B) \subseteq S_3(B_2)$, by (84) and (94), so

$$S(B) \subseteq \mathit{Below}(S_3B_2, (S(B), \mu')) = B' = \mathit{Below}(B', (S(B), \mu'))$$

as required. But then, by the definition of *Retract* and by (109) we have

$$\begin{pmatrix} S((A_2 \setminus A) \cup B) \\ B' \end{pmatrix}, S(\mathit{TE}) \vdash_{\mathrm{P}} \texttt{letregion}\ B_{drop}\ \texttt{in}\ e_3 : \mu' : \varphi'\ \texttt{end} : \mu' : \varphi$$

where $B_{drop} = S_3(B_2) \setminus B'$ and φ is the effect returned by $\mathcal{S}$.

Case $\underline{e} \equiv \texttt{letrec}\ f : \forall\alpha_1\cdots\alpha_n.\underline{\tau}_0(x)\ \texttt{=}\ \underline{e}_1\ \texttt{in}\ \underline{e}_2\ \texttt{end}$. Here (48) must have been inferred from premises

$$\underline{\mathit{TE}} + \{f \mapsto \underline{\tau}_0\} \vdash \lambda x : \underline{\tau}_x.\underline{e}_1 : \underline{\tau}_0 \tag{110}$$

$$\underline{\mathit{TE}} + \{f \mapsto \forall\alpha_1\cdots\alpha_n.\underline{\tau}_0\} \vdash \underline{e}_2 : \underline{\tau} \tag{111}$$

$$\underline{\tau}_0 = \underline{\tau}_x \to \underline{\tau}_1 \qquad \{\alpha_1, \ldots, \alpha_n\} \subseteq \mathit{ftv}(\underline{\tau}_0) \setminus \mathit{ftv}(\underline{\mathit{TE}}). \tag{112}$$

After line 22 of $\mathcal{S}$ we have

$$S_0 \text{ is a contraction of } A \uplus \{\rho\}, \text{ which is consistent and simple} \tag{113}$$

$$\begin{pmatrix} S_0(A \uplus \{\rho\}) \\ B \uplus \{\rho\} \end{pmatrix} \text{ is a cone} \tag{114}$$

$$B \uplus \{\rho\} \vdash \mathit{TE} + \{f \mapsto (\hat{\sigma}, \rho)\} \tag{115}$$

$$\mathrm{ML}(TE + \{f \mapsto (\hat{\sigma}, \rho)\}) = \underline{TE} + \{f \mapsto \underline{\tau}_0\}. \tag{116}$$

Thus by induction on (110) and (113)–(116) we have that line 23 of $\mathcal{S}$ succeeds and that

$$\mathrm{ML}(\lambda x : \mu_x.t_1') = \lambda x : \underline{\tau}_x.\underline{e}_1 \tag{117}$$

$$\mathrm{ML}(\tau_1, \rho_1) = \underline{\tau}_0 \tag{118}$$

$$\begin{pmatrix} A_1 \\ A \uplus \{\rho\} \end{pmatrix} \text{ is a simple cone} \tag{119}$$

$$S_1(B \uplus \{\rho\}) \subseteq B_1 \tag{120}$$

$$B_1 = \mathit{Below}(B_1, (S_1(B \uplus \{\rho\}), (\tau_1, \rho_1))). \tag{121}$$

Furthermore, letting

$$B_1'' = (A_1 \setminus (A \uplus \{\rho\})) \cup (B \uplus \{\rho\}) \tag{122}$$

we have

$$B_1'' \text{ is consistent} \tag{123}$$

$$S_1 \text{ is a contraction of } B_1'' \tag{124}$$

$$\begin{pmatrix} S_1 B_1'' \\ B_1 \end{pmatrix}, S_1(TE + \{f \mapsto (\hat{\sigma}, \rho)\}) \vdash_{\mathrm{P}} (\lambda x : \mu_x.t_1') : (\tau_1, \rho_1) : \varphi_1. \tag{125}$$

From (125) we get $\{S_1\rho, \rho_1\} \subseteq Q\,\mathit{of}\,B_1$. Thus line 24 of $\mathcal{S}$ succeeds and Lemma A.1.2 on (125) yields

$$\begin{pmatrix} S_2(S_1 B_1'') \\ S_2 B_1 \end{pmatrix}, S_2(S_1(TE + \{f \mapsto (\hat{\sigma}, \rho)\})) \vdash_{\mathrm{P}} S_2(\lambda x : \mu_x.t_1') : S_2(\tau_1, \rho_1) : S_2(\varphi_1). \tag{126}$$

Also, from (121) and $S_2(\rho_1) = S_2(S_1(\rho))$ we get

$$S_2 B_1 = \mathit{Below}(S_2 B_1, (S_2(S_1(B \uplus \{\rho\})), S_2\tau_1)). \tag{127}$$

We also have $S_2(B_1) \vdash S_2\tau_1$ and $S_2 B_1 \vdash S_2\varphi_1$, by Lemma 7.3.1 on (126). Also, $S_2(S_1(B \uplus \{\rho\})) \subseteq S_2(B_1)$, by (120). Thus by Lemma 6.6.1 on (127) we get that line 25 of $\mathcal{S}$ succeeds and

$$B_2 \vdash \sigma_1 \tag{128}$$

$$B_2 \vdash \varphi \tag{129}$$

$$S_2(B_1) \supseteq B_2 \supseteq S_2(S_1(B \uplus \{\rho\})). \tag{130}$$

Since $\mathrm{ML}(S_1\tau_1) = \mathrm{ML}(\tau_1) = \underline{\tau}_0 = \mathrm{ML}(\tau_0)$ we have $\hat{\sigma} \sqsupseteq \sigma_1$. From this and (126) we get that there exists a t_1'' such that $S_2 t_1' \; R^{\hat{\sigma} \sqsupseteq \sigma_1} \; t_1''$. Thus line 26 of $\mathcal{S}$ succeeds, and from Lemma 7.4.1 on (126) we have

$$\begin{pmatrix} S_2(S_1 B_1'') \\ S_2 B_1 \end{pmatrix}, S_2(S_1(TE)) + \{f \mapsto (\sigma_1, S_2\rho_1)\} \vdash_{\mathrm{P}} \lambda x : S_2(\mu_x).t_1'' : S_2(\tau_1, \rho_1) : S_2\varphi_1. \tag{131}$$

Moreover, from the definition of *RegEffGen* we have that $B_2 \uplus \mathit{bound}(\sigma_1)$ exists, is consistent, and

$$S_2(B_1) = B_2 \uplus \mathit{bound}(\sigma_1). \tag{132}$$

We now prepare to use induction a second time. From (49), (50), (122), and (124) we get

$$S_2 \circ S_1 \circ S_0 \text{ is a contraction of } A_1\text{, which is consistent and simple.} \tag{133}$$

Next, we have

$$S_2(S_1 B_1'') \subseteq (S_2 \circ S_1 \circ S_0)(A_1), \tag{134}$$

since

$$\begin{array}{rcll} S_2(S_1(B_1'')) & = & S_2(S_1(A_1 \setminus A \cup B)) & \text{by (122) and } \rho \in A_1 \setminus A \\ & \subseteq & S_2(S_1(A_1 \setminus A \cup S_0 A)) & \text{by (50)} \\ & = & (S_2 \circ S_1 \circ S_0)(A_1 \setminus A \cup A) & \text{by (49) and (119)} \\ & = & (S_2 \circ S_1 \circ S_0)(A_1). & \end{array}$$

Thus from (131), (134), and (130) we get

$$\begin{pmatrix} (S_2 \circ S_1 \circ S_0)A_1 \\ B_2 \end{pmatrix} \text{ is a cone .} \tag{135}$$

By (112) we have $\{\alpha_1, \ldots, \alpha_n\} \subseteq \mathit{ftv}(\sigma_1)$. Then, by Definition 6.4.1, (51), (130), and (128) we get

$$B_2 \vdash S_2(S_1 \mathit{TE}) + \{f \mapsto (\forall \alpha_1 \cdots \alpha_n . \sigma_1, S_2 \rho_1)\}. \tag{136}$$

Also, from (52), (118), and the fact that both S_1 and S_2 are contractions, we get

$$\mathrm{ML}(S_2(S_1(\mathit{TE})) + \{f \mapsto (\forall \alpha_1 \cdots \alpha_n . \sigma_1, S_2 \rho_1)\}) = \underline{\mathit{TE}} + \{f \mapsto \forall \alpha_1 \cdots \alpha_n . \underline{\tau}_0\}. \tag{137}$$

Thus by induction on (111), (133), and (135)–(137) we have that line 27 of $\mathcal{S}$ succeeds and that

$$\mathrm{ML}(e_2) = \underline{e}_2 \tag{138}$$

$$\mathrm{ML}(\mu) = \underline{\tau} \tag{139}$$

$$\begin{pmatrix} A_3 \\ A_1 \end{pmatrix} \text{ is a simple cone} \tag{140}$$

$$S_3(B_2) \subseteq B_3 \tag{141}$$

$$B_3 = \mathit{Below}(B_3, (S_3(B_2), \mu)). \tag{142}$$

Furthermore, letting

$$B_3'' = A_3 \setminus A_1 \cup B_2 \tag{143}$$

$$S = S_3 \circ S_2 \circ S_1 \tag{144}$$

we have

$$B_3'' \text{ is consistent} \tag{145}$$

$$S_3 \text{ is a contraction on } B_3'' \tag{146}$$

$$\begin{pmatrix} S_3 B_3'' \\ B_3 \end{pmatrix}, S(TE) + \{f \mapsto (S_3(\forall\alpha_1 \cdots \alpha_n.\sigma_1), S_3(S_2\rho_1))\} \vdash_{\mathrm{P}} e_2 : \mu : \varphi_2. \tag{147}$$

From (140), (143), (131), and (132) we get

$$\begin{pmatrix} S_2(S_1 B_1'') \\ B_2 \uplus \mathit{bound}\,\sigma_1 \end{pmatrix} \uplus \begin{pmatrix} B_3'' \uplus \mathit{bound}\,\sigma_1 \\ B_2 \uplus \mathit{bound}\,\sigma_1 \end{pmatrix} \quad \text{exists.} \tag{148}$$

Also, from (132), (141), (146), and (147) we get

$$B_3 \uplus S_3(\mathit{bound}\,\sigma_1) \text{ exists and is consistent} \tag{149}$$

$$S_3(B_2 \uplus \mathit{bound}\,\sigma_1) \subseteq B_3 \uplus S_3(\mathit{bound}\,\sigma_1) \subseteq S_3 B_3'' \uplus S_3(\mathit{bound}\,\sigma_1). \tag{150}$$

Moreover, from (143) we get that $B_3'' \uplus \mathit{bound}\,\sigma_1$ exists and is consistent. From (146) we get

$$S_3 \text{ is a contraction on } B_3'' \uplus \mathit{bound}\,\sigma_1 \tag{151}$$

$$S_3(B_3'' \uplus \mathit{bound}\,\sigma_1) = S_3(B_3'') \uplus S_3(\mathit{bound}\,\sigma_1).$$

By Lemma A.1 on (131), (132), (148), (151), (149), and (150) we have

$$\begin{pmatrix} SB_1'' \cup B_3^+ \\ B_3^+ \end{pmatrix}, S(TE) + \{f \mapsto (S_3\sigma_1, S_3(S_2\rho_1))\} \vdash_{\mathrm{P}} \begin{array}{r} \lambda x : S_3(S_2\mu_x).S_3 t_1'' : \\ S_3(S_2(\tau_1, \rho_1)) : S_3(S_2\varphi_1) \end{array} \tag{152}$$

$$\begin{pmatrix} SB_1'' \cup B_3^+ \\ B_3^+ \end{pmatrix} \uplus \begin{pmatrix} S_3 B_3'' \uplus S_3(\mathit{bound}\,\sigma_1) \\ B_3^+ \end{pmatrix} \quad \text{exists} \tag{153}$$

where $B_3^+ = B_3 \uplus S_3(\mathit{bound}(\sigma_1))$. But then, since B_3 is consistent and the disjoint sum $B_3 \uplus S_3(\mathit{bound}\,\sigma_1)$ exists, we have that

$$\begin{pmatrix} SB_1'' \cup B_3^+ \\ B_3 \end{pmatrix} \uplus \begin{pmatrix} S_3 B_3'' \\ B_3 \end{pmatrix} \quad \text{exists.} \tag{154}$$

Now $\mathit{bound}(S_3\sigma_1) = S_3(\mathit{bound}\,\sigma_1)$, by (146), (143), and (132). Also, we have

$$B_3 \vdash (S(TE), S_3(S_2\varphi_1)) \tag{155}$$

by (51), (130), (141), and the fact that, by (125), $\varphi_1 = \{\rho_1\}$, so $S_3(S_2\varphi_1) = \{S_3(S_2\rho_1)\} = \{S(\rho)\}$. By Rule 18 on (155), (152), (147), and (154), we have

$$\begin{pmatrix} B_4 \\ B_3 \end{pmatrix}, S(TE) \vdash_{\mathrm{P}} \begin{array}{l} \texttt{letrec}\ f : (\forall\alpha_1 \cdots \alpha_n.S_3\sigma_1, S_3(S_2\rho_1))\texttt{(}x\texttt{)}\ \texttt{=}\ S_3 t_1'' \\ \texttt{in}\ e_2 : \mu : \varphi_2\ \texttt{end} : \mu : S_3(S_2\varphi_1) \cup \varphi_2 \end{array} \tag{156}$$

where

$$B_4 \quad = \quad S(B_1'') \cup B_3^+ \cup S_3(B_3'')$$

$$
\begin{array}{lll}
= & S(B_1'') \cup S_3(\mathit{bound}\,\sigma_1) \cup S_3(B_3'') & \text{by (147)} \\
= & S(B_1'') \cup S_3(B_3'') & \text{by (132) and (125)} \\
= & S(B_1'') \cup S_3(A_3 \setminus A_1 \cup B_2) & \text{by (143)} \\
= & S(B_1'') \cup S_3(A_3 \setminus A_1) & \text{by (126) and (130)} \\
= & S(A_1 \setminus A \cup B) \cup S_3(A_3 \setminus A_1) & \text{by (122) and } \rho \in A_1 \setminus A \\
= & S(A_1 \setminus A \cup B) \cup S(A_3 \setminus A_1) & \\
= & S(A_3 \setminus A \cup B). &
\end{array}
$$

It follows that $\mathcal{S}$ succeeds and $\mathrm{ML}(e) = \underline{e}$ and $\mathrm{ML}(\mu) = \underline{\tau}$. Moreover, $\binom{A'}{A} = \binom{A_3}{A}$ is a simple cone and, by the definition of *Retract*, $S(B) \subseteq B' = \mathit{Below}(B_3, (S(B), \mu))$. Thus also $B' = \mathit{Below}(B', (S(B), \mu))$. Furthermore, letting $B'' = (A' \setminus A) \cup B$ (i.e., $B'' = (A_3 \setminus A) \cup B$), we have that B'' is consistent. Also, S is a contraction of B'', by (122), (124), (126), (130), (143), (146), and the fact that S_2 is a contraction of $S_1(B_1'')$. Finally, Rule 19 on (156) gives the desired $\binom{S(B'')}{B'}, S(TE) \vdash_{\mathrm{P}} e : \mu : \varphi$. □

C. PROOF OF CORRECTNESS OF $\mathcal{R}$

THEOREM CORRECTNESS OF $\mathcal{R}$. *Let $t = e : \mu : \varphi$ be saturated. If $\binom{A}{B}, TE \vdash_{\mathrm{P}} t$ then $(S, B', t'$ as $(e' : \mu' : \varphi')) = \mathcal{R}(B, TE, t)$ succeeds and S is a contraction of A, $\binom{S(A)}{B'}, S(TE) \vdash_{W} t'$, $\mathrm{ML}(t') = \mathrm{ML}(t)$, $B' \supseteq S(B)$, $\mu' = S(\mu)$, and $\varphi' \supseteq S(\varphi)$. Furthermore, if $S(A) = A$, then $S = \mathrm{Id}$, $t' = t$, and $B' = B$. Also, if t is strongly saturated then $B' = \mathit{Below}(S(A), S(B))$.*

PROOF. The proof is by induction on the depth of the proof of $\binom{A}{B}, TE \vdash_{\mathrm{P}} t$, with an inner induction on the size of A, where the size of a basis A is defined as the largest number n for which there exists a strictly decreasing sequence of n bases starting from A:

$$A = A_1 > A_2 \ldots > A_n.$$

(This is well-defined by Lemma 5.3.1.)

Case $e =$ `letrec` $f : (\sigma_0, \rho_0)$`(`x`) =` t_1 `in` t_2 `end`. Since t is assumed saturated, t_1 and t_2 are strongly saturated. Assume $\binom{A}{B}, TE \vdash_{\mathrm{P}} t$. Then lines 25 and 26 of $\mathcal{R}$ succeed. Since $\binom{A}{B}, TE \vdash_{\mathrm{P}} t$ must have been inferred by Rule 18, there exist B_{11}, A_{11}, A_{12}, φ_{11}, and φ_{12} such that

$$B_{11} = \mathit{bound}(\hat{\sigma_0}) \tag{157}$$

$$B \vdash (TE, \varphi_{11}) \tag{158}$$

$$\mathit{ftv}(\vec{\alpha}) \cap \mathit{ftv}(TE) = \emptyset \tag{159}$$

$$B \uplus B_{11} \text{ exists} \tag{160}$$

$$\binom{A_{11}}{B \uplus B_{11}}, TE + \{f \mapsto (\hat{\sigma_0}, \rho_0)\} \vdash_{\mathrm{P}} \lambda x : \mu_x . t_1 : (\tau_0, \rho_0) : \varphi_{11} \tag{161}$$

$$\begin{pmatrix} A_{12} \\ B \end{pmatrix}, TE + \{f \mapsto (\sigma_0, \rho_0)\} \vdash_{\mathrm{P}} e_2 : \mu : \varphi_{12} \tag{162}$$

$$t_2 = e_2 : \mu : \varphi_{12} \land \varphi = \varphi_{11} \cup \varphi_{12} \tag{163}$$

$$\begin{pmatrix} A \\ B \end{pmatrix} = \begin{pmatrix} A_{11} \\ B \end{pmatrix} \uplus \begin{pmatrix} A_{12} \\ B \end{pmatrix}. \tag{164}$$

The next thing that happens in $\mathcal{R}$ is that $\mathcal{R}^{rec}$ is called. We now prove the following lemma concerning $\mathcal{R}^{rec}$.

Lemma C.1. *Let B_3, $\hat{\sigma_3}$, t_3, and S_{acc} be given. Let $B_{13} = bound\,\hat{\sigma_3}$. IF*

$$t_3 \textit{ is strongly saturated and } \mathrm{ML}(t_3) = \mathrm{ML}(\lambda x : \mu_x.t_1) \tag{165}$$

$$t_3 = e_3 : (body(\hat{\sigma_3}), S_{acc}(\rho_0)) : \varphi_3, \textit{ for some } e_3 \textit{ and } \varphi_3 \tag{166}$$

$$S_{acc} \textit{ is a contraction of } A_{11} \tag{167}$$

$$S_{acc}A_{11} \supseteq B_3 \supseteq S_{acc}B \tag{168}$$

$$B_3 \uplus B_{13} \textit{ exists} \tag{169}$$

$$\begin{pmatrix} S_{acc}A_{11} \\ B_3 \uplus B_{13} \end{pmatrix}, S_{acc}TE + \{f \mapsto (\hat{\sigma_3}, S_{acc}(\rho_0))\} \vdash_P t_3 \tag{170}$$

$$B_3 \vdash S_{acc}\hat{\sigma_0} \sqsupseteq \hat{\sigma_3} \tag{171}$$

THEN $(S_1, B_1, t_1' : \hat{\sigma_1'} : \varphi_1') = \mathcal{R}^{rec}(B_3, \hat{\sigma_3}, t_3, S_{acc})$ succeeds and, letting $B_1' = bound(\hat{\sigma_1'})$,

$$t_1' \textit{ is strongly saturated and } \mathrm{ML}(t_1') = \mathrm{ML}(\lambda x : \mu_x.t_1) \tag{172}$$

$$S_1 \textit{ is a contraction of } A_{11} \tag{173}$$

$$S_1A_{11} \supseteq B_1 \supseteq S_1B \tag{174}$$

$$B_1 \uplus B_1' \textit{ exists} \tag{175}$$

$$\begin{pmatrix} S_1A_{11} \\ B_1 \uplus B_1' \end{pmatrix}, S_1(TE) + \{f \mapsto (\hat{\sigma_1'}, S_1\rho_0)\} \vdash_W t_1' \tag{176}$$

$$B_1 \vdash S_1\hat{\sigma_0} \sqsupseteq \hat{\sigma_1'} \tag{177}$$

$$t_1' = e_1' : (body(\hat{\sigma_1'}), S_1\rho_0) : \varphi_1', \textit{ for some } e_1' \textit{ and } \varphi_1' \tag{178}$$

$$S_1(A_{11}) = S_{acc}(A_{11}) \Rightarrow (S_1 = S_{acc} \land t_1' = t_3 \land B_1 = B_3). \tag{179}$$

We prove this lemma by an inner induction on the size of $S_{acc}A_{11}$. By (165) we have $\mathrm{ML}(\lambda x : \mu_x.t_1) = \mathrm{ML}(t_3)$ and both $\lambda x : \mu_x.t_1$ and t_3 are strongly saturated. Thus the proofs of (170) and (161) have the same depth, which is smaller than the depth of the proof of $\binom{A}{B}, TE \vdash_{\mathrm{P}} t$. Thus we can use the outer induction on (170) to conclude that line 27 of $\mathcal{R}$ succeeds and that

$$S_4 \text{ is a contraction of } S_{acc}A_{11} \tag{180}$$

$$\begin{pmatrix} S_4(S_{acc}A_{11}) \\ B_4 \end{pmatrix}, S_4(S_{acc}TE) + \{f \mapsto S_4(\hat{\sigma_3}, S_{acc}\rho_0)\} \vdash_{\mathrm{W}} t_4 \tag{181}$$

$$B_4 \supseteq S_4(B_3 \uplus B_{13}) \tag{182}$$

$$S_4(S_{acc}A_{11}) = S_{acc}A_{11} \Rightarrow (S_4 = \mathrm{Id} \wedge t_4 = t_3 \wedge B_4 = B_3 \uplus B_{13}) \tag{183}$$

$$(\tau_4, \rho_4) = S_4\mu_3 \wedge \varphi_4 \supseteq S_4\varphi_3 \tag{184}$$

$$t_4 \text{ is strongly saturated and } \mathrm{ML}(t_4) = \mathrm{ML}(t_3) \tag{185}$$

$$B_4 = \mathit{Below}(S_4(S_{acc}A_{11}), S_4(B_3 \uplus B_{13})) \tag{186}$$

where $(\_, \mu_3, \varphi_3) = t_3$ and $(\_, (\tau_4, \rho_4), \varphi_4) = t_4$. (In (185) we have used Lemma 9.2.1.) By (181) we have $B_4 \vdash \tau_4$ and $B_4 \vdash \varphi_4$; by (182) we have $B_4 \supseteq S_4B_3$. Also, by the definition of B_{13} and the fact that ρ *of* $\mu_3 = S_{acc}\rho_0 \in Q$ *of* $(S_{acc}B) \subseteq Q$ *of* B_3 we have $\mathit{Below}(B, (S_4B_3, S_4\tau_3)) = \mathit{Below}(B, S_4(B_3 \uplus B_{13}, \mu_3))$, for all bases B. Thus by (186) and (184) we have

$$B_4 = \mathit{Below}(B_4, (S_4B_3, \tau_4)). \tag{187}$$

By Lemma 6.6.1 on $B_4 \vdash t_4$, $B_4 \vdash \varphi_4$, $S_4B_3 \subseteq B_4$, and (187) we have that line 28 of $\mathcal{R}$ succeeds and that

$$B_4 \supseteq B_5 \supseteq S_4B_3 \tag{188}$$

$$B_5 \vdash \hat{\sigma_5} \wedge B_5 \vdash \varphi_4. \tag{189}$$

From (171) and $B_5 \supseteq S_4B_3$ we get

$$B_5 \vdash S_4(S_{acc}\hat{\sigma_0}) \sqsupseteq S_4(\hat{\sigma_3}). \tag{190}$$

Moreover, since $\tau_4 = S_4(\tau_3)$ and $bv(\hat{\sigma_5}) \cap fv(S_4\hat{\sigma_3}) \subseteq bv(\hat{\sigma_5}) \cap fv(S_4\, B_3) = \emptyset$ we have

$$B_5 \vdash S_4(\hat{\sigma_3}) \sqsupseteq \hat{\sigma_5}. \tag{191}$$

Here we have used that S_4 is the identity on all type variables, by (180).

We now consider the two possible outcomes of the comparison in line 29 of $\mathcal{R}^{rec}$.

Case $S_4\hat{\sigma_3} = \hat{\sigma_5}$. Then $\mathcal{R}^{rec}$ returns with result

$$(S_1, B_1, t_1' : \hat{\sigma_1'} : \varphi_1') = (S_4 \circ S_{acc}, B_5, t_4 : \hat{\sigma_5} : \varphi_4).$$

We have to prove (172)–(179). Let $B_1' = \mathit{bound}\,\hat{\sigma_1'} = \mathit{bound}\,\hat{\sigma_5}$. Now (172) follows from (185) and (165). Next, (173) follows from (167) and (180). Moreover, by (181), (182), and (168) we have $S_1A_{11} = S_4(S_{acc}A_{11}) \supseteq B_4 \supseteq B_5 = B_1 \supseteq S_4\, B_3 \supseteq S_1\, B$ showing (174). By construction of $\hat{\sigma_5}$ we have (175), indeed $B_1 \uplus B_1' = B_5 \uplus$

bound $\hat{\sigma_5} = B_4$, by (187). Thus (176) follows from (181) and $S_4(\hat{\sigma_3}) = \hat{\sigma_5} = \hat{\sigma'_1}$. Furthermore, (177) follows from (190) and (191). Next, (178) follows from (181). Finally, assume $(S_4 \circ S_{acc})A_{11} = S_{acc}A_{11}$. Then by (183) we have $S_4 = \mathrm{Id}$, $t_4 = t_3$, and $B_4 = B_3 \uplus B_{13}$. But then we have $S_1 = S_4 \circ S_{acc} = S_{acc}$, as desired; $t'_1 = t_4 = t_3$, as desired; and $B_1 = B_5 = B_3$, as desired.

Case $S_4\hat{\sigma_3} \neq \hat{\sigma_5}$. Then by (191) we have

$$B_5 \vdash S_4(\hat{\sigma_3}) \sqsupset \hat{\sigma_5}. \tag{192}$$

There exists a t_5 such that $t_4 \ R_f^{S_4\hat{\sigma_3} \sqsupset \hat{\sigma_5}} \ t_5$, so line 31 of $\mathcal{R}^{rec}$ succeeds. We now want to use induction on the recursive call in line 32, so we need to establish that the conditions for applying induction (see 165–171) are met. From (185) and $t_4 \ R_f^{S_4\hat{\sigma_3} \sqsupset \hat{\sigma_5}} \ t_5$ we get

$$t_5 \text{ is strongly saturated and } \mathrm{ML}(t_5) = \mathrm{ML}(\lambda x : \mu_x . t_1). \tag{193}$$

As in the previous case we have

$$S_4 \circ S_{acc} \text{ is a contraction of } A_{11} \tag{194}$$

$$(S_4 \circ S_{acc})A_{11} \supseteq B_5 \supseteq (S_4 \circ S_{acc})B \tag{195}$$

$$t_4 = e_4 : (\tau_4, (S_4 \circ S_{acc})\rho_0) : \varphi_4 \tag{196}$$

for some e_4 and $\tau_4 = \mathit{body}(\hat{\sigma_5})$. From (196) we get

$$t_5 = e_5 : (\tau_4, (S_4 \circ S_{acc})\rho_0) : \varphi_4 \tag{197}$$

for some e_5 and φ_4. Also, as in the previous case we have

$$B_5 \uplus B'_1 \text{ exists and equals } B_4 \tag{198}$$

where $B'_1 = \mathit{bound}\,\hat{\sigma_5}$. From Lemma 7.4.1 on (181) and (191) we then get

$$\begin{pmatrix} (S_4 \circ S_{acc})A_{11} \\ B_5 \uplus B'_1 \end{pmatrix}, (S_4 \circ S_{acc})TE + \{f \mapsto (\hat{\sigma_5}, (S_4 \circ S_{acc})\rho_0)\} \vdash_{\mathrm{P}} t_5. \tag{199}$$

Also, we must have $(S_4 \circ S_{acc})A_{11} < S_{acc}A_{11}$. For otherwise, by (183), we would have

$$S_4 = \mathrm{Id} \wedge t_4 = t_3 \wedge B_4 = B_3 \uplus B_{13}.$$

But then we would have $S_4(\hat{\sigma_3}) = \hat{\sigma_3} = \hat{\sigma_5}$, a contradiction with our assumption $S_4\hat{\sigma_3} \neq \hat{\sigma_5}$. (This is where the clause

"Furthermore, if $S(A) = A$, then $S = \mathrm{Id}$, $t' = t$, and $B' = B$"

of Theorem 9.2.2 is important.) Thus by the inner induction on (193), (196), (194), (195), (198), (199), (190), and (191), the tail call in line 32 of $\mathcal{R}^{rec}$ succeeds and, still letting $B'_1 = \mathit{bound}(\hat{\sigma_1})$, we have (173)–(178) and also

$$t'_1 \text{ is strongly saturated and } \mathrm{ML}(t'_1) = \mathrm{ML}(t_5). \tag{200}$$

But then $\mathcal{R}^{rec}(B_3, \hat{\sigma_3}, t_3, S_{acc})$ returns with the same result. Thus (173)–(178) still hold, whereas (172) follows from (200) and (193). As for (179), we have $S_1A_{11} \neq$

$S_{acc}A_{11}$, since S_1 is a contraction and $S_4(S_{acc}A_{11}) < S_{acc}A_{11}$, so (179) is trivially satisfied. This concludes the proof of Lemma C.1.

Now let us return to line 33 of $\mathcal{R}$. The conditions for applying Lemma C.1 are all met. Thus the call in line 33 of $\mathcal{R}$ succeeds and (172)–(178) hold plus

$$S_1 A_{11} = A_{11} \Rightarrow (S_1 = \mathrm{Id} \wedge t_1' = \lambda x : \mu_x.t_1 \wedge B_1 = B). \tag{201}$$

From (177) we get $B_1 \vdash S_1(\sigma_0) \sqsupseteq \forall\vec{\alpha}.\hat{\sigma_1'}$. Thus there exists a t_2'' as in line 34 of $\mathcal{R}^{rec}$. Since t_2 is strongly saturated, so is t_2''. From Lemma A.1 on (162), (164), (173), and (174) we get

$$\begin{pmatrix} S_1 A_{12} \cup B_1 \\ B_1 \end{pmatrix}, S_1 TE + \{f \mapsto (S_1\sigma_0, S_1(\rho_0))\} \vdash_{\mathrm{P}} S_1 t_2 \tag{202}$$

$$\begin{pmatrix} S_1 A_{11} \\ B_1 \end{pmatrix} \uplus \begin{pmatrix} S_1 A_{12} \cup B_1 \\ B_1 \end{pmatrix} \qquad \text{exists.} \tag{203}$$

By Lemma A.1.3 the depth of the proof of (202) equals the depth of the proof of (162). By Lemma 7.4.1 on (202) we get

$$\begin{pmatrix} S_1 A_{12} \cup B_1 \\ B_1 \end{pmatrix}, S_1\, TE + \{f \mapsto (\forall\vec{\alpha}.\hat{\sigma_1'}, S_1(\rho_0))\} \vdash_{\mathrm{P}} t_2'' \tag{204}$$

again by a proof of depth at most the depth of the proof of (162). Thus by induction we get that line 35 of $\mathcal{R}$ succeeds and that

$$S_2 \text{ is a contraction of } S_1 A_{12} \cup B_1 \tag{205}$$

$$\begin{pmatrix} S_2(S_1 A_{12} \cup B_1) \\ B' \end{pmatrix}, S_2(S_1\, TE) + \{f \mapsto S_2(\forall\vec{\alpha}.\sigma_1', S_1(\rho_0))\} \vdash_{\mathrm{W}} t_2' \tag{206}$$

$$\mathrm{ML}(t_2') = \mathrm{ML}(t_2'') \tag{207}$$

$$B' \supseteq S_2 B_1 \tag{208}$$

$$S_2(S_1 A_{12} \cup B_1) = S_1 A_{12} \cup B_1 \Rightarrow (S_2 = \mathrm{Id} \wedge t_2' = t_2'' \wedge B' = B_1) \tag{209}$$

$$\mu' = S_2(S_1\mu) \wedge \varphi_2' \supseteq S_2(S_1\varphi_{12}) \tag{210}$$

$$B' = \mathit{Below}(S_2(S_1 A_{12} \cup B_1), S_2 B_1). \tag{211}$$

But then $\mathcal{R}(B, TE, t)$ succeeds with results $S = S_2 \circ S_1$, $B' = B'$, $\mu' = \mu'$, $\varphi' = S_2(\varphi_1') \cup \varphi_2'$, and $t' = (e' : \mu' : \varphi')$, where

$$e' = \texttt{letrec}\ f : (\forall\vec{\alpha}.S_2\hat{\sigma_1'}, S_2(S_1\,\rho_0))\ \texttt{=}\ S_2 t_1'\ \texttt{in}\ t_2'\ \texttt{end}.$$

We now wish to "apply" S_2 to the sentence (176), using Lemma A.1. By (176) we have $B_1' \subseteq S_1 A_{11}$. Thus from (203) we get

$$\begin{pmatrix} S_1 A_{11} \\ B_1 \uplus B_1' \end{pmatrix} \uplus \begin{pmatrix} S_1 A_{12} \cup B_1 \cup B_1' \\ B_1 \uplus B_1' \end{pmatrix} \qquad \text{exists} \tag{212}$$

and from (205),

$$S_2 \text{ is a contraction of } S_1 A_{12} \cup B_1 \cup B_1'. \tag{213}$$

Also, from (206) and (208) we have

$$S_2(S_1A_{12} \cup B_1 \cup B'_1) \supseteq B' \cup S_2B'_1 \supseteq S_2(B_1 \uplus B'_1). \tag{214}$$

Since $\vdash S_1A_{12} \cup B_1 \cup B'_1$ and S_2 is a contraction on $S_1A_{12} \cup B_1$ we have $\vdash S_2(S_1A_{12}) \cup S_2B_1 \cup S_2B'_1$, so

$$\vdash B' \cup S_2B'_1 \tag{215}$$

since $\vdash B'$ and $B' \supseteq S_2B_1$. By Lemma A.1 on (176), and (212)–(215) we get

$$\begin{pmatrix} S_2(S_1A_{11}) \cup B' \cup S_2B'_1 \\ B' \uplus S_2B'_1 \end{pmatrix}, S_2(S_1TE) + \{f \mapsto S_2(\hat{\sigma'_1}, S_1(\rho_0))\} \vdash_{\mathrm{W}} S_2t'_1 \tag{216}$$

$$\begin{pmatrix} S_2(S_1A_{11}) \cup B' \cup S_2B'_1 \\ B' \uplus S_2B'_1 \end{pmatrix} \uplus \begin{pmatrix} S_2(S_1A_{12} \cup B_1 \cup B'_1) \\ B' \uplus S_2B'_1 \end{pmatrix} \quad \text{exists.} \tag{217}$$

Since $Inv(S_2) \cap Dom(B'_1) = \emptyset$ we get from (217) that

$$\begin{pmatrix} S_2(S_1A_{11}) \cup B' \cup S_2B'_1 \\ B' \end{pmatrix} \uplus \begin{pmatrix} S_2(S_1A_{12} \cup B_1) \\ B' \end{pmatrix} \quad \text{exists.} \tag{218}$$

We have $S_2B'_1 = bound(S_2\hat{\sigma_1})$, since $Inv(S_2) \cap bv(\hat{\sigma'_1}) = Inv(S_2) \cap Dom(B'_1) = \emptyset$. Also, $ftv(\vec{\alpha}) \cap S_2(S_1TE) = \emptyset$. Also, since we can apply S_2 to $\hat{\sigma'_1}$ without renaming of $bv(\hat{\sigma'_1})$, we have $body(S_2\hat{\sigma_1}) = S_2(body(\hat{\sigma_1}))$. Also $B' \uplus S_2B'_1$ exists and

$$S_2(S_1A_{11}) \cup B' \cup S_2B'_1 \cup S_2(S_1A_{12} \cup B_1) = S_2(S_1A) \tag{219}$$

by (164), (206), and (176). Also, by (174), (208), and (158) we have $B' \vdash ((S_1 \circ S_1)(TE), S_2(\varphi'_1))$. Thus by Rule 18 on (216), (218), and (206) we get

$$\begin{pmatrix} A \\ B' \end{pmatrix}, (S_2 \circ S_1)TE \vdash_{\mathrm{W}} e' : \mu' : S_2\varphi'_1 \cup \varphi'_2 \tag{220}$$

as required. Furthermore, S is a contraction of A, by (173), (174), and (205). Also $\mathrm{ML}(t') = \mathrm{ML}(t)$. Moreover, $B' \supseteq S(B)$, by (208) and (174). Moreover, $S(A) = A$ implies $S = \mathrm{Id}$, $t' = t$, and $B' = B$, by (201), (174), and (209). We have $\mu' = S(\mu)$, by (210). Since t'_1 has to start with a λ, we have $\varphi'_1 = \{S_1\,\rho_0\}$. Thus

$$\begin{array}{rcll} \varphi' & = & S_2\varphi'_1 \cup \varphi'_2 & \\ & = & S_2(S_1\varphi_{11}) \cup \varphi'_2 & \\ & \supseteq & S_2(S_1(\varphi_{11} \cup \varphi_{12})) & \text{by (210)} \\ & = & S(\varphi) & \text{by (163).} \end{array}$$

Since t is not strongly saturated, we have completed the proof concerning `letrec`.

Case $e = f_{il}$ `at` ρ'. We have

$$\begin{pmatrix} A \\ B \end{pmatrix}, TE \vdash_{\mathrm{P}} f_{il}\ \mathtt{at}\ \rho' : (\tau, \rho') : \varphi \tag{221}$$

where $(\tau, \rho') = \mu$. By Rule 15, there exist σ and ρ such that $TE(f) = (\sigma, \rho)$ and $B \vdash (TE, il, \mu)$, $A = B$, $\varphi = \{\rho, \rho'\}$, and

$$\mathrm{ML}(\tau) = \mathrm{ML}(S_I(\tau_0)) \tag{222}$$

where $\tau_0 = body(\sigma)$ and $S_I = zip(\sigma, il)$. By Lemma 6.5.1 on $B \vdash \sigma$ and $B \vdash il$ we have that line 2 of $\mathcal{R}$ succeeds and

$$S_1^e \text{ is a contraction of } B \tag{223}$$

$$S_1^e(\sigma) \geq \tau_1 \ \textit{via}\ S_1^e(il) \quad \text{and} \quad S_1^e(B) \vdash \tau_1 \tag{224}$$

$$S_1^e(B) = B \Rightarrow S_1^e = \mathrm{Id}^e. \tag{225}$$

From (222) and (224) we get

$$\mathrm{ML}(S_1^e(\tau)) = \mathrm{ML}(\tau_1). \tag{226}$$

Now $B \vdash \tau$ implies $S_1^e(B) \vdash S_1^e\tau$. Using Lemma 5.4.2 on this and $S_1^e(B) \vdash \tau_1$ and (226) we get that line 4 of $\mathcal{R}$ succeeds and

$$S_2 \text{ is a most general unifier for } \tau_1 \text{ and } S_1^e(\tau) \text{ in } S_1^e(B) \tag{227}$$

$$S_2 \text{ is a contraction of } S_1^e(B) \tag{228}$$

$$S_2(S_1^e B) = S_1^e(B) \Rightarrow S_2 = \mathrm{Id}. \tag{229}$$

Let $S = S_2 \circ S_1^e$. From $B \vdash (TE, il, \mu)$ and the fact that S is a contraction of B we get $S(B) \vdash (S(TE), S(il), S(\mu))$. Moreover, $(S\,TE)(f) = (S(\sigma), S\,\rho)$ and from (224) we have $S(\sigma) \geq S_2(\tau_1)\,\textit{via}\,S(il)$; that is, by (227), $S(\sigma) \geq S(\tau)\,\textit{via}\,S(il)$. Thus

$$\begin{pmatrix} S(B) \\ S(A) \end{pmatrix}, S(TE) \vdash_{\mathrm{W}} f_{S(il)}\ \mathtt{at}\ S(\rho') : S(\mu) : S(\varphi)$$

as required. Certainly $\mathrm{ML}(f_{S(il)}\ \mathtt{at}\ S(\rho') : S(\mu) : S(\varphi)) = f = \mathrm{ML}(t)$. Also, $B' = S(B) \supseteq S(B)$. Also $S(A) = A$ implies $S = \mathrm{Id}$, $t' = t$, and $B' = B$, by (229) and (225). Also, $\mu' = S(\mu)$ and $\varphi' = S(\varphi)$. Finally, $B' = Below(S(A), S(B))$, since $A = B$.

Case $e = x$. Straightforward.

Case $e = \mathtt{letregion}\ B_1\ \mathtt{in}\ t_1\ \mathtt{end}$. We have that t is strongly saturated and that $\binom{A}{B}, e \vdash_{\mathrm{P}} t$ must have been inferred by Rule 19. Thus

$$t_1 \text{ is saturated} \tag{230}$$

and $B \uplus B_1$ exists. Moreover, there exist $e_1$and φ_1 such that

$$\begin{pmatrix} A \\ B \uplus B_1 \end{pmatrix}, TE \vdash_{\mathrm{P}} e_1 : \mu : \varphi_1 \tag{231}$$

$$t_1 = e_1 : \mu : \varphi_1 \tag{232}$$

$$B \vdash TE, \mu \tag{233}$$

$$\varphi = Observe(B, \varphi_1). \tag{234}$$

By induction on (230) and (231), the call in line 21 of $\mathcal{R}$ succeeds and

$$S \text{ is a contraction of } A \tag{235}$$

$$\begin{pmatrix} S(A) \\ B_2 \end{pmatrix}, S(TE) \vdash_{\mathrm{W}} e_2 : \mu_2 : \varphi_2 \tag{236}$$

$$B_2 \supseteq S(B \uplus B_1) \tag{237}$$

$$S(A) = A \Rightarrow (S = \mathrm{Id} \wedge t_2 = t_1 \wedge B_2 = B \uplus B_1) \tag{238}$$

$$\mu_2 = S(\mu) \wedge \varphi_2 \supseteq S(\varphi_1) \tag{239}$$

where $t_2 = e_2 : \mu_2 : \varphi_2$. After line 22 of $\mathcal{R}$ we have

$$B_2 = (B_2 \setminus B') \uplus B' \text{ and } \vdash B' \text{ and } B' \supseteq S(B) \tag{240}$$

From (240) and (233) we get

$$B' \vdash S(TE), \mu_2. \tag{241}$$

By Rule 19 on (236), (240), and (241) we then get

$$\begin{pmatrix} S(A) \\ B' \end{pmatrix}, S(TE) \vdash_{\mathrm{W}} \texttt{letregion}\ B_2 \setminus B'\ \texttt{in}\ t_2\ \texttt{end} : \mu_2 : \varphi' \tag{242}$$

as required. Next, assume $S(A) = A$. Then, by (238), we have $S = \mathrm{Id}$, $t_2 = t_1$, and $B_2 = B \uplus B_1$. Then $B' = \mathit{Below}(B \uplus B_1, B) = B$ and $t' = t$, as required. We have already proved $\mu' = \mu_2 = S(\mu)$. We are left with proving $\varphi' \supseteq S(\varphi)$:

$$\begin{array}{rcll} \varphi' & = & \mathit{Observe}(B', \varphi_2) & \\ & \supseteq & \mathit{Observe}(B', S(\varphi_1)) & \text{by (239)} \\ & \supseteq & \mathit{Observe}(S(B), S(\varphi_1)) & \text{by (240)} \\ & \supseteq & S(\mathit{Observe}(B, \varphi_1)) & \text{by Lemma 7.3.2} \\ & = & S(\varphi) & \end{array}$$

as required. Finally, we have $B' = \mathit{Below}(B_2, S(B)) = \mathit{Below}(S(A), S(B))$, by (236) and (237), as desired.

Case $e = \lambda x : \mu_x . t_1$. We have that t is strongly saturated and

$$\begin{pmatrix} A \\ B \end{pmatrix}, TE \vdash_{\mathrm{P}} \lambda x : \mu_x . t_1 : (\tau, \rho) : \varphi \tag{243}$$

where $(\tau, \rho) = \mu$. Now (243) must have been inferred using Rule 16. Thus $\varphi = \{\rho\}$ and there exist μ_2, $\epsilon.\varphi_0$, μ_1, e_1 such that $\mu_2 \xrightarrow{\epsilon.\varphi_0} \mu_1 = \tau$, $\mu_2 = \mu_x$, and

$$\begin{pmatrix} A \\ B \end{pmatrix}, TE + \{x \mapsto \mu_2\} \vdash_{\mathrm{P}} t_1 \tag{244}$$

$$t_1 = e_1 : \mu_1 : \varphi_1 \tag{245}$$

$$\varphi_0 \supseteq \varphi_1 \text{ and } (\{\rho\}, \{\epsilon.\varphi_0\}) \subseteq B \tag{246}$$

$$B \vdash TE. \tag{247}$$

In particular, line 6 of $\mathcal{R}$ succeeds. Moreover, since t is strongly saturated, t_1 is also strongly saturated. Hence by induction on (244) we get that line 7 of $\mathcal{R}$ succeeds and that

$$S_1 \text{ is a contraction of } A \tag{248}$$

$$\begin{pmatrix} S_1(A) \\ B_1 \end{pmatrix}, S_1(TE) + \{x \mapsto S_1(\mu_2)\} \vdash_{\mathrm{W}} t_1' \tag{249}$$

$$\mathrm{ML}(t_1') = \mathrm{ML}(t_1) \tag{250}$$

$$B_1 \supseteq S_1(B) \tag{251}$$

$$S_1(A) = A \Rightarrow (S_1 = \mathrm{Id} \wedge t_1' = t_1 \wedge B_1 = B) \tag{252}$$

$$\mu_1' = S_1(\mu_1) \text{ and } \varphi_1' \supseteq S_1(\varphi_1) \tag{253}$$

$$B_1 = \mathit{Below}(S_1(A), S_1(B)). \tag{254}$$

As in $\mathcal{R}$, let $\epsilon_1'.\varphi_1'' = S_1(\epsilon.\varphi_0)$. By (246) we have $\varphi_1'' \supseteq S_1(\varphi_1)$; we also have $\varphi_1' \supseteq S_1(\varphi_1)$. (However, we need neither have $\varphi_1'' \supseteq \varphi_1'$ nor $\varphi_1' \supseteq \varphi_1''$.) After lines 9 and 10 of $\mathcal{R}$ we have that

$$S_2 \text{ is a contraction of } B_1 \text{ and } S_2(B_1) \text{ is consistent} \tag{255}$$

$$S_2(S_1(A)) = S_1(A) \Rightarrow S_2 = \mathrm{Id} \tag{256}$$

$$\varphi \text{ } \mathit{of} \text{ } S(\epsilon.\varphi_0) \supseteq S_2(\varphi_1') \text{ and } (\{S(\rho)\}, \{S(\epsilon.\varphi_0)\}) \subseteq S_2(B_1). \tag{257}$$

By Lemma A.1.2 on (249) and (255) we get

$$\begin{pmatrix} S(A) \\ S_2(B_1) \end{pmatrix}, S(TE) + \{x \mapsto S(\mu_2)\} \vdash_{\mathrm{W}} S_2 t_1'. \tag{258}$$

From (247), (251), (249), and (255) we have $S_2(B_1) \vdash S(TE)$. By Rule 16 on this, (258), and (257) we get

$$\begin{pmatrix} S(A) \\ S_2(B_1) \end{pmatrix}, S(TE) \vdash_{\mathrm{W}} (\lambda\, x : S(\mu_2).S_2(t_1')) : (S\mu_2 \xrightarrow{S(\epsilon.\varphi_0)} S(\mu_1), S\,\rho) : \{S(\rho)\} \tag{259}$$

as desired. Certainly, S is a contraction on A and, by (250), $\mathrm{ML}(t') = \mathrm{ML}(t)$. Also, $S_2 B_1 \supseteq S_2(S_1 B) = S(B)$, by (251). Assume $S(A) = A$. Then $S_2 = \mathrm{Id}$ and $S_1 = \mathrm{Id}$ by (252) and (256), so $S = \mathrm{Id}$. Moreover, by (252) we have $t_1' = t_1$ and $B_1 = B$, so $t' = t$ and $S_2(B_1) = B$. Clearly, $\mu' = S(\mu)$ and $\varphi' = \{S(\rho)\} = S(\{\rho\}) = S(\varphi)$. Finally, t' is strongly saturated and, by (254), we have $S_2(B_1) = S_2(\mathit{Below}(S_1(A), S_1(B))) = \mathit{Below}(S(A), S(B))$, since $B_1 \vdash \varphi_1'$.

Case $e = t_1\ \ t_2$. Here $\binom{A}{B}, TE \vdash_{\mathrm{P}} t_1\ t_2 : \mu : \varphi$ must have been inferred by Rule 17, so there exist A_1, A_2, e_1, φ_1, μ', $\epsilon_0.\varphi_0$, ρ_0, e_2, and φ_2 such that

$$\begin{pmatrix} A_1 \\ B \end{pmatrix}, TE \vdash_{\mathrm{P}} t_1 \text{ and } t_1 = e_1 : (\mu' \xrightarrow{\epsilon_0.\varphi_0} \mu, \rho_0) : \varphi_1 \tag{260}$$

$$\begin{pmatrix} A_2 \\ B \end{pmatrix}, TE \vdash_{\mathrm{P}} t_2 \text{ and } t_2 = e_2 : \mu' : \varphi_2 \tag{261}$$

$$\begin{pmatrix} A \\ B \end{pmatrix} = \begin{pmatrix} A_1 \\ B \end{pmatrix} \uplus \begin{pmatrix} A_2 \\ B \end{pmatrix} \tag{262}$$

$$\varphi = \varphi_1 \cup \varphi_2 \cup \{\epsilon_0, \rho_0\} \cup \varphi_0. \tag{263}$$

Since t is saturated, we have have that t_1 and t_2 are strongly saturated. Then by induction on (260) we have that line 12 in $\mathcal{R}$ succeeds and

$$S_1 \text{ is a contraction on } A_1 \tag{264}$$

$$\begin{pmatrix} S_1 A_1 \\ B_1 \end{pmatrix}, S_1\, TE \vdash_{\mathrm{W}} t'_1 \tag{265}$$

$$B_1 \supseteq S_1(B) \tag{266}$$

$$S_1\, A_1 = A_1 \Rightarrow (S_1 = \mathrm{Id} \wedge t'_1 = t_1 \wedge B_1 = B) \tag{267}$$

$$(\mu'_1 \xrightarrow{\epsilon'_0 . \varphi'_0} \mu''_1, \rho') = S_1(\mu' \xrightarrow{\epsilon_0 . \varphi_0} \mu, \rho_0) \tag{268}$$

$$\varphi'_1 \supseteq S_1\, \varphi_1 \tag{269}$$

$$t'_1 \text{ is strongly saturated and } \mathrm{ML}(t'_1) = t_1 \tag{270}$$

$$B_1 = \mathit{Below}(S_1\, A_1, S_1\, B). \tag{271}$$

We have $S_1\, A_1 \supseteq B_1 \supseteq S_1\, B$. Using Lemma A.1 on this and (261), (262), and (264) we get

$$\begin{pmatrix} S_1(A_2) \cup B_1 \\ B_1 \end{pmatrix}, S_1\, TE \vdash_{\mathrm{P}} S_1\, e_2 : S_1\, \mu' : S_1\, \varphi_2 \tag{272}$$

$$\begin{pmatrix} S_1\, A_1 \\ B_1 \end{pmatrix} \uplus \begin{pmatrix} S_1(A_2) \cup B_1 \\ B_1 \end{pmatrix} \qquad \text{exists.} \tag{273}$$

By Lemma A.1.3 we have that (272) is proved by a proof of the same depth as the proof of (261). Since $S_1\, t_2$ is strongly saturated we can use induction on (272) and get that line 13 of $\mathcal{R}$ succeeds and

$$S_2 \text{ is a contraction of } S_1(A_2) \cup B_1 \tag{274}$$

$$\begin{pmatrix} S_2(S_1\, A_2) \cup S_2(B_1) \\ B' \end{pmatrix}, S_2(S_1 TE) \vdash_{\mathrm{W}} t'_2 \tag{275}$$

$$B' \supseteq S_2\, B_1 \tag{276}$$

$$S_2(S_1(A_2) \cup B_1) = S_1(A_2) \cup B_1 \Rightarrow (S_2 = \mathrm{Id} \wedge t'_2 = S_1(t_2) \wedge B' = B_1) \tag{277}$$

$$\mu'_2 = S_2(S_1(\mu')) \wedge \varphi'_2 \supseteq S_2(S_1(\varphi_2)) \tag{278}$$

$$t_2' \text{ is strongly saturated and } \mathrm{ML}(t_2') = \mathrm{ML}(S_1\, t_2) \tag{279}$$

$$B' = \mathit{Below}(S_2(S_1(A_2) \cup B_1), S_2\, B_1). \tag{280}$$

Let S and φ' be as in $\mathcal{R}$. $\mathcal{R}$ succeeds and we have $S_2(S_1(A_2) \cup B_1) \supseteq B' \supseteq S_2(B_1)$. Using Lemma A.1 on this, (265), (273), and (274) we get

$$\begin{pmatrix} S(A_1) \cup B' \\ B' \end{pmatrix}, S\,TE \vdash_{\mathrm{W}} S_2\, t_1' \tag{281}$$

$$\begin{pmatrix} S(A_2) \cup S_2(B_1) \\ B' \end{pmatrix} \uplus \begin{pmatrix} S(A_1) \cup B' \\ B' \end{pmatrix} \qquad \text{exists.} \tag{282}$$

Now

$$\begin{array}{rcll} S(A_1) \cup S_2\, B_1 \cup S(A_2) \cup B' & = & S(A_1) \cup S(A_2) \cup B' & \text{by (276)} \\ & = & S_2(S_1(A_1 \cup A_2)) & \text{by (271) and (280)} \\ & = & S(A) & \text{by (262).} \end{array}$$

Also, $S_2 t_1' = S_2(e_1' : (\mu_1' \xrightarrow{\epsilon_0'.\varphi_0'} \mu_1'', \rho') : \varphi_1') = S_2(e_1') : S(\mu' \xrightarrow{\epsilon_0.\varphi_0} \mu, \rho_0) : S_2(\varphi_1')$, by (268). Moreover, $\mu_2' = S(\mu')$, by (278). Thus by Rule 17 on (281), (275), and (282) we get the desired

$$\begin{pmatrix} S(A) \\ B' \end{pmatrix}, S(TE) \vdash_{\mathrm{W}} ((S_2\, t_1') t_2') : S(\mu) : \varphi'.$$

We have that S is a contraction of A, by (264) and (274). Also, $B' \supseteq S_2(B_1) \supseteq S(B)$, by (280) and (266), as required. Moreover, $S(A) = A$ implies $S = \mathrm{Id}$, $t' = t$, and $B' = B$, by (267) and (277). Finally,

$$\begin{array}{rcll} \varphi' & = & S_2\varphi_1' \cup S_2\{\epsilon_0', \rho'\} \cup S_2\varphi_0' \cup \varphi_2' & \\ & \supseteq & S(\varphi_1) \cup S\{\epsilon_0, \rho_0\} \cup S(\varphi_0) \cup S(\varphi_2) & \text{by (268), (269), and (278)} \\ & = & S(\varphi). & \end{array}$$

Since t is not strongly saturated, we are done with this case. This concludes the proof of the theorem. $\square$