Multiplicity Inference

Inferring sizes of regions

Master's thesis, DIKU 1994

Author: Magnus Vejlstrup

Supervisor: Mads Tofte

Table of Contents

Foreword

This report is about implementation of functional programming languages. In these languages the programmer does not need to worry about de-allocating data structures, when they go out of use. This is tranditionally done for him by the garbage collector of the run-time system.

Region inference is a new program analysis, developed by Mads Tofte and Jean-Pierre Talpin, which translates a functional program to a *region expression*, where the all values produced by the expression are explicitly put in regions, which are allocated and de-allocated in stack order. A region expression takes care of memory management itself and does not depend on help from a garbage collector to do this.

Traditional imperative programming languages like Pascal and C work partly in this way, too, as activation records, which contain local variables, are stacked at run-time. But whereas it is easy in these languages to determine at compile-time how much memory must be allocated for the local variables in the activation record, it is less obvious in region inference how much memory a region takes up. This is a problem, if regions are to be stacked on top of each other.

Multiplicity inference, which we present in this report, takes as its input an annotated region expression and produces an expression whose regions have been attached sizes. Such an analysis has not been made before.

In the first chapter we discuss memory management in general, how it is handled both in imperative languages and in traditional implementations of functional languages, and how memory could be managed by a region program.

We then dive into region inference in more detail in the second chapter. We describe the dynamic semantics of its source and target languages, as well as the inference rules for translation.

Multiplicity inference is presented in the third chapter. We first describe the input and output languages of the analysis. Then we give a set of inference rules for translation.

In the fourth chapter we construct a translation algorithm. This algorithm is proven sound with respect to the translation semantics of multiplicity inference. We also discuss how it can be implemented efficiently.

We discuss experimental results from a prototype region inference compiler with multiplicity inference in the fifth chapter.

Finally, in the sixth chapter, we sum up our results.

The reader should be familiar with type inference

1 Introduction

In computer systems processor time and memory will always be scarce resources. The way memory is allocated and reclaimed is of paramount importance for the utilization of both.

One of the great inventions in the history of programming languages is the use of a stack as a means of storage allocation for block-oriented languages. This classical technology has two main virtues

- For most programs it gives excellent memory utilization. A memory cell will usually be reclaimed shortly after it goes out of use.
- It is very efficient. Memory is allocated and reclaimed by updating the stack pointer.

In the traditional imperative languages the life times of the local variables of a procedure is the time between its activation and its termination. Therefore it is natural to create blocks when procedures are invoked and reclaim them at procedure return. So the local variables of a procedure are allocated as part of its activation record.

This stack discipline only solves the memory allocation problems for shallow objects, i.e. objects which contain no pointers. The allocation and de-allocation of deep objects such as trees and lists have in traditional programming languages such as Pascal and C been delegated to application programmers. In these languages the automatic memory allocation is called *static*, and the memory management taken care of by the programmer is called *dynamic*. The usually non-trivial problem of dynamic memory management, which often involves dangerous traversals of partly un- or de-allocated pointer structures and applications of low-level system calls, puts a tremendous strain even on experienced programmers.

Modern programming languages provide built-in deep data types such as lists, recursive data types, and functions. In these languages the programmer is freed from worrying about the de-allocation of the data structures which he constructs. This is necessary, for whereas manual memory management is hard work in traditional programming languages, it would be unbearable in a language which has data objects such as closures. But by relieving the application programmer of the yoke of data de-allocation, the language implementer is left with a hard memory management problem himself.

Traditionally, this problem is solved by using some kind of garbage collection technique to take care of de-allocation.

1.1 Garbage collection

In a system with garbage collection the execution of a program is interleaved with with a coroutine, called the garbage collector, which at intervals analyses the store, finds out which data are live, and then cleans up the dead areas.

Garbage collection consists of two phases: identification of live data, and removal of garbage.

The rule which the garbage collector uses for identifying live data is that everything pointed to by a live object is live. And all data in the *root set* [Wil92, p. 3], i.e. the global variables, the variables on the stack, and in registers are live. So the garbage collectors starts at the root set and follows all pointers from there into the store. All objects must have tags which tell the garbage collector which of the memory words are pointers. This process is often referred to as *tracing out* the live data. The data which are identified as

live in this way are, of course, not necessarily live in the sense that all the pointers will be dereferenced by the program in the future. But at least we have not missed any data which could potentially be live.

Now the live data been identified, comes the phase of cleaning up garbage. There are two basic ways of doing this. *Mark-sweep* collectors de-allocate garbage explicitly. *Copying collectors* ignore the dead data and just copy the live data into one contiguous block of memory. It is not clear which of the approaches is the best. Mark-sweep collectors have the advantage that de-allocating is faster than copying. Copying have the advantage that the work to be done when collecting garbage is proportional to the amount of live data, and if the amount of live data is small, it is nice to be able to ignore the dead data altogether.

Usually the garbage collector is envoked when the program runs out of memory. But for real-time applications *incremental* garbage techniques have been developed, i.e. techniques that allow garbage collection to be more fine-grainedly interlieved with the execution of the program, so that the program will not be down for seconds while the whole memory is being inspected [Wil92, p. 20-30].

All kinds of garbage collectors face two main problems

- Poor memory utilization. A memory cell can stay unused for a long time before being reclaimed.
- Speed. Garbage collection can result in tracing out and deleting/copying great amounts of data.

These two problems can be solved separately, but, unfortunately, there is an unresolvable conflict between them. The garbage collection overhead can be diminished by reserving more memory for the program, because the program will then not have to collect garbage as often. But this means poorer memory utilization: more memory is used for doing the same. Correspondingly, one could improve memory utilization by giving the program a minimal block of memory. But then it will probably spend much of its time collecting garbage. It is in the quest of getting less overhead in a smaller memory that garbage collection has grown into a science in its own right.

One interesting development is the *generational* garbage collectors. They are based on the insight that the more recently an object has been allocated the shorter will its lifetime probably be. This is what is called *the strong generational hypothesis* in the [Hay91].

In [LH83] a scheme is presented where the heap is organized as a list of subheaps, of which the newly created ones are garbage collected more often than the old ones. A new value is always allocated in the newest subheap. When the newest subheap is full, a new one is created and appended to the list. As time goes by, the oldest subheaps will become more and more sparse, and maybe even disappear. Here performance is gained, because it is the parts of memory, the newest subheaps, where there is likely to be a lot of dead data, which are inspected most frequently by the garbage collector, whereas the older and more permanent objects in the old subheaps are only looked at rarely.

The garbage collection scheme in [LH83] could seem like a simulation of a stack discipline. And what the strong generational hypothesis really says is that stack-order allocation of objects would be appropriate. The problem is, of course, that we must know in which order objects should be pushed onto the stack, and this requires knowledge about when it is sound to de-allocate them. A new program analysis, *region inference*, infers this information.

1.2 Region inference - a deep stack discipline

As mentioned before, stack allocation in the traditional form does not manage deep objects. To do this, it is necessary to distinguish block allocation from the mecanics of procedure invocation. That the two do not go hand in hand for deep objects does not mean that it is impossible to put them in blocks that are allocated in stack order.

But in the absence of procedure activations as the point of memory allocation we are left with two problems: we must find out which values to put in which blocks and decide when the blocks should be allocated and deallocated. These two problems are solved by region inference[TT93a, TT93b]. The input of

the analysis is the typed call-by-value λ-calculus with let-polymorphism and recursive functions. The output is a program which is identical to the input, except for some region annotations. There are two ways an expression can be annotated

- e `at` ρ - signifying that the value produced by e is put in region ρ.
- `letregion` ρ `in` e `end` - this constructs introduces a region ρ that can be used in e for storing values.

Example 1.2. This expression

$$\texttt{fst}(1, 2)$$

would by region inference be annotated like this

$$\begin{array}{l}\texttt{letregion}\ \rho_2, \rho_3\ \texttt{in}\\ \quad \texttt{fst}(1\ \texttt{at}\ \rho_1, 2\ \texttt{at}\ \rho_2)\ \texttt{at}\ \rho_3\\ \texttt{end}\end{array}$$

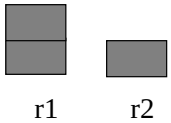
After executing this statement, only ρ_1, which contains 1, is left. The regions ρ_2 and ρ_3 are temporary, and therefore wrapped up in a letregion-declaration.

At run-time an expression `letregion` ρ `in` e `end` is executed by binding a concrete region `r` in the store to ρ. A subexpression e' `at` ρ is then evaluated by putting the value produced by e into a fresh memory cell in `r`, the store region currently attached to ρ. So `r` will grow every time an '`at` ρ'-expression is executed. At the '`end`' matching the `letregion` ρ the store region `r` is deallocated. The fact that letregion-expressions are nested within each other ensures that regions are allocated and deallocated in stack order.

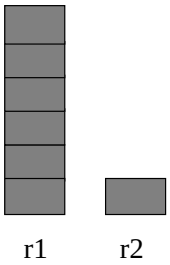
Let us consider the following program At the point where the `letregion` ρ_1 has just been executed, the store looks like this

where `r1` is the store region attached to ρ_1. Suppose the program now puts a value into `r1`. Then the store looks like this at just after `letregion` ρ_2

Here `r2` is the store region that has been bound to ρ_2. After `letregion` ρ_3 the store could look like this



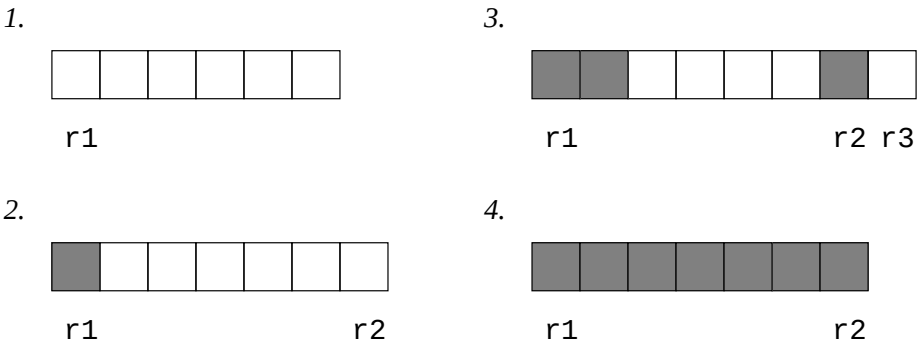
Here `r1` and `r2` have grown again. After the first `end` the store contains



As can be seen from these snapshots regions continue to grow until they are deallocated. This complicates the problem of mapping store regions to concrete blocks of memory when implementing the region language. In the classical stack discipline we know the size of an activation record at allocation-time, which makes it possible to stack the blocks on top of each other. If we could find upper bounds for the regions' sizes at compile time, they could be stacked in the same way. This could be done by an analysis which could take the above program as input and produce a program like this one

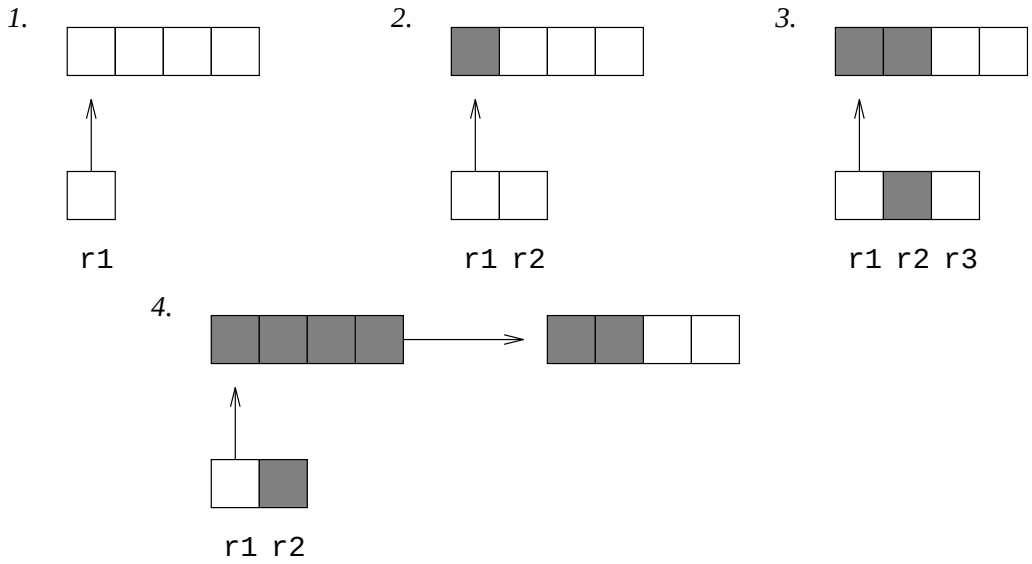
```
letregion ρ1 : 6 in
          ...
          letregion ρ2 : 1 in
                    ...
                    letregion ρ3 : 1 in
                              ...
                    end
                    ...
          end
          ...
end
```

If we look at the store at the same breakpoints as before, it will now look like this



But we have perhaps been a bit over-optimistic. In the presence of recursive functions, we cannot expect to be able to determine the exact sizes of all regions. Now and then the analysis will probably have to resign to giving some regions size ∞.

In the program above, ρ_1 might be such a region. Clearly, a region whose size is unknown cannot fit into the stack itself. But it can be represented by a pointer to the heap, where it could be contained in a linked list of blocks. The program that is identical to the one above, but where ρ_1 has size ∞ would make the memory look like this, assuming that the block size in the heap is 4:



It is obvious that regions of unknown size impose some overhead on the program. But if they are rare, the expense will be bearable.

In the next chapters we shall address the problem of constructing an analysis which given a region expression produces a program whose letregion-expressions are annotated with sizes.

But we must first deal with region inference in greater detail.

2 Region inference

In this chapter region inference will be dealt with in some detail. The analysis takes as its input a functional program and produces a region expression, a more low-level program with explicit storage allocation instructions.

After having introduced some basic notation, we present these source and target languages, and then the translation semantics.

The following sections are based on [TT93a, TT93b].

2.1 Notation

In this section we introduce some basic notation.

By $\mathrm{fin}(A)$ we mean the set of finite subsets of A, and by $\mathrm{Vec}(A)$ the set of finite sequences of members of A. By $A \xrightarrow{\mathrm{fin}} B$ we denote the set of finite maps from A to B. We use $\{\}$ to denote the empty set or the empty map.

By $\mathbf{dom}(f)$ and $\mathbf{rng}(f)$ we mean the domain and range of a map f, the restriction of f to A is denoted $f\downarrow_A$, and $f \setminus\setminus A$ is the restriction of f to $\mathbf{dom}(f) \setminus A$. When f and g are maps then $f+g$ is the map defined by

$$(f+g)(a) = \text{if } a \in \mathbf{dom}(g) \text{ then } g(a) \text{ else } f(a) \qquad \textit{, for all } a \in \mathbf{dom}(f) \cup \mathbf{dom}(g)$$

When $\mathbf{dom}(f) \cap \mathbf{dom}(g) = \{\}$ we also write $f \mid g$ for $f+g$.

By $\vec{v} \setminus\setminus \{i_1, \ldots, i_j\}$ we mean the vector obtained by removing from $\vec{v}$ the components at position $i_1, \ldots, i_j$. If v occurs in $\vec{v}$ only at position i, we regard $\vec{v} \setminus\setminus \{v\}$ as synonymous to $\vec{v} \setminus\setminus \{i\}$. Let $\vec{x} = (x_1, \ldots, x_k)$ and $\vec{y} = (y_1, \ldots, y_k)$ be a vectors. Then we use $[\vec{x}/\vec{y}]$ as a shorthand for the finite map $\{y_1 \mapsto x_1, \ldots, y_k \mapsto x_k\}$.

2.2 The source language

The source expressions, SExp, are defined by the following grammar

$$\begin{array}{rcl} e & ::= & \texttt{true} \mid \texttt{false} \mid x \mid f \mid \lambda x.e \mid e\ e' \mid \texttt{let}\ x = e\ \texttt{in}\ e' \mid \\ & & \texttt{letrec}\ f = e\ \texttt{in}\ e' \mid \texttt{if}\ e\ \texttt{then}\ e'\ \texttt{else}\ e'' \end{array}$$

Here x and f range over the set Var of variables. For brevity we have omitted constants, arithmetic operators, and tuples from the grammar. But the language contains booleans and conditionals, as the if-then-else-expression will get special treatment in our analysis. The semantic objects are the following

$$\begin{array}{rrcl} v \in & \mathrm{Val} & = & \mathrm{Bool} \cup \mathrm{Clos} \cup \mathrm{RecClos} \\ & \mathrm{Bool} & = & \{\texttt{true}, \texttt{false}\} \\ \langle x, e, E\rangle \in & \mathrm{Clos} & = & \mathrm{Var} \times \mathrm{SourceExp} \times \mathrm{Env} \\ \langle x, e, E, f\rangle \in & \mathrm{RecClos} & = & \mathrm{Var} \times \mathrm{SourceExp} \times \mathrm{Env} \times \mathrm{Var} \\ E \in & \mathrm{Env} & = & \mathrm{Var} \xrightarrow{\mathrm{fin}} \mathrm{Val} \end{array}$$

We distinguish between recursive functions and plain ones. Recursive functions have an extra name component so it can add itself to the environment every time it is applied, as can be seen below.

The dynamic semantics of the language contains nothing extraordinary.

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

$$\frac{E(x) = v}{E \vdash x \to v} \qquad \frac{}{E \vdash \lambda x.e \to \langle x, e, E\rangle}$$

$$\frac{}{E \vdash \texttt{true} \to \texttt{true}} \qquad \frac{}{E \vdash \texttt{false} \to \texttt{false}}$$

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

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

$$\frac{E \vdash e_1 \to v_1 \quad E + \{x \mapsto v_1\} \vdash e_2 \to v}{E \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \to v} \qquad \frac{E + \{f \mapsto \langle x, e_1, E, f\rangle\} \vdash e_2 \to v}{E \vdash \texttt{letrec}\ f(x) = e_1\ \texttt{in}\ e_2 \to v}$$

$$\frac{E \vdash e_1 \to \texttt{true} \quad E \vdash e_2 \to v}{E \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \to v} \qquad \frac{E \vdash e_1 \to \texttt{false} \quad E \vdash e_3 \to v}{E \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \to v}$$

2.3 The target language

In this section we shall discuss the region language, which explicitly puts values in regions and explicitly allocates and deallocates regions. The expressions, TExp, are defined by the following grammar

$$
\begin{array}{rcl}
p & ::= & \rho \mid r \\
e & ::= & \texttt{true at } p \mid \texttt{false at } p \mid x \mid f[\vec{p}] \texttt{ at } p \mid \lambda x.e \texttt{ at } p \mid e\ e' \mid \texttt{let } x = e \texttt{ in } e' \mid \\
 & & \texttt{letrec } f\,[\vec{\rho}] \texttt{ at } p = e \texttt{ in } e' \mid \\
 & & \texttt{letregion } \rho_1 \cdots \rho_k \texttt{ in } e \mid \texttt{if } e \texttt{ then } e' \texttt{ else } e''
\end{array}
$$

All value-producing expressions are annotated with the place where the value is to be put. A place is either a region variable or a region name. A region variable, a member of the set RegVar $= \{\rho_1, \rho_2, \ldots\}$, should be viewed as an abstract or potentiel region, which is instantiated to concrete a store region when its letregion-declaration is executed. Concrete regions are represented by the set RegName of region names, which is ranged over by r. The evaluation of the letregion-expression, as can be seen below, rely on substituting region names for region variables in expressions. The semantic objects are the following

$$
\begin{array}{rcccl}
o & \in & \text{Offset} & & \\
r & \in & \text{RegionName} & & \\
s & \in & \text{Store} & = & \text{RegionName} \xrightarrow{\text{fin}} \text{Region} \\
reg & \in & \text{Region} & = & \text{Offset} \xrightarrow{\text{fin}} \text{StoreVal} \\
v & \in & \text{TargetVal} & = & \text{Addr} \\
sv & \in & \text{StoreVal} & = & \text{Bool} \cup \text{Clos} \cup \text{RegionClos} \\
 & & \text{Bool} & = & \{\texttt{true}, \texttt{false}\} \\
\langle x, e, VE \rangle & \in & \text{Clos} & = & \text{Var} \times \text{TargetExp} \times \text{VarEnv} \\
\langle \vec{\rho}, x, e, E \rangle & \in & \text{RegionClos} & = & \text{Vec(RegVar)} \times \text{Var} \times \text{TargetExp} \times \text{VarEnv} \\
a \text{ or } (r, o) & \in & \text{Addr} & = & \text{RegionName} \times \text{Offset} \\
VE & \in & \text{VarEnv} & = & \text{Var} \xrightarrow{\text{fin}} \text{TargetVal}
\end{array}
$$

In this language values are addresses. An address (r, o) consists of a name of the region and the offset within the region. A store is a map from region name to regions. Regions are maps from offsets to store values. So as we are dealing with to levels of indirection, we introduce some shorthand notation. Whenever a is and address (r, o) and s a store, we write $s(a)$ as a shorthand for $s(r)(o)$ and $a \in \mathbf{dom}(s)$ to mean $r \in \mathbf{dom}(s)$ and $o \in \mathbf{dom}(r)$. When sv is a storable value, we write $s + \{(r, o) \mapsto sv\}$ to mean $s + \{r \mapsto s(r) + \{o \mapsto sv\}\}$.

By $e[r_1/\rho_1 \cdots r_k/\rho_k]$ we mean the result of simultaneously substituting $r_1 \cdots r_k$ for $\rho_1 \cdots \rho_k$ in e. Recall that the notation $f \setminus\setminus \{A\}$ means the map f restricted to $\mathbf{dom}(f) \setminus A$. So $s \setminus\setminus \{r\}$ means the store obtained by removing the region r.

We have to types of closures. Plain closures are concrete functions that can be applied. Region closures contain an extra component, namely a vector of region variables. Instantiating these region variables to actual places renders a plain closure. That is why all variables that stand for region closures have the form $f[\vec{p}]$ `at` p, where $\vec{p}$ are the actual place parameters, and p is the place where the resulting plain closure is to be put. Correspondingly, the functions introduced by the letrec-construct are supplied with a vector of region variables, the formal region parameters. One could view region closures as curried functions that have to be applied to a vector of places before being able to be applied to their arguments. Region closures introduce *region polymorfism* into the language. Region polymorfism is crucial for memory utilization in the target language. It allows region closures to be instantiated to actual places whose life-times are shorter than that of the region closures.

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

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

$$\frac{o \notin \mathbf{dom}(s(r)) \quad VE(f) = a \quad s(a) = \langle \rho_1, \ldots, \rho_k, x, e, VE_0 \rangle \quad sv = \langle x, e[p_1/\rho_1, \ldots, p_k/\rho_k], VE_0 \rangle}{s, VE \vdash f[p_1, \ldots, p_k] \texttt{ at } r \to (r, o), s + \{(r, o) \mapsto sv\}}$$

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

$$\frac{s, VE \vdash e_1 \to a_1, s_1 \quad s_1(a_1) = \langle x_0, e_0, VE_0 \rangle \quad s_1, VE \vdash e_2 \to v_2, s_2 \quad s_2, VE_0 + \{x_0 \mapsto v_2\} \vdash e_0 \to v, s'}{s, VE \vdash e_1\ e_2 \to v, s'}$$

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

$$\frac{o \notin \mathbf{dom}(s(r)) \quad VE' = VE + \{f \mapsto (r, o)\} \quad s + \{(r, o) \mapsto \langle \rho_1, \ldots, \rho_k, x, e_1, VE' \rangle\}, VE' \vdash e_2 \to v, s'}{s, VE \vdash \texttt{letrec } f[\rho_1, \ldots, \rho_k](x) \texttt{ at } r = e_1 \texttt{ in } e_2 \to v, s'}$$

$$\frac{r \notin \mathbf{dom}(s) \quad s + \{r \mapsto \{\}\}, VE \vdash e[r/\rho] \to v, s_1}{s, VE \vdash \texttt{letregion } \rho \texttt{ in } e \texttt{ end} \to v, s_1 \setminus\setminus \{r\}}$$

$$\frac{s, VE \vdash e_1 \to a, s_1 \quad s_1(a) = \texttt{true} \quad s_1, VE \vdash e_2 \to v, s'}{s, VE \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \to v, s'}$$

$$\frac{s, VE \vdash e_1 \to a, s_1 \quad s_1(a) = \texttt{false} \quad s_1, VE \vdash e_3 \to v, s'}{s, VE \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \to v, s'}$$

Note that it is the letrec-bound functions that are region polymorfic. It is a design decision to let recursion and region polymorfism be combined in one construct. This poses no restrictions on the expressive power of the language, since letrec-bound functions need not be recursive and region polymorfic at once.

The target expressions do not differ fundamentally from the source expressions. A given target expression can be viewed as a more explicit low-level version of some underlying source expression. We define the following function, which maps target expressions to source expressions (by pealing off the region annotations):

$$
\begin{aligned}
\overline{c \texttt{ at } \rho} &= c \\
\overline{x} &= x \\
\overline{f[\vec{\rho}] \texttt{ at } \rho} &= f \\
\overline{\lambda x.e \texttt{ at } \rho} &= \lambda x.\overline{e} \\
\overline{e\ e'} &= \overline{e}\ \overline{e'} \\
\overline{\texttt{letrec } f\,[\vec{\rho}] \texttt{ at } \rho = e \texttt{ in } e'} &= \texttt{letrec } f = \overline{e} \texttt{ in } \overline{e'} \\
\overline{\texttt{let } x = e \texttt{ in } e'} &= \texttt{let } x = \overline{e} \texttt{ in } \overline{e'} \\
\overline{\texttt{letregion } \rho_1 \cdots \rho_k \texttt{ in } e} &= \overline{e} \\
\overline{\texttt{if } e \texttt{ then } e' \texttt{ else } e''} &= \texttt{if } \overline{e} \texttt{ then } \overline{e'} \texttt{ else } \overline{e''}
\end{aligned}
$$

2.4 Translation semantics

The aim of the analysis is to translate source programs to target expressions that do the same. As pointed out in the introduction this involves deciding which values to put in which blocks and finding out when it is safe to de-allocate a given block.

It is important for the performance of the analysis that only values of approximately the same life-time are put at the same place. For a region can only be soundly de-allocated when all its values have gone out of use. To group values according to their life-times, region inference uses the observation that values produced by two expressions e_1 and e_2 often have similar life-times, if the types of the expressions are unified during type checking [Bak90]. This makes type inference an ideal vehicle for deciding the region annotations of expressions, as the grouping can be integrated into the type checking mecanism by adding places to types. In this way, places are required to match at applications just as types are. So in addition to the target expression we must infer a type and its place. And to infer types we need a type environment.

The integration of types and places also helps to solve the problem of deciding at what points in the program regions can be de-allocated, i.e. to decide where the letregion-declarations should be inserted. Suppose we are translating an expression e and have just elaborated a subexpression e_0 and translated it to an expression e_0'. Before going any further in the translation of the program, we would like to know if it is sound to de-allocate the region ρ at this point, i.e. if we should wrap a `letregion` ρ-expression around e_0'. This is clearly not a good idea, if ρ occurs somewhere in the type environment or in the type and place inferred for e_0', as we would then expect the rest of the target program to access the region. On the other hand, we cannot be sure that a region can be safely de-allocated just because it is not the place for any type. For even though a function is itself situated in ρ_1 takes an argument from ρ_2 and produces a result in ρ_3, it might need to consult some information in ρ to compute the result. So to determine the life-times of regions we need additional information about which regions expressions access when they are executed. We call this information the *effect* of an expression.

All in all the translation relation looks like this

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

read: *in TE, e translates to e', which has type and place μ and effect φ.*

In region inference the exact definition of the semantic objects of the translation is as follows

$$
\begin{array}{rcll}
\alpha & \in & \text{TyVar} & \\
\rho & \in & \text{RegVar} & \\
\varepsilon & \in & \text{EffectVar} & \\
\upsilon & \in & \text{EffectVar} \cup \text{RegVar} \cup \text{TyVar} & \\
\varphi & \in & \text{Effect} & = \text{Fin}(\text{AtomicEffect}) \\
ae & \in & \text{AtomicEffect} & = \text{EffectVar} \cup \text{GetEffect} \cup \text{PutEffect} \\
\text{put}(p) & \in & \text{PutEffect} & = \text{RegVar} \\
\text{get}(p) & \in & \text{GetEffect} & = \text{RegVar} \\
\tau & \in & \text{Type} & = \text{Tyvar} \cup \text{BoolType} \cup \text{FunType} \\
 & & \text{SimpleTypeScheme} & = \cup_{n\geq 0}\text{TyVar}^n \times \cup_{m\geq 0}\text{EffectVar}^m \times \text{Type} \\
 & & \text{CompoundTypeScheme} & = \cup_{k\geq 0}\text{RegVar}^k \times \cup_{n\geq 0}\text{TyVar}^n \times \\
 & & & \quad \cup_{m\geq 0}\text{EffectVar}^m \times \text{Type} \\
\sigma & \in & \text{TypeScheme} & = \text{SimpleTypeScheme} \cup \text{CompoundTypeScheme} \\
c & \in & \text{BoolType} & = \{bool\} \\
\mu \xrightarrow{\varepsilon.\varphi} \mu' & \in & \text{FunType} & = \text{TypeAndPlace} \times \text{ArrowEffect} \times \text{TypeAndPlace} \\
\varepsilon.\varphi & \in & \text{ArrowEffect} & = \text{EffectVar} \times \text{Effect} \\
\mu & \in & \text{TypeAndPlace} & = \text{Type} \times \text{Place} \\
TE & \in & \text{TyEnv} & = \text{Var} \xrightarrow{\text{fin}} (\text{TypeScheme} \times \text{Place})
\end{array}
$$

For any semantic object A, $\text{frv}(A)$, $\text{frn}(A)$, $\text{ftv}(A)$, and $\text{fev}(A)$ denote the set of region variables, region names, type variables, and effect variables, respectively, that occur free in A. We define $\text{fvars}(A)$ to be the union $\text{frv}(A) \cup \text{ftv}(A) \cup \text{fev}(A)$.

Effects are sets of atomic effects. There are three kinds of atomic effects: effect variables, put-effects, and get-effects. If $\text{put}(\rho)$ is in the effect inferred for some expression e, e might put a new value in ρ, whereas if $\text{get}(\rho)$ is in its effect, it might access some store value contained in ρ. Effect variables do not correspond to any notion in the target language, but they can be instantiated to put- and get-effects.

The effects on function arrows are paired with an effect variable to form an arrow effect. This effect variable, which we call the *principal* effect variable of the arrow effect, should be viewed as a representative of the effect. By quantifing effect variables, an arrow effect can be instantiated to different arrow effects.

A substitution is a triple (S_r, S_t, S_e), where S_r is a map from region variables to places, S_t is a map from type variables to types, and S_e is a map from effect variables to arrow effects.

Effect substitutions work like this on effects

$$S_e(\varphi) = \varphi \setminus \mathbf{dom}(S_e) \cup \bigcup_{\substack{\varepsilon_i \in \mathbf{dom}(S_e) \cap \varphi \\ \varepsilon_i'.\varphi_i' = S_e(\varepsilon_i)}} (\{\varepsilon_i'\} \cup \varphi_i')$$

Or in others words: whenever $\varepsilon \in \varphi$ and $S_e(\varepsilon) = \varepsilon'.\varphi'$, ε is exchanged by ε', and φ' is added to the effect.

On arrow effects an effect substitution works like this

$$S_e(\varepsilon.\varphi) = \begin{cases} \varepsilon'.S_e(\varphi) \cup \varphi', & \text{if } S_e(\varepsilon) = \varepsilon'.\varphi' \\ \varepsilon.S_e(\varphi), & \text{otherwise} \end{cases}$$

The effect of a triple substitution (S_r, S_t, S_e) is to apply the three substitutions simultaneously. Substitutions that are applied to types and effects are always implicitly extended to be the identity outside their domain.

Note that there are two kinds of type schemes. Compound type schemes, which are polymorfic in both types, effects, and regions, abstract region closures, whereas simple type schemes, which are only polymorfic in types and effects, abstract other values. We demand that all the quantified variables in a type scheme are distinct, i.e. that the same variable is never quantified twice. We regard type schemes that can be obtained from each other by renaming bound variables as equal. When a substitution S is applied to a type scheme σ, we pick $\vec{\rho}$, $\vec{\alpha}$, and $\vec{\varepsilon}$ such that

$$\sigma = \forall \vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau$$

and

$$\mathrm{frv}(\vec{\rho}) \cup \mathrm{ftv}(\vec{\alpha}) \cup \mathrm{fev}(\vec{\varepsilon}) \cap (\mathbf{dom}(S) \cup \mathrm{fvars}(\mathbf{rng}(S))) = \{\}$$

and then

$$S(\sigma) = \forall \vec{\rho}\vec{\alpha}\vec{\varepsilon}.S(\tau)$$

We say that a type scheme $\sigma = \forall \vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau'$ is instantiated to the type τ via *the instantiation vector* $\mathcal{I} = [\vec{p}, \vec{\tau}, \vec{\varepsilon.\psi}]$, written $\sigma \succ \tau$ via $\mathcal{I}$, if $\tau = S(\tau')$, where $S = ([\vec{p}/\vec{\rho}], [\vec{\tau}/\vec{\alpha}], [\vec{\varepsilon.\psi}/\vec{\varepsilon}])$. In this case, we will call S *the instantiation substitution*, and use the alternative notation $\sigma \succ \tau$ via S or just $\sigma \succ \tau$.

Now let us look at the translation rules. There is one rule for every construct in the target language.

$$\frac{}{TE \vdash \texttt{true} \Rightarrow \texttt{true at } p : bool, \mathrm{put}(p)} \qquad \frac{}{TE \vdash \texttt{false} \Rightarrow \texttt{false at } p : bool, \mathrm{put}(p)}$$

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

This rule contains nothing surprising, except, perhaps, for the fact that it has no effect to access a variable. Here one must bear in mind that get-effects stand for reads in the store. The target program x gets a value from the environment, an address, without inspecting what is contained at the memory location it refers to.

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

A mentioned above, compound type schemes abstract region closures in the target language. So when $TE(f)$ is a compound type scheme, we must generate a region-closure instantiation expression. Such an expression is executed by consulting the region closure and creating a new plain closure. Hence the inferred effect.

$$\frac{TE + \{x \mapsto \mu_1\} \vdash e \Rightarrow e' : \mu_2, \varphi \quad \varphi \subseteq \varphi'}{TE \vdash \lambda x.e \Rightarrow \lambda x.e' \texttt{ at } p : \left(\mu_1 \xrightarrow{\varepsilon.\varphi} \mu_2, p\right), \{\mathrm{put}(p)\}}$$

μ_1 is here viewed as the simple type scheme which has no quantifications. Note that the effect inferred for e moves onto the arrow in the function type, whereas the effect of the abstraction is only $\mathrm{put}(p)$. This is because $\lambda x.e$ `at` p is executed by creating a closure and putting it at p. The effect of applying the closure at some later time is φ, the *latent* effect of the function. We allow the function to have a larger effect than the one inferred for its expression, because we want to be able to type expressions like

$$\texttt{if } \cdots \texttt{ then } \lambda x.x \texttt{ at } \rho \texttt{ else } \lambda x.(\texttt{true at } \rho_1) \texttt{ at } \rho$$

where the two functions must have the same type, though the expression `true at` ρ_1 has effect $\mathrm{put}(\rho_1)$, and the expression x has the empty effect.

$$\frac{TE \vdash e_1 \Rightarrow e_1' : \left(\mu' \xrightarrow{\varepsilon.\varphi} \mu, p\right), \varphi_1 \quad TE \vdash e_2 \Rightarrow e_2' : \mu', \varphi_2}{TE \vdash e_1\ e_2 \Rightarrow e_1'\ e_2' : \mu, \varphi_1 \cup \varphi_2 \cup \varphi \cup \{\varepsilon, \mathrm{get}(p)\}}$$

The effect of the application is the union of the effects of e_1 and e_2 plus the latent effect of the function and a get-effect from accessing the closure in the store. Note that the principal effect variable ε is also part of

the function's latent effect, because the principal effect variable of an effect must always occur together with it as its representative.

$$\frac{TE \vdash e_1 \Rightarrow e_1' : (\tau, \rho), \varphi_1 \quad TE + \{x \mapsto (\sigma, p)\} \vdash e_2 \Rightarrow e_2' : \mu, \varphi_2 \quad \sigma = \mathrm{TyEffGen}(TE, \varphi_1)(\tau)}{TE \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Rightarrow \texttt{let}\ x = e_1'\ \texttt{in}\ e_2' : \mu, \varphi_1 \cup \varphi_2}$$

This rules introduces polymorfism of types and effects.

Let A be a semantic or assembly of semantic objects, let τ be a type, let $\{\alpha_1, \ldots, \alpha_n\} = \mathrm{ftv}(\tau) \setminus \mathrm{ftv}(A)$, let $\{\rho_1, \ldots, \rho_k\} = \mathrm{frv}(\tau) \setminus \mathrm{frv}(A)$, and let $\{\varepsilon_1, \ldots, \varepsilon_m\} = \mathrm{fev}(\tau) \setminus \mathrm{fev}(A)$. The operations for closing τ w.r.t. A are the following

$$\begin{aligned}
\mathrm{RegTyEffGen}(A)(\tau) &= \forall \rho_1 \cdots \rho_k \alpha_1 \cdots \alpha_n \varepsilon_1 \cdots \varepsilon_m.\tau \\
\mathrm{TyEffGen}(A)(\tau) &= \alpha_1 \cdots \alpha_n \varepsilon_1 \cdots \varepsilon_m.\tau \\
\mathrm{RegEffGen}(A)(\tau) &= \forall \rho_1 \cdots \rho_k \varepsilon_1 \cdots \varepsilon_m.\tau
\end{aligned}$$

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

The letrec-expression is used for introducing region closures in the target language, which in the translation are abstracted by compound type schemes. Note that f is only polymorfic in regions and effects within its own body. Had the rule allowed type-polymorfic recursion, tybability would have been undecidable [Hen93].

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

Here *the observable part of* φ *w.r.t.* A, written $\mathrm{Observe}(A)(\varphi)$, where A is some semantic object, and φ an effect, is defined in the following way

$$\begin{aligned}
\mathrm{Observe}(A)(\varphi) = &\{\mathrm{put}(p) \in \varphi \mid p \in \mathrm{frv}(A) \cup \mathrm{frn}(A)\} \\
&\cup \{\mathrm{get}(p) \in \varphi \mid p \in \mathrm{frv}(A) \cup \mathrm{frn}(A)\} \\
&\cup \{\varepsilon \in \varphi \mid \varepsilon \in \mathrm{fev}(A)\}
\end{aligned}$$

This rule is the very heart of region inference. This is where block declarations are introduced. It turns out that this rule is not only sound, it also makes spectacular memory utilization possible, as we shall see in later examples.

$$\frac{TE \vdash e_1 \Rightarrow e_1' : (bool, p, \varphi_1) \quad TE \vdash e_2 \Rightarrow e_2' : \mu, \varphi_2 \quad TE \vdash e_3 \Rightarrow e_3' : \mu, \varphi_3}{TE \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Rightarrow \texttt{if}\ e_1'\ \texttt{then}\ e_2'\ \texttt{else}\ e_3' : \mu, \varphi_1 \cup \varphi_2 \cup \varphi_3 \cup \{\mathrm{get}(p)\}}$$

The atomic effect $\mathrm{get}(p)$ is added to the effect, because the execution of the if-then-else-expression involves reading in the store whether the memory cell to which the value generated by e_1 refers contains `true` or `false`.

As the choice of branch is determined by the value of the test, and as values are only indirectly present in the translation as types, we have no means of knowing which branch is chosen at run-time. Therefore we add the union of the effects of e_2' and e_3' to the effect.

A fundamental requirement is that region inference be safe: we would like a target expression generated by the translation to evaluate to a value that corresponds to the value to which the original source expression evaluates. That this always holds is far from obvious.

The formulation of safeness as a concrete theorem is naturally complicated, as we have to relate two languages, three kinds of environments, two kinds of values, a store, as well as effects, places, and types. The reader can consult [TT93a] for a formulation of the theorem and a detailed proof of it.

3 Multiplicity inference

In this chapter we deal with the multiplicity inference discipline, which translates an annotated region expression to a region expression with sizes.

First we describe the syntax of the input and output expressions. We then introduce effect consistency, a concept which is used for identifying the subset of possible input expressions which we will consider in our analysis.

We then define a semantics for translating input expressions to output expressions.

3.1 Input and output of the translation

The input of the analysis, annotated region expressions, has the following form

$$\begin{array}{rcl} te & ::= & \texttt{true at } p \mid \texttt{false at } p \mid x[\vec{\tau},\varepsilon.\vec{\varphi}] \mid f[\vec{p},\vec{\tau},\varepsilon.\vec{\varphi}] \texttt{ at } p \mid \lambda\!:\!\mu_1 \xrightarrow{\varepsilon.\varphi} \mu_2\ x.te \texttt{ at } p \mid \\ & & te_1\ te_2 : p,\varepsilon \mid \texttt{let } x\!:\!\forall\vec{\varepsilon}\vec{\alpha}.\tau = te_1 \texttt{ in } te_2 \mid \texttt{letrec } f\!:\!\forall\vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau \texttt{ at } p = te_1 \texttt{ in } te_2 \mid \\ & & \texttt{letregion } \varphi \texttt{ in } te \mid \texttt{if } te_1 : p \texttt{ then } te_2 \texttt{ else } te_3 \end{array}$$

Together with some matching type environment, a annotated region expression should be viewed as the inference tree for a typing in the translation semantics of region inference. It is intended that

- the places, types, and arrow effects a variable is annotated with are the ones which are substituted for the region-, type, and effect variables of the type scheme of the variable during translation.
- the function type at a λ should be the type of the lambda abstaction.
- the p at an application should be the place of the applied function, and the ε should be the principal effect variable on its arrow.
- The p at the first subexpression of the if-then-else statement is the place of the boolean which is tested.
- the type schemes at the let- and letrec-cases should be the ones inferred for the variable.
- the φ in the letregion-case should be the whole effect that is 'observed away' when the letregion-expression is introduced.

We call an input expression which satisfies these requirements *well formed*.

Typed region expressions are the target expressions of region inference annotated with extra type information. We define the following map from annotated region expressions to plain region expressions

$$\begin{array}{rl} \overline{\texttt{true at } p} & = \texttt{true at } p \\ \overline{\texttt{false at } p} & = \texttt{false at } p \\ \overline{x[\vec{\tau},\varepsilon.\vec{\varphi}]} & = x \\ \overline{f[\vec{p},\vec{\tau},\varepsilon.\vec{\varphi}] \texttt{ at } p} & = f[\vec{p}] \texttt{ at } p \\ \overline{\lambda\!:\!\mu_1 \xrightarrow{\varepsilon.\varphi} \mu_2\ x.te \texttt{ at } p} & = \lambda x.\overline{te} \texttt{ at } p \\ \overline{te_1\ te_2 : p,\varepsilon} & = \overline{te_1}\ \overline{te_2} \\ \overline{\texttt{letrec } f\!:\!\forall\vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau \texttt{ at } p = te_1 \texttt{ in } te_2} & = \texttt{letrec } f[\vec{\rho}] \texttt{ at } p = \overline{te_1} \texttt{ in } \overline{te_2} \\ \overline{\texttt{let } x\!:\!\forall\vec{\varepsilon}\vec{\alpha}.\tau = te_1 \texttt{ in } te_2} & = \texttt{let } x = \overline{te_1} \texttt{ in } \overline{te_2} \\ \overline{\texttt{letregion } \varphi \texttt{ in } te} & = \texttt{letregion } \mathrm{frv}(\varphi) \texttt{ in } \overline{te_2} \\ \overline{\texttt{if } te_1 : p \texttt{ then } te_2 \texttt{ else } te_3} & = \texttt{if } \overline{te_1} \texttt{ then } \overline{te_2} \texttt{ else } \overline{te_3} \end{array}$$

In the previous chapter we defined a map $\overline{\ }$ from target expressions to source expressions. We now define $\overline{\overline{\ }}$ to be the composition of the two functions, i.e. a map from annotated region expressions to source expressions.

The output of our translation is an expression that conforms to this grammar

$$\begin{array}{rcl} e & ::= & \texttt{true at } p \mid \texttt{false at } p \mid x \mid f[\vec{p}] \texttt{ at } p \mid \lambda x.e \texttt{ at } p \mid \\ & & e\ e' \mid \texttt{let } x = e \texttt{ in } e' \mid \texttt{letrec } f\,[\vec{\rho}] \texttt{ at } p = e \texttt{ in } e' \mid \\ & & \texttt{letregion } \rho_1 : n_1 \cdots \rho_k : n_k \texttt{ in } e \mid \texttt{if } e \texttt{ then } e' \texttt{ else } e'' \end{array}$$

Note that the output expressions are identical to the target expressions of region inference, except for the letregion-expression where all regions are supplied with a multiplicity. This multiplicity is either a natural number or ∞.

The dynamic semantics of the output expressions differs from the semantics of the plain region expressions, in that the cardinality of the domain of a region is limited by the number in its declaration. It is therefore a fundamental requirement that our analysis generates expressions whose regions are sufficiently large. However, we shall not prove our translation rules safe, but regard them as intuitively correct, and we will therefore not bother to introduce an elaborate dynamic semantics for the output expressions.

If e is an output expression, we let $\underline{e}$ mean the target expressions obtained from e by removing the multiplicities of the letregion declarations.

3.2 Effect consistency

In this section we shall pose some semantic restrictions on the typed region expressions. First of all it makes no sense to consider expressions whose annotations do not satisfy the requirements stated in the previous section. We only want to consider input expressions which represent an inference tree in region inference. But we will not accept any inference tree from region inference, but only ones which are *effect consistent*, a notion we will define below.

Before introducing the exact definition of effect consistency let us study the expression $(\lambda x.1, \lambda x.x)$ (it is no problem to add pairs to region inference). Under the empty environment, several different typings could be inferred from this expression region inference, for example

$$\begin{array}{rcl} \{\} \vdash (\lambda x.1, \lambda x.x) & \Rightarrow & (\lambda x.(1 \texttt{ at } \rho_1) \texttt{ at } \rho_2, \lambda x.x \texttt{ at } \rho_3) \texttt{ at } \rho_4 : \\ & & (((\alpha, \rho_5) \xrightarrow{\varepsilon_1.\{\mathrm{put}(\rho_1)\}} (int, \rho_1), \rho_2) * ((\alpha, \rho_6) \xrightarrow{\varepsilon_1.\{\}} (\alpha, \rho_7), \rho_3), \rho_4), \\ & & \{\mathrm{put}(\rho_2), \mathrm{put}(\rho_3), \mathrm{put}(\rho_4)\} \end{array}$$

or

$$\begin{array}{rcl} \{\} \vdash (\lambda x.1, \lambda x.x) & \Rightarrow & (\lambda x.(1 \texttt{ at } \rho_1) \texttt{ at } \rho_2, \lambda x.x \texttt{ at } \rho_3) \texttt{ at } \rho_4 : \\ & & (((\alpha, \rho_5) \xrightarrow{\varepsilon_1.\{\mathrm{put}(\rho_1)\}} (int, \rho_1), \rho_2) * ((\alpha, \rho_6) \xrightarrow{\varepsilon_2.\{\}} (\alpha, \rho_7), \rho_3), \rho_4), \\ & & \{\mathrm{put}(\rho_2), \mathrm{put}(\rho_3), \mathrm{put}(\rho_4)\} \end{array}$$

The only difference between the two examples is the effect variable on the arrow of the identity function. In the first example it is ε_1, the same variable as on the constant function, in the second it is ε_2, its own variable. The type of the first typing is not effect-consistent, whereas the type of the second is.

In multiplicity inference, we only deal with typings in which the effects variables never occur with different effects at different places in the same expression. The reason is that the inference algorithm we shall later introduce is based on having a table of all arrow effects in the inference tree, and in this table there must only be one effect for every principal effect variable. But even disregarding this need of our inference algorithm, the demand that an effect variable always occur together with the same effect seems to be quite reasonable and should certainly not limit the number of source programs covered by the analysis.

If we look again at the typings above, the second type is in a way the most general one, since it could be turned into the first one by the effect substitution $\{\varepsilon_2 \mapsto \varepsilon_1.\{\}\}$, whereas the opposite is not the case. The typings returned by $\mathcal{RE}$ [TT93a] appear to be effect consistent.

We will now first introduce the exact definition of effect consistency of a set of arrow effects. In the following we let Φ range over sets of arrow effects.

Definition 3.2. *Let Φ be a set of arrow effects. We say that Φ is* effect-consistent, *if*

1) whenever $\varepsilon.\varphi \in \Phi$ and $\varepsilon'.\varphi' \in \Phi$, if $\varepsilon = \varepsilon'$ then $\varphi = \varphi'$
2) whenever $\varepsilon.\varphi \in \Phi$ and $\varepsilon'.\varphi' \in \Phi$, if $\varepsilon \in \varphi'$ then $\varphi \subseteq \varphi'$
3) whenever $\varepsilon.\varphi \in \Phi$ and $\varepsilon' \in \varphi$, then there exists φ' such that $\varepsilon'.\varphi' \in \Phi$

If Φ is effect-consistent, and $\varepsilon \in \mathrm{fev}(\Phi)$, then by $\Phi[\varepsilon]$ we mean the effect φ for which it holds that $\varepsilon.\varphi \in \Phi$. By $\hat{\Phi}$ we mean the map

$$\hat{\Phi}(\varepsilon) = \Phi[\varepsilon], \qquad \textit{for all } \varepsilon \in \mathrm{fpev}(\Phi)$$

The first requirement in the definition ensures that a principal effect variable only occurs with one effect. The second requirement extends the first requirement to cover non-principal occurrences of an effect variable: its own effect must always be included in effects in which the effect variable occurs. This last requirement rules out the existence of orphan effect variables in an effect-consistent set of arrow effects: all effect variables must be associated with an effect. It is equivalent to demanding that $\mathrm{fev}(\Phi) = \mathrm{fpev}(\Phi)$.

Example 3.3. Define

$$\begin{aligned}
\Phi_1 &= \{\ \varepsilon_1.\{\mathrm{put}(\rho_1)\}, \varepsilon_1.\{\mathrm{put}(\rho_2)\}\ \} \\
\Phi_2 &= \{\ \varepsilon_1.\{\mathrm{put}(\rho_1)\}, \varepsilon_2.\{\mathrm{put}(\rho_2)\}\ \} \\
\Phi_3 &= \{\ \varepsilon_1.\{\mathrm{put}(\rho_1)\}, \varepsilon_2.\{\mathrm{put}(\rho_2), \varepsilon_1\}\ \} \\
\Phi_4 &= \{\ \varepsilon_1.\{\mathrm{put}(\rho_1)\}, \varepsilon_2.\{\mathrm{put}(\rho_1), \mathrm{put}(\rho_2), \varepsilon_1\}\ \} \\
\Phi_5 &= \{\ \varepsilon_2.\{\mathrm{put}(\rho_1), \mathrm{put}(\rho_2), \varepsilon_1\}\ \}
\end{aligned}$$

Φ_2 and Φ_4 are each effect-consistent, Φ_1, Φ_3, and Φ_5 are not.

We now extend the notion of effect consistency so that it covers types, type schemes, TypeAndPlace, and effects:

Definition 3.4. *We define the four relations $\Phi \models \tau$, $\Phi \models \mu$, $\Phi \models \sigma$, and $\Phi \models \varphi$ as the smallest relations that satisfy the following inference rules follows:*

$$\frac{\Phi \text{ is effect-consistent}}{\Phi \models \mathit{bool}} \qquad \frac{\Phi \text{ is effect-consistent}}{\Phi \models \alpha} \qquad \frac{\Phi \models \tau}{\Phi \models (\tau, \rho)} \qquad \frac{\Phi \models \sigma}{\Phi \models (\sigma, p)}$$

$$\frac{\Phi \models \mu' \quad \Phi \models \mu \quad \hat{\Phi}(\varepsilon) = \varphi}{\Phi \models \mu \xrightarrow{\varepsilon.\varphi} \mu'} \qquad \frac{\begin{array}{c}\Phi \text{ is effect-consistent} \\ \sigma = \forall \vec{\rho}\vec{\varepsilon}\vec{\alpha}.\tau \quad \Phi \cup \Phi' \models \tau \\ \mathrm{fev}(\sigma) \subseteq \mathrm{fev}(\Phi) \\ \mathrm{fev}(\vec{\varepsilon}) \cap \mathrm{fev}(\Psi) = \emptyset \\ \mathrm{frv}(\vec{\rho}) \cap \mathrm{frv}(\Psi) = \emptyset\end{array}}{\Phi \models \sigma} \qquad \frac{\begin{array}{c}\varepsilon \notin \mathrm{fev}(\Phi) \cup \varphi \\ \{\varepsilon.\varphi\} \cup \Phi \text{ is effect-consistent}\end{array}}{\Phi \models \varphi}$$

$\Phi \models \tau$ *should be read* τ is effect-consistent under Φ. *In the same way we read* $\Phi \models \mu$, $\Phi \models \sigma$, *and* $\Phi \models \varphi$.

Let TE be a type environment. We say that TE is effect-consistent under Φ, *written* $\Phi \models TE$, *if for all* $x \in \mathbf{dom}(TE)$ *it holds that* $\Phi \models TE(x)$.

Let A be a set of types, typeschemes, effects, and type environments. We say that $\Phi \models A$, *if for all* $a \in A$ *it holds that* $\Phi \models a$.

The rule for type schemes says that all the free effect and region variables must be contained in Φ, and if a version of the type scheme is picked so that the bound effect and region variables are distinct from the ones in Φ then it must be possible to extend Φ with an addendum Φ' so that the type of the type scheme is effect-consistent under $\Phi \cup \Phi'$.

The premises for concluding $\Phi \models \varphi$ are equivalent to demanding that whenever $\varepsilon' \in \mathrm{fev}(\varphi)$ then there exists φ' such that $\varepsilon'.\varphi' \in \Phi$ and $\varphi' \subseteq \varphi$.

Example 3.5. In this example we refer to the sets of arrow effects from example 3.3. Define

$$\varphi_1 = \{\varepsilon_1, \mathrm{put}(\rho_2)\}$$
$$\varphi_2 = \{\varepsilon_1, \mathrm{put}(\rho_1)\}$$

Here φ_2 is effect-consistent under Φ_2, whereas φ_1 is not, since $\hat{\Phi}_2(\varepsilon_1) = \{\mathrm{put}(\rho_1)\} \not\subseteq \varphi_1$.

Define

$$\sigma_1 = \forall \varepsilon_3 \varepsilon_2.\mu \xrightarrow{\varepsilon_3.\{\mathrm{put}(\rho_1), \varepsilon_1, \varepsilon_2\}} \mu$$
$$\sigma_2 = \forall \varepsilon_3 \varepsilon_1.\mu \xrightarrow{\varepsilon_3.\{\mathrm{put}(\rho_1), \varepsilon_1, \varepsilon_2\}} \mu$$

We use μ to signify any TypeAndPlace without any arrow effects. We have that σ_1 is effect-consistent under Φ_4. For

$$\sigma_1 \stackrel{\alpha}{=} \forall \varepsilon_3 \varepsilon_4.\mu \xrightarrow{\varepsilon_3.\{\mathrm{put}(\rho_1), \varepsilon_1, \varepsilon_4\}} \mu$$

$$\varepsilon_3, \varepsilon_4 \cap \mathrm{fev}(\Phi_4) = \{\} \qquad \mathrm{fev}(\forall \varepsilon_3.\varepsilon_4.\mu \xrightarrow{\varepsilon_3.\{\mathrm{put}(\rho_1), \varepsilon_1, \varepsilon_4\}} \mu) \subseteq \mathbf{dom}(\hat{\Phi}_4)$$

and

$$\Phi_4 \cup \{\ \varepsilon_3.\{\mathrm{put}(\rho_1), \varepsilon_1, \varepsilon_4\}, \varepsilon_4.\emptyset\ \} \models \mu \xrightarrow{\varepsilon_3.\{\mathrm{put}(\rho_1), \varepsilon_1, \varepsilon_4\}} \mu$$

σ_2 is not effect-consistent under Φ_4. After renaming the bound variables so that they are distinct from the domain of $\hat{\Phi}_4$, σ_2 could look like this

$$\forall \varepsilon_3 \varepsilon_4.\mu \xrightarrow{\varepsilon_3.\{\mathrm{put}(\rho_1), \varepsilon_4, \varepsilon_2\}} \mu$$

But here ε_2 occurs free in an effect that does not contain ε_1. And since $\varepsilon_1 \in \hat{\Phi}_4(\varepsilon_2)$, there is no consistent extension of Φ_4, under which $\mu \xrightarrow{\varepsilon_3.\{\mathrm{put}(\rho_1), \varepsilon_4, \varepsilon_2\}} \mu$ would be consistent.

Here are some lemmas about effect consistency

Lemma 3.6. *Suppose* $\Phi \models \varphi_1$ *and* $\Phi \models \varphi_2$ *then*

$$\Phi \models \varphi_1 \cup \varphi_2$$

Proof.
Let $\varphi = \varphi_1 \cup \varphi_2$. Pick an ε such that $\varepsilon \notin \mathbf{dom}(\hat{\Phi}) \cup \varphi$. We then must show that $\Phi \cup \{\varepsilon.\varphi\}$ is effect-consistent. Since ε does not occur anywhere in Φ, we only have to show that for all $\varepsilon'.\varphi' \in \Phi$ it holds that if $\varepsilon' \in \varphi$ then $\varphi' \subseteq \varphi$. If $\varepsilon' \in \varphi$ then ε' must occur in either φ_1 or φ_2. Suppose it occurs in φ_1 (The case for φ_2 is similar). Then by $\Phi \models \varphi_1$ we have that

$$\varphi' \subseteq \varphi_1 \subseteq \varphi$$

■

Lemma 3.7. *Suppose* $\Phi \models TE, \mu, \varphi$. *Then*

$$\Phi \models \text{Observe}(TE, \mu)(\varphi)$$

Proof.
Assume

$$\Phi \models TE, \mu, \varphi \tag{0}$$

We know that

$$\text{Observe}(TE, \mu)(\varphi) \subseteq \varphi \tag{1}$$

So the only reason why it could not be the case that $\Phi \models \text{Observe}(TE, \mu)(\varphi)$ would be if there was some $\varepsilon' \in \text{fev}(\text{Observe}(TE, \mu)(\varphi))$ such that $\hat{\Phi}(\varepsilon') \not\subseteq \text{Observe}(TE, \mu)(\varphi)$. But since we know, by (0), that $\hat{\Phi}(\varepsilon) \subseteq \varphi$, the atomic effects lacking in $\text{Observe}(TE, \mu)(\varphi)$ must have been observed away. This, again, amounts to saying that ε occurs free somewhere in TE or μ without these atomic effects, which contradicts (0).

■

Lemma 3.8. *Let* Φ *be a set of arrow effects,* τ *a type, and* S_t *a type substitution. We have that*

$$\Phi \models S_t(\tau) \quad \Longrightarrow \quad \Phi \models \tau$$

Proof.
The proof is by induction on the structure of τ. The only case where τ can differ from $S_t(\tau)$ is when τ is an effect variable and $S_t(\tau)$ is some other type. And in this case effect consistency holds vacously.

■

We now extend the notion of effect consistency to cover input expressions. We do this by formulating an algorithm which checks the well-formedness of the annotations and makes sure that all arrow effects in the expression are contained in an effect consistent set of arrow effects. It is not meant as a practical program that should ever be run on the input expressions, but as a detailed specification of what we expect of input expressions.

$\text{T}(\Phi, TE, \texttt{true at } p) \Longrightarrow$
  if Φ is effect-consistent
  then $((bool, p), \{\text{put}(p)\})$
  else fail
$\text{T}(\Phi, TE, \texttt{false at } p) \Longrightarrow$
  if Φ is effect-consistent
  then $((bool, p), \{\text{put}(p)\})$
  else fail
$\text{T}(\Phi, TE, x[\vec{\tau}, \varepsilon.\vec{\varphi}]) \Longrightarrow$
  if $(\forall \vec{\alpha}\vec{\varepsilon}.\tau, p) = TE(x)$
    $\wedge \ \mid \vec{\tau} \mid = \mid \vec{\alpha} \mid \ \wedge \ \mid \varepsilon.\vec{\varphi} \mid = \mid \vec{\varepsilon} \mid$
    $\wedge \ \text{aref}(\varepsilon.\vec{\varphi}) \subseteq \Phi \ \wedge \ \Phi \models (\{\}, \vec{\tau}/\vec{\alpha}, \varepsilon.\vec{\varphi}/\vec{\varepsilon})(\tau)$
  then $((\{\}, \vec{\tau}/\vec{\alpha}, \varepsilon.\vec{\varphi}/\vec{\varepsilon})(\tau), p), \{\})$
  else fail

$$\mathrm{T}(\Phi, TE, f[\vec{p},\vec{\tau},\vec{\varepsilon.\varphi}] \texttt{ at } p) \Longrightarrow$$
if $(\forall\vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau, p') = TE(f)$
$\wedge \ |\vec{p}| = |\vec{\rho}| \ \wedge \ |\vec{\tau}| = |\vec{\alpha}| \ \wedge \ |\vec{\varepsilon.\varphi}| = |\vec{\varepsilon}|$
$\wedge \ \mathrm{aref}(\vec{\varepsilon.\varphi}) \subseteq \Phi \ \wedge \ \Phi \models (\vec{p}/\vec{\rho},\vec{\tau}/\vec{\alpha},\vec{\varepsilon.\varphi}/\vec{\varepsilon})(\tau)$
then $(((\vec{p}/\vec{\rho},\vec{\tau}/\vec{\alpha},\vec{\varepsilon.\varphi}/\vec{\varepsilon})(\tau), p), \{\mathrm{put}(p), \mathrm{get}(p')\})$
else fail

$$\mathrm{T}(\Phi, TE, \lambda{:}\mu_1 \xrightarrow{\varepsilon.\varphi} \mu_2 \ x.te \texttt{ at } p) \Longrightarrow$$
if $\Phi \models \mu_1 \ \wedge \ \varepsilon.\varphi \in \Phi \ \wedge \ \mathrm{T}(\Phi, TE + \{x \mapsto \mu_1\}, te) = (\mu_2, \varphi_1) \ \wedge \ \varphi_1 \subseteq \varphi$
then $\left(\left(\mu_1 \xrightarrow{\varepsilon.\varphi} \mu_2, p\right), \{\mathrm{put}(p)\}\right)$
else fail

$$\mathrm{T}(\Phi, TE, te_1\ te_2 : p, \varepsilon) \Longrightarrow$$
if $\mathrm{T}(\Phi, TE, te_1) = \left(\left(\mu_2 \xrightarrow{\varepsilon.\varphi} \mu_1, p\right), \varphi_1\right) \ \wedge \ \mathrm{T}(\Phi, TE, te_2) = (\mu_2, \varphi_2)$
then $(\mu_1, \varphi \cup \varphi_1 \cup \varphi_2 \cup \{\mathrm{get}(p)\} \cup \{\varepsilon\})$
else fail

$$\mathrm{T}(\Phi, TE, \texttt{let } x{:}\forall\vec{\varepsilon}\vec{\alpha}.\tau = te_1 \texttt{ in } te_2) \Longrightarrow$$
if $\mathrm{T}(\Phi, TE, te_1) = ((\tau, p), \varphi_1) \ \wedge \ \forall\vec{\varepsilon}\vec{\alpha}.\tau = \mathrm{TyEffGen}(TE, \varphi_1)(\tau)$
$\wedge \ \mathrm{T}(\Phi, TE + \{x \mapsto (\forall\vec{\varepsilon}\vec{\alpha}.\tau, p)\}, te_2) = (\mu, \varphi_2)$
then $(\mu, \varphi_1 \cup \varphi_2)$
else fail

$$\mathrm{T}(\Phi, TE, \texttt{letrec } f{:}\forall\vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau \texttt{ at } p = te_1 \texttt{ in } te_2) \Longrightarrow$$
if $\Phi \models \forall\vec{\rho}\vec{\varepsilon}\vec{\alpha}.\tau$
$\wedge \ \mathrm{T}(\Phi, TE + \{f \mapsto (\forall\vec{\rho}\vec{\varepsilon}.\tau, p)\}, te_1) = ((\tau, p), \varphi_1)$
$\wedge \ \forall\vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau = \mathrm{RegTyEffGen}(TE, \varphi_1)(\tau)$
$\wedge \ \mathrm{T}(\Phi, TE + \{x \mapsto (\forall\vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau, p)\}, te_2) = (\mu_2, \varphi_2)$
then $(\mu_2, \varphi_1 \cup \varphi_2)$
else fail

$$\mathrm{T}(\Phi, TE, \texttt{letregion } \varphi \texttt{ in } te) \Longrightarrow$$
if $\mathrm{T}(\Phi, TE, te) = (\mu, \varphi') \ \wedge \ \varphi'' = \mathrm{Observe}(TE, \mu)(\varphi') \ \wedge \ \varphi = \varphi' \setminus \varphi''$
then (μ, φ'')
else fail

$$\mathrm{T}(\Phi, TE, \texttt{if } te_1 : p \texttt{ then } te_2 \texttt{ else } te_3) \Longrightarrow$$
if $\mathrm{T}(\Phi, TE, te_1) = ((bool, p), \varphi_1) \ \wedge \ \mathrm{T}(\Phi, TE, te_2) = (\mu, \varphi_2) \ \wedge \ \mathrm{T}(\Phi, TE, te_3) = (\mu, \varphi_3)$
then $(\mu, \varphi_1 \cup \varphi_2 \cup \varphi_3 \cup \{\mathrm{get}(p)\})$
else fail

The argument Φ is a set of arrow effects containing at least all the arrows effects that occur somewhere in the inference tree. At every node in the inference tree the algorithm checks that new arrow effects are consistent with the ones in Φ, i.e. if they are consistent with all the other arrow effects in the inference tree. Note that Φ also contains all arrow effects from let and letrec subexpressions, although these arrow effects are quantified in the type schemes of the subexpressions. So Φ is a truly global reference of all syntactical occurrences of arrow effects.

If $\mathrm{T}(\Phi, TE, te) = (\mu, \varphi)$, we say that te is *well formed under* TE.

RegTyEffGen, TyEffGen, and RegEffGen are defined in the previous chapter, and they quantify all free effect variables, not only the ones that occur in principal positions. As pointed out in [TT93a], it may be necessary to redefine these operations, so they allow a bit less polymorphism. This is fine with multiplicity inference, as long as the closure operations generate effect consistent type schemes. This property is what is expressed by the following lemma, which we prove for the traditional definitions of the closure operations:

Lemma 3.9. *If* $\Phi \models \{\tau, TE, \varphi\}$ *then*

$$\Phi \models \mathrm{RegTyEffGen}(TE, \varphi)(\tau)$$

$$\Phi \models \mathrm{RegEffGen}(TE, \varphi)(\tau)$$

and

$$\Phi \models \mathrm{TyEffGen}(TE, \varphi)(\tau)$$

Proof.
se appendix A

■

Let us now state some properties of T and input expressions:

Theorem 3.10. *If* $T(\Phi, TE, te) = (\mu, \varphi)$ *then*

$$TE \vdash \overline{\overline{te}} \Rightarrow \overline{te} : \mu, \varphi$$

Proof.
By structural induction on te.

■

Lemma 3.11. *If* $T(\Phi, TE, te) = (\mu, \varphi)$ *then* $\Phi \models \mu, \varphi$.

Proof.
Straightforward structural induction on te, using lemma 3.6, lemma 3.7, and lemma 3.9.

■

3.3 Translation semantics

In this section we will first introduce the semantic objects of multiplicity inference and then give a set of inference rules that relate the input expressions to the output expressions.

3.3.1 Semantic objects

The aim of this analysis is to infer sizes for regions. And as regions grow every time a value is put in it, the problem amounts to inferring a limit to the number of times an expression can write to a region. In the translation semantics of region inference it is inferred whether or not an expression writes to a region. So what we need is only a refinement of the already existing effect inference.

It is obvious that the put-effects must somehow be supplied with multiplicity information. And in that case we must also attach multiplicities to effect variables, because they can be instantiated to put-effects. The get-effects are in this connection irrelevant, since it has no influence on the size of a region how many times an expression reads in it. Quantifications and insertion of letregion-declarations have already been completed at the time our input was created, so we will not have to use the get-effects for this either. But as they do no harm, we shall infer read multiplicities as a bonus without using them for anything.

This leads us to the following semantic objects in addition to the ones from the translation semantics of region inference

$$\begin{array}{llcl} n & \in & \text{Multiplicity} & = & \{0,1,\ldots,K\} \cup \{\infty\} \\ \psi & \in & \text{MulEff} & = & \text{AtomicEffect} \xrightarrow{\text{fin}} \text{Multiplicity} \\ \varepsilon.\psi & \in & \text{MulArrowEff} & = & \text{EffVar} \times \text{MulEff} \end{array}$$

Multiplicity arrow effects can occur over arrows in types. So there are two kinds of types, type schemes, type environments, and substitutions, some with plain arrow effects and some with multiplicity arrow effects. There will be no need for objects with both kinds of arrow effects at once.

Note that Multiplicity is a finite set that consists of the value ∞ and some initial segment of the natural numbers $\{0,1,\ldots,K\}$. So the semantics is parametrised by K.

On effects we define the following operations

$$\begin{array}{rcll} (\varphi_1 \oplus \varphi_2)\,(ae) & = & \varphi_1^+(ae) \oplus \varphi_2^+(ae) & \text{for all } ae \in \mathbf{dom}(\varphi_1) \cup \mathbf{dom}(\varphi_2) \\ (\varphi_1 \ominus \varphi_2)(ae) & = & \varphi_1^+(ae) \ominus \varphi_2^+(ae) & \text{for all } ae \in \mathbf{dom}(\varphi_1) \cup \mathbf{dom}(\varphi_2) \\ (\varphi \otimes n)\,(ae) & = & \varphi(ae) \otimes n & \text{for all } ae \in \mathbf{dom}(\varphi) \\ \lceil \varphi_1, \varphi_2 \rceil\,(ae) & = & \text{Max}(\varphi_1^+(ae), \varphi_2^+(ae)) & \text{for all } ae \in \mathbf{dom}(\varphi_1) \cup \mathbf{dom}(\varphi_2) \end{array}$$

ψ^+ is the map which is equal to ψ on ψ's domain and 0 elsewhere. The operators $\oplus$, $\ominus$, and $\otimes$ on Multiplicity are defined as follows:

$$\begin{array}{rcll} \infty \ominus n & = & \infty & \\ n_1 \ominus n_2 & = & 0 & n_1, n_2 \in \text{Multiplicity} \setminus \{\infty\} \quad n_2 > n_1 \\ n_1 \ominus n_2 & = & n_1 - n_2 & n_1, n_2 \in \text{Multiplicity} \setminus \{\infty\} \quad n_2 \leq n_1 \end{array}$$

$$\begin{array}{rcll} \infty \oplus n & = & \infty & \\ n \oplus \infty & = & \infty & \\ n_1 \oplus n_2 & = & \infty & n_1, n_2 \in \text{Multiplicity} \setminus \{\infty\} \quad n_2 + n_1 > K \\ n_1 \oplus n_2 & = & n_1 + n_2 & n_1, n_2 \in \text{Multiplicity} \setminus \{\infty\} \quad n_2 + n_1 \leq K \end{array}$$

$$\begin{array}{rcll} \infty \otimes n & = & \infty & \\ n \otimes \infty & = & \infty & \\ n_1 \otimes n_2 & = & \infty & n_1, n_2 \in \text{Multiplicity} \setminus \{\infty\} \quad n_2 \times n_1 > K \\ n_1 \otimes n_2 & = & n_1 \times n_2 & n_1, n_2 \in \text{Multiplicity} \setminus \{\infty\} \quad n_2 \times n_1 \leq K \end{array}$$

where $+$, $-$, and $\times$ are the usual operators on the natural numbers.

By $\underline{\psi}$ we mean the domain of ψ; notice that $\underline{\psi}$ is a plain effect. We call $\underline{\psi}$ the *underlying effect* of ψ. If A is some semantic object that contains multiplicity effects, $\underline{A}$ is the object which is identical to A, except that whereever A has an effect ψ, $\underline{A}$ has a plain $\underline{\psi}$.

We shall say that $\psi \leq \psi'$ if $\underline{\psi} = \underline{\psi'}$ and for all $ae \in \underline{\psi}$ it holds that $\psi(ae) \leq \psi'(ae)$. If A and A' are some semantic objects that contain multiplicity effects, we say that $A \leq A'$, if $\underline{A} = \underline{A'}$, and whereever A has an effect ψ, A' has ψ' such that $\psi \leq \psi'$. In the same way we define $<$.

By φ^n we shall mean the multiplicity effect for which it holds that $\underline{\varphi^n} = \varphi$ and for all $ae \in \underline{\varphi^n}$ it holds that $\varphi^n(ae) = n$.

Let us now state some facts about multiplicity effects

Lemma 3.12. *for all multiplicity effect ψ and ψ' and multiplicities n it holds that*

$$\begin{aligned}\underline{n \otimes \psi} &= \underline{\psi} \\ \underline{\psi \oplus \psi'} &= \underline{\psi} \cup \underline{\psi'} \\ \underline{\lceil \psi, \psi' \rceil} &= \underline{\psi} \cup \underline{\psi'} \\ \underline{\psi \ominus \psi'} &= \underline{\psi} \cup \underline{\psi'}\end{aligned}$$

Proof.

$$\begin{aligned}\underline{n \otimes \psi} &= \mathbf{dom}(n \otimes \psi) = \mathbf{dom}(\psi) = \underline{\psi} \\ \underline{\psi \oplus \psi'} &= \mathbf{dom}(\psi \oplus \psi') = \mathbf{dom}(\psi) \cup \mathbf{dom}(\psi') = \underline{\psi} \cup \underline{\psi'} \\ \lceil \psi, \psi' \rceil &= \mathbf{dom}(\lceil \psi, \psi' \rceil = \mathbf{dom}(\psi) \cup \mathbf{dom}(\psi') = \underline{\psi} \cup \underline{\psi'} \\ \underline{\psi \ominus \psi'} &= \mathbf{dom}(\psi \ominus \psi') = \mathbf{dom}(\psi) \cup \mathbf{dom}(\psi') = \underline{\psi} \cup \underline{\psi'}\end{aligned}$$

■

Lemma 3.13. *For all multiplicity effects ψ_1 and ψ_2 it holds that*

$$(\psi_1 \ominus \psi_2) \oplus \psi_2 = \lceil \psi_1, \psi_2 \rceil$$

Proof.
We have that

$$\begin{aligned}\underline{\lceil \psi_1, \psi_2 \rceil} &= \underline{\psi_1} \cup \underline{\psi_2} && \text{lemma 3.12} \\ &= \underline{\psi_1} \cup \underline{\psi_2} \cup \underline{\psi_2} \\ &= \underline{(\psi_1 \ominus \psi_2) \oplus \psi_2} && \text{lemma 3.12}\end{aligned}$$

So we only need to show that the multiplicities are equal, too. Let ae be an arbitrary atomic effect in $\underline{\psi_1} \cup \underline{\psi_2}$. We will show that $\lceil \psi_1, \psi_2 \rceil (ae) = ((\psi_1 \ominus \psi_2) \oplus \psi_2)(ae)$. Let us consider two cases.

1) $\psi_2^+(ae) \geq \psi_1^+(ae)$. In this case we have

$$\begin{aligned}((\psi_1 \ominus \psi_2) \oplus \psi_2)(ae) &= (\psi_1 \ominus \psi_2)^+(ae) \oplus \psi_2^+(ae) \\ &= (\psi_1^+(ae) \ominus \psi_2^+(ae)) \oplus \psi_2^+(ae) \\ &= 0 \oplus \psi_2^+(ae) \\ &= \lceil \psi_1, \psi_2 \rceil (ae)\end{aligned}$$

2) $\psi_2^+(ae) < \psi_1^+(ae)$. In this case we have

$$\begin{aligned}((\psi_1 \ominus \psi_2) \oplus \psi_2)(ae) &= (\psi_1 \ominus \psi_2)^+(ae) \oplus \psi_2^+(ae) \\ &= (\psi_1^+(ae) \ominus \psi_2^+(ae)) \oplus \psi_2^+(ae) \\ &= \psi_1^+(ae) \\ &= \lceil \psi_1, \psi_2 \rceil (ae)\end{aligned}$$

■

Lemma 3.14. *Let $\Psi_1, \Psi_2, \ldots$ be an infinite sequence of sets of multiplicity arrow effects such $\Psi_1 \leq \Psi_2 \leq \cdots$. Then there exists p such that $\Psi_p = \Psi_{p+1}$.*

Proof.
The result follows directly from the fact that for any set Φ of plain arrow effects there are only finitely many sets Ψ of multiplicity arrow effects such that $\underline{\Psi'} = \Phi$, namely

$$\prod_{\varepsilon.\varphi \in \Phi} (K+2)^{|\varphi|}$$

■

In region inference substitutions are triples (S_r, S_t, S_e) of simultaneous region, type, and effect substitutions. We now need substitutions whose range contains multiplicities and which work in a way so that they respect already existing multiplicities. A region substitution works like this

$$S_r(\psi) = \bigoplus_{\substack{\mathrm{put}(p) \in \underline{\psi} \\ n = \psi(\mathrm{put}(p))}} \{\mathrm{put}(S_r(p)) \mapsto n\} \;\oplus\; \bigoplus_{\substack{\mathrm{get}(p) \in \underline{\psi} \\ n = \psi(\mathrm{get}(p))}} \{\mathrm{get}(S_r(p)) \mapsto n\} \;\oplus\; \bigoplus_{\substack{\varepsilon \in \underline{\psi} \\ n = \psi(\varepsilon)}} \{\varepsilon \mapsto n\}$$

Note that the S_r does not change the total of multiplities in ψ: if $\{\mathrm{put}(\rho) \mapsto n\} \subseteq \psi$ and S_r transforms $\mathrm{put}(\rho)$ into $\mathrm{put}(p)$, then $\mathrm{put}(\rho)$ is removed from ψ and $\{\mathrm{put}(p) \mapsto n\}$ is added to the resulting effect.

Correspondingly effect substitutions behave in the following way when applied to an effect

$$S_e(\psi) = (\psi \setminus\setminus \mathbf{dom}(\mathrm{S_e})) \;\oplus \bigoplus_{\substack{\varepsilon'_\mathrm{i} \in \mathbf{dom}(\mathrm{S_e}) \cap \mathrm{fev}(\psi) \\ \varepsilon_\mathrm{i}.\psi_\mathrm{i} = \mathrm{S_e}(\varepsilon'_\mathrm{i})}} (\{\varepsilon_\mathrm{i} \mapsto 1\} \oplus \psi_\mathrm{i}) \otimes \psi(\varepsilon'_\mathrm{i})$$

Or in others words: whenever $\psi(\varepsilon) = n$ and $S_e(\varepsilon) = \varepsilon'.\psi'$, $\{\varepsilon \mapsto n\}$ is exchanged by $\{\varepsilon' \mapsto n\}$, and $\psi' \otimes n$ is added to the effect. The application of an effect substitution to an arrow effect is defined as follows

$$S_e(\varepsilon.\psi) = \begin{cases} \varepsilon'.S_e(\psi) \oplus \psi', & \text{if } S_e(\varepsilon) = \varepsilon'.\psi' \\ \varepsilon.S_e(\psi), & \text{otherwise} \end{cases}$$

Example 3.15. Define

$$\begin{aligned} S_e &= \{\ \varepsilon_1 \mapsto \varepsilon_2.\{\varepsilon_3 \mapsto 2, \mathrm{put}(\rho_2) \mapsto 2\}, \varepsilon_3 \mapsto \varepsilon_5.\{\mathrm{put}(\rho_5) \mapsto 2\}\ \} \\ S_r &= \{\rho_1 \mapsto \rho_2\} \\ \psi &= \{\mathrm{put}(\rho_1) \mapsto 1, \varepsilon_1 \mapsto 2\} \end{aligned}$$

We have that

$$\begin{aligned} S_r(\psi) &= \{\mathrm{put}(\rho_2) \mapsto 1\} \oplus \{\varepsilon_1 \mapsto 2\} = \{\mathrm{put}(\rho_2) \mapsto 1, \varepsilon_1 \mapsto 2\} \\ S_e(\psi) &= \psi \setminus\setminus \{\varepsilon_1\} \oplus\ (\{\varepsilon_2 \mapsto 2\} \oplus \{\varepsilon_3 \mapsto 2, \mathrm{put}(\rho_2) \mapsto 2\}) \otimes 2 \\ &= \{\mathrm{put}(\rho_1) \mapsto 1, \mathrm{put}(\rho_2) \mapsto 4, \varepsilon_2 \mapsto 2, \varepsilon_3 \mapsto 4\} \\ (S_r, \{\}, S_e)(\psi) &= \{\mathrm{put}(\rho_2) \mapsto 5, \varepsilon_2 \mapsto 2, \varepsilon_3 \mapsto 4\} \\ S_e(\varepsilon_3.\psi) &= \varepsilon_5.S_e(\psi) \oplus \{\mathrm{put}(\rho_5) \mapsto 2\} \\ &= \varepsilon_5.\{\mathrm{put}(\rho_1) \mapsto 1, \mathrm{put}(\rho_2) \mapsto 4, \mathrm{put}(\rho_5) \mapsto 2, \varepsilon_2 \mapsto 2, \varepsilon_3 \mapsto 4\} \\ S_e(\varepsilon_4.\psi) &= \varepsilon_4.S_e(\psi) = \varepsilon_4.\{\mathrm{put}(\rho_1) \mapsto 1, \mathrm{put}(\rho_2) \mapsto 4, \varepsilon_2 \mapsto 2, \varepsilon_3 \mapsto 4\} \end{aligned}$$

Here are some lemmas about substitutions

Lemma 3.16. *Let S and S' be two substitutions such that* $\mathbf{dom}(S) \cap \mathbf{dom}(S') = \emptyset$, *and* $\mathbf{dom}(S) \cap \mathrm{fvars}(\mathbf{rng}(S')) = \emptyset$ *then*

$$S \mid S' = S \circ S'$$

Proof.
Assume that

$$\mathbf{dom}(S) \cap \mathbf{dom}(S') = \emptyset \tag{0}$$

$$\mathbf{dom}(S) \cap \mathrm{fvars}(\mathbf{rng}(S')) = \emptyset \tag{1}$$

We have that $\mathbf{dom}(S \mid S') = \mathbf{dom}(S) \cup \mathbf{dom}(S') = \mathbf{dom}(S \circ S')$. Now let υ be an arbitrary variable in $\mathbf{dom}(S) \cup \mathbf{dom}(S')$. We will now show that $(S \mid S')(\upsilon) = (S \circ S')(\upsilon)$. We distinguish between two cases (remember that substitutions are extended to be the identity outside their domains when applied to type and effects)

1) $\upsilon \in \mathbf{dom}(S)$. In this case we have that

$$\begin{aligned} (S \mid S')(\upsilon) &= S(\upsilon) \\ &= S(S'(\upsilon)) \qquad (0) \\ &= (S \circ S')(\upsilon) \end{aligned}$$

2) $\upsilon \in \mathbf{dom}(S')$. In this case we have that

$$\begin{aligned} (S \mid S')(\upsilon) &= S'(\upsilon) \\ &= S(S'(\upsilon)) \qquad (1) \\ &= (S \circ S')(\upsilon) \end{aligned}$$

■

Lemma 3.17. *For all types τ and all substitutions (S_r, S_t, S_e), be they with or without multiplicities, the following equation holds*

$$(S_r, S_t, S_e)(\tau) = S_t(S_e(S_r(\tau)))$$

Proof.
Applying three substitutions with disjoint domains simultaneously can only give a result that is different from that of applying them sequentially, if the domain of one of the substitutions overlaps with the range of one of the substitutions that were applied before.

But in our case we have

$$\mathbf{dom}(S_t) \cap (\mathrm{ftv}(\mathbf{rng}(S_e)) \cup \mathrm{ftv}(\mathbf{rng}(S_r))) = \mathbf{dom}(S_t) \cap \{\} = \{\}$$

and

$$\mathbf{dom}(S_e) \cap \mathrm{fev}(\mathbf{rng}(S_r)) = \mathbf{dom}(S_e) \cap \{\} = \{\}$$

■

Lemma 3.18. *for all substitutions* (S_r, S_t, S_e)*, arrow effects* $\varepsilon.\psi$ *and types* τ *with multiplicities it holds that*

a) $\underline{S_r(\varepsilon.\psi)} = S_r(\underline{\varepsilon.\psi})$
b) $\underline{S_e(\psi)} = \underline{S_e}(\underline{\psi})$
c) $\underline{S_e(\varepsilon.\psi)} = \underline{S_e}(\underline{\varepsilon.\psi})$
d) $\underline{(S_r, S_t, S_e)(\tau)} = \underline{(S_r, S_t, S_e)}(\underline{\tau})$

Proof.

a) lemma 3.12 gives us that
$\underline{S_r(\varepsilon.\psi)} =$

$$\bigoplus_{\substack{\mathrm{put}(p)\in\underline{\psi}\\ n=\psi(\mathrm{put}(p))}} \{\mathrm{put}(S_r(p)) \mapsto n\} \;\oplus \bigoplus_{\substack{\mathrm{get}(p)\in\underline{\psi}\\ n=\psi(\mathrm{get}(p))}} \{\mathrm{get}(S_r(p)) \mapsto n\} \;\oplus \bigoplus_{\substack{\varepsilon\in\underline{\psi}\\ n=\psi(\varepsilon)}} \{\varepsilon \mapsto n\} \quad =$$

$$\bigcup_{\mathrm{put}(p)\in\underline{\psi}} \{\mathrm{put}(S_r(p))\} \;\cup \bigcup_{\mathrm{get}(p)\in\underline{\psi}} \{\mathrm{get}(S_r(p))\} \;\cup \bigcup_{\varepsilon\in\underline{\psi}} \{\varepsilon\} \quad =$$

$S_r(\underline{\varepsilon.\psi})$

b) By lemma 3.12 we have that
$\underline{S_e(\psi)} =$

$$(\psi \setminus\setminus \mathbf{dom}(\mathrm{S_e})) \;\oplus \bigoplus_{\substack{\varepsilon'_i\in\mathbf{dom}(\mathrm{S_e})\cap\mathrm{fev}(\psi)\\ \varepsilon_i.\psi_i=\mathrm{S_e}(\varepsilon'_i)}} (\{\varepsilon_i \mapsto 1\} \oplus \psi_i) \otimes \psi(\varepsilon'_i) \quad =$$

$$\underline{(\psi \setminus\setminus \mathbf{dom}(\mathrm{S_e}))} \;\cup \bigcup_{\substack{\varepsilon'_i\in\mathbf{dom}(S_e)\cap\mathrm{fev}(\psi)\\ \varepsilon_i.\psi_i=S_e(\varepsilon'_i)}} (\{\varepsilon_i\} \cup \underline{\psi_i}) \quad =$$

$\underline{S_e}(\underline{\psi})$

c) Is proven using b)

d) By induction on the structure of τ. The only interesting case is when τ is a function. This case is proven using induction on the subtypes and a) and c), sequentially, on the arrow effect.

■

3.3.2 Inference rules

We now present the rules for translating input expressions to regions expressions with sizes. The rules are very simple, and their adequacy should be obvious. They will therefore not be proven consistent with any dynamic semantics.

TRUE

$$\frac{}{TE \vdash \texttt{true at } p \hookrightarrow \texttt{true at } p : (bool, p), \{\mathrm{put}(p) \mapsto 1\}}$$

FALSE

$$\frac{}{TE \vdash \texttt{false at } p \hookrightarrow \texttt{false at } p : (bool, p), \{\mathrm{put}(p) \mapsto 1\}}$$

VARSIMPLE

$$\frac{(\sigma, p) = (\forall \vec{\alpha}\vec{\varepsilon}.\tau, p) = TE(x) \quad \underline{S} = (\{\}, \vec{\tau}/\vec{\alpha}, \varepsilon.\vec{\varphi}/\vec{\varepsilon})}{TE \vdash x[\vec{\tau}, \varepsilon.\vec{\varphi}] \hookrightarrow x : (S(\tau), p), \{\}}$$

VARCOMPOUND

$$\frac{(\forall \vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau, p') = TE(x) \quad \underline{S} = (\vec{p}/\vec{\rho}, \vec{\tau}/\vec{\alpha}, \varepsilon.\vec{\varphi}/\vec{\varepsilon})}{TE \vdash f[\vec{p}, \vec{\tau}, \varepsilon.\vec{\varphi}] \texttt{ at } p \hookrightarrow f[\vec{p}] \texttt{ at } p : (S(\tau), p), \{\mathrm{put}(p) \mapsto 1, \mathrm{get}(p') \mapsto 1\}}$$

We can only use an instantiation substitution with multiplicities. Any such substitution will do as long as the underlying instantiation is the one from the notes.

ABSTR

$$\frac{\underline{\mu_1'} = \mu_1 \quad TE + \{x \mapsto \mu_1'\} \vdash te \hookrightarrow e : \mu_2', \psi \quad \psi'' = \psi \oplus \psi' \quad \underline{\psi''} = \varphi}{TE \vdash \lambda : \mu_1 \xrightarrow{\varepsilon.\varphi} \mu_2 \; x.te \texttt{ at } p \hookrightarrow \lambda x.e \texttt{ at } p : \left(\mu_1' \xrightarrow{\varepsilon.\psi''} \mu_2', p\right), \{\mathrm{put}(p) \mapsto 1\}}$$

Note that we cannot extend the type assumtions with the type from the notes, but must pick a type with multiplicities. We insist that the effect over the arrow of the inferred type have the same atomic effects as the plain effect from the annotations.

APPL

$$\frac{TE \vdash te_1 \hookrightarrow e_1 : \left(\mu_2 \xrightarrow{\varepsilon.\psi} \mu_1, p\right), \psi_1 \quad TE \vdash te_2 \hookrightarrow e_2 : \mu_2, \psi_2}{TE \vdash te_1 \; te_2 : p, \varepsilon \hookrightarrow e_1 \; e_2 : \mu_1, \psi \oplus \psi_1 \oplus \psi_2 \oplus \{\varepsilon \mapsto 1\} \oplus \{\mathrm{get}(p) \mapsto 1\}}$$

LET

$$\frac{TE \vdash te_1 \hookrightarrow e_1 : (\tau', p), \psi_1 \quad TE + \{x \mapsto \sigma\} \vdash te_2 \hookrightarrow e_2 : \mu, \psi_2 \qquad \sigma = (\forall \vec{\varepsilon}\vec{\alpha}.\tau', p)}{TE \vdash \texttt{let } x : \forall \vec{\varepsilon}\vec{\alpha}.\tau = te_1 \texttt{ in } te_2 \hookrightarrow \texttt{let } x = e_1 \texttt{ in } e_2 : \mu, \psi_1 \oplus \psi_2}$$

We here use the quantifications from the notes. Of course, this only makes sense, if they really close the type w.r.t. to surroundings. As we shall se below that is the case for all the input expressions that are well formed under the type environment.

LETREC

$$\frac{\underline{\tau'} = \tau \quad TE + \{f \mapsto (\forall \vec{\rho}\vec{\varepsilon}.\tau', p)\} \vdash te_1 \hookrightarrow e_1 : (\tau', p), \psi_1 \quad TE + \{x \mapsto (\forall \vec{\rho}\vec{\varepsilon}\vec{\alpha}.\tau', p)\} \vdash te_2 \hookrightarrow e_2 : \mu_2, \psi_2}{TE \vdash \texttt{letrec } f : \forall \vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau \texttt{ at } p = te_1 \texttt{ in } te_2 \hookrightarrow \texttt{letrec } f[\vec{\rho}] \texttt{ at } p = e_1 \texttt{ in } e_2 : \mu_2, \psi_1 \oplus \psi_2}$$

LETREGION

$$\frac{TE \vdash te \hookrightarrow e : \mu, \psi \quad \mathrm{put}(\rho_1) \cdots \mathrm{put}(\rho_k) = \mathrm{PutEffects}(\varphi)}{TE \vdash \texttt{letregion}\ \varphi\ \texttt{in}\ te \hookrightarrow \texttt{letregion}\ \rho_1 : \psi^+(\mathrm{put}(\rho_1)) \cdots \rho_k : \psi^+(\mathrm{put}(\rho_k))\ \texttt{in}\ e : \mu, \psi \setminus\setminus \varphi}$$

Again we use the φ from the notes, without observing it away explicitly. But as we shall see later, φ is indeed the unobservable part of the inferred effect for all the input expressions that are well formed under the type environment.

IF

$$\frac{TE \vdash te_1 \hookrightarrow e_1 : (bool, p), \varphi_1 \quad TE \vdash te_2 \hookrightarrow e_2 : \mu, \psi_2 \quad TE \vdash te_3 \hookrightarrow e_3 : \mu, \psi_3}{TE \vdash \texttt{if}\ te_1 : p\ \texttt{then}\ te_2\ \texttt{else}\ te_3 \hookrightarrow \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \mu, \{\mathrm{get}(p') \mapsto 1\} \oplus \psi_1 \oplus \lceil \psi_2, \psi_3 \rceil}$$

The IF-rule does not add the effects of its two branches but takes the maximum of them. This is motivated by the fact that only one of the branches will be taken at run time.

Our maximum operation interacts with substitutions in the following way:

Theorem 3.19. *For all effect substitutions S_e and effects ψ_1 and ψ_2 it holds that*

$$S_e(\lceil \psi_1, \psi_2 \rceil) \geq \lceil S_e(\psi_1), S_e(\psi_2) \rceil$$

Proof.
We that

$$\begin{array}{lll}
\underline{S_e(\lceil \psi_1, \psi_2 \rceil)} & = \underline{S_e}(\lceil \psi_1, \psi_2 \rceil) & \text{lemma 3.18} \\
& = \overline{S_e}(\overline{\psi_1 \cup \psi_2}) & \text{lemma 3.12} \\
& = \overline{S_e}(\underline{\psi_1}) \cup \overline{S_e}(\underline{\psi_2}) & \\
& = \underline{S_e}(\overline{\psi_1}) \cup \underline{S_e}(\overline{\psi_2}) & \text{lemma 3.18} \\
& = \underline{\lceil S_e(\overline{\psi_1}), S_e(\overline{\psi_2}) \rceil} & \text{lemma 3.12}
\end{array} \qquad (0)$$

Now we have only left to show that for all $ae \in \underline{\psi_1} \cup \underline{\psi_2}$ it holds that $S_e(\lceil \psi_1, \psi_2 \rceil)(ae) \geq \lceil S_e(\psi_1), S_e(\psi_2) \rceil\,(ae)$. Let ae be an arbitrary atomic effect in $\underline{\psi_1} \cup \underline{\psi_2}$. By definition we have

$$\lceil S_e(\psi_1), S_e(\psi_2) \rceil\,(ae) = \mathrm{Max}(S_e(\psi_1)^+(ae), S_e(\psi_2)^+(ae))$$

Suppose $S_e(\psi_1)^+(ae) \geq S_e(\psi_2)^+(ae)$ (the opposite case is similar). We have that

$$S_e(\psi_1)^+(ae) \leq S_e(\lceil \psi_1, \psi_2 \rceil)^+(ae) = S_e(\lceil \psi_1, \psi_2 \rceil)(ae)$$

■

Example 3.20. Define

$$\begin{aligned}
S_e &= \{\ \varepsilon_1 \mapsto \varepsilon_1.\{\mathrm{put}(\rho_1) \mapsto 1\}\ \} \\
\psi &= \{\varepsilon_1 \mapsto 1, \mathrm{put}(\rho_1) \mapsto 1\} \\
\psi' &= \{\varepsilon_2 \mapsto 1, \mathrm{put}(\rho_1) \mapsto 2\}
\end{aligned}$$

We have that

$$\begin{aligned}
S_e(\lceil \psi, \psi' \rceil) &= \{\varepsilon_1 \mapsto 1, \varepsilon_2 \mapsto 1, \mathrm{put}(\rho_1) \mapsto 3\} \\
&> \{\varepsilon_1 \mapsto 1, \varepsilon_2 \mapsto 1, \mathrm{put}(\rho_1) \mapsto 2\} \\
&= \lceil S_e(\psi), S_e(\psi') \rceil
\end{aligned}$$

As shown in the example, theorem 3.19 cannot be strengthened to equality. This is problematic, as we we would like typings to be preserved under substitutions. This is important, if we want to have the possibility of using substitutions as a refinement mecanism in the inference algorithm. The problem arises, because our maximum operator is over-pessimistic: it takes the maximum one atomic effect at a time and does not take into account that two different effect variables by later substitutions can contribute with effects that overlap. The problem can be solved by adding an extra rule to the system

GE
$$\frac{TE \vdash te \hookrightarrow e : \mu, \psi \quad \psi' \geq \psi}{TE \vdash te \hookrightarrow e : \mu, \psi'}$$

Although this rule introduces some imprecision into the system, there is certainly nothing unsound about increasing the multiplicities of the effect. It can only cause the regions of the output expression to get greater multiplicities.

We shall not need a full-grown triple substitutions in the algorithm. We need to refine multiplicity information only by using substitutions which we call *multiplicity substitutions*:

Definition 3.21. *Let $\mathcal{S}$ be an effect substitution, and let A be an arrow effect, a type, a type scheme, or a type environment. We call $\mathcal{S}$* a multiplicity substitution on *A, if*

1. *for all $\varepsilon \in \mathbf{dom}(\mathcal{S})$ there exists ψ such that $\varepsilon.\psi = \mathcal{S}(\varepsilon)$.*
2. $\underline{\mathcal{S}(A)} = \underline{A}$

Example 3.22. Define
$$S_e = \{\ \varepsilon_1 \mapsto \varepsilon_1.\{\mathrm{put}(\rho_1) \mapsto 1\}\ \}$$
$$\psi = \{\varepsilon_1 \mapsto 1, \mathrm{put}(\rho_1) \mapsto 1\}$$
$$\psi' = \{\varepsilon_1 \mapsto 1, \mathrm{put}(\rho_2) \mapsto 2\}$$
S_e is a multiplicity substitution on ψ, but not on ψ'.

Example 3.23. The reason why the second condition in the definition, namely that $\underline{\mathcal{S}(A)} = \underline{A}$ is not sufficient, is that $\mathcal{S}$ could rename all effect variables in A, allthough it does not show, when you abstract from A's multiplicities. As an example, suppose
$$A = \psi = \{\varepsilon_1 \mapsto 1, \varepsilon_2 \mapsto \infty\}$$
And
$$\mathcal{S} = \{\varepsilon_1 \mapsto \varepsilon_2.\{\}, \varepsilon_2 \mapsto \varepsilon_1.\{\}\}$$
In this case we have
$$\underline{\mathcal{S}(\psi)} = \underline{\{\varepsilon_2 \mapsto 1, \varepsilon_1 \mapsto \infty\}} = \{\varepsilon_1, \varepsilon_2\} = \underline{\psi}$$
Here the effect of $\mathcal{S}$ is to let ε_1 and ε_2 swap multiplicities, which is a behaviour we cannot accept from a multiplicity substitution. If we did, it would not be possible to prove that a multiplicity substitution can only increase multiplicities, i.e. that if $\mathcal{S}$ is a multiplicity substitution on A then
$$\mathcal{S}(A) \geq A$$
As we shall see later, this property is necessary for proving that our inference algorithm terminates. We would also not be able to prove that a subset of a multiplicity substitution is also a multiplicity substitution.
Therefore we must also have the demand that a multiplicity substitution does not rename any effect variables, which means that all effect variables in its domain must be mapped to arrow effects with the same principal effect variable.

Here are some other useful lemmas about multiplicity substitutions

Lemma 3.24. *If $\mathcal{S}$ is a multiplicity substitution on a set of arrow effects Ψ, and $\mathcal{S}' \subseteq \mathcal{S}$ then $\mathcal{S}'$ is a also multiplicity substitution on Ψ.*

Proof.
The first requirement, name that $\mathcal{S}'$ should not rename any effect variables is obviously satisfied. Moreover, it is clear that if $\mathcal{S}$ does not introduce any new atomic effects, then $\mathcal{S}'$ has even less of a chance to do so.

■

Lemma 3.25. *Let A be an effect, arrow effect, type, type scheme, or type environment, and let $\mathcal{S}$ be a multiplicity substitution on A. It then holds that*

$$\mathcal{S}(A) \geq A$$

Proof.
Applying an effect substitution results in substituting arrows effects for effect variables. By the definition of multiplicity substitutions we know that $\mathcal{S}$ renames no effect variables, and we know that $\underline{\mathcal{S}(A)} = \underline{A}$. So $\mathcal{S}$ can only add effects at places where the atomic effects are already there. And as we have no negative multiplicities, the only effect of this can be to increase multiplicities.

■

We have now provided the foundation for using multiplicity substitutions as a refinement mechanism. We postpone the formulation of the concrete substitution lemma til the next chapter.

Now let us conclude this chapter by stating some important properties of the translation semantics.

First of all we want to be sure that our translation only adds multiplicities to the underlying typings:

Theorem 3.26. *If $T(\Phi, \underline{TE}, te) = (\mu, \varphi)$ and $TE \vdash te \hookrightarrow e : \mu', \psi$, then $\underline{\mu'} = \mu$, $\underline{\psi} = \varphi$, and $\underline{e} = \overline{te}$.*

Proof.
By structural induction on te

■

Now we know why it makes sense to follow the notes in the let, letrec, and letregion case. The environment, type, and effect contain exactly the same free type, effect, and region variables as at the same point of inference in region inference.

Finally, it is necessary to make sure that it is possible to translate all input expressions that are well formed under some type environment. That is what this theorem states

Theorem 3.27. *If $T(\Phi, \underline{TE}, te) = (\mu, \varphi)$, then there exist e, μ' og ψ such that $TE \vdash te \hookrightarrow e : \mu', \psi$.*

Proof.
By structural induction on te

■

4 A multiplicity inference algorithm

In this chapter we present an algorithm for the translation semantics of the previous chapter.

Type, region and effect inference have already been completed at the time the input expression was created, and as suggested by theorem 3.26, multiplicity inference is the problem of inferring multiplicities for the same types, effects and target expressions already inherently present in the input expression.

First we will discuss the problems that are connected with algorithm-W-like inference. Then we will show how these problems can be dealt with and construct another kind of algorithm, which makes use of the effect consistency of the input.

We prove that this algorithm is sound with respect to the translation semantics of multiplicity inference and that it always terminates. We then show that the algorithm is incomplete. Finally, we discuss how the algorithm can be implemented efficiently.

4.1 Algorithm-W-style inference

When faced with the problem of constructing an algorithm to go with the semantics of the previous chapter, it is natural to try to stay as close as possible to algorithm W, the classical type inference algorithm [Tof88, chapter 2, DM82]. This involves inferring as general typings for every subexpression as possible and then refining them by unification.

The most general assumptions we can make when e.g. the type environment is extended with a new variable in the λ case is to give all atomic effects minimal multiplicities. But how should unification be implemented? We would probably want a unifier to affect multiplicities only, and so it must be a multiplicity substitution on the environment, types, and effects. Let us look at the unification situation in the application case. The algorithm, let us call it $\mathcal{M}_W$, would look like this

$$
\begin{array}{l}
\mathcal{M}_W(TE, te_1\ te_2 : p, \varepsilon) \Longrightarrow \\
\quad \textbf{let} \quad (\mathcal{S}_1, \left(\mu_1 \xrightarrow{\varepsilon.\psi} \mu_2, \rho\right), \psi_1, e_1) = \mathcal{M}_W(te_1, TE) \\
\qquad\quad\ \ (\mathcal{S}_2, \mu_1', \psi_2, e_2) = \mathcal{M}_W(te_2, \mathcal{S}_1(TE)) \\
\qquad\quad\ \ \mathcal{S} = \mathrm{Unify}(\mathcal{S}_2(\mu_1), \mu_1') \\
\quad \textbf{in} \quad\ \ (\mathcal{S} \circ \mathcal{S}_2 \circ \mathcal{S}_1, e_1\ e_2, \mathcal{S}(\mathcal{S}_2(\mu_2)), \{\mathrm{get}(\rho) \mapsto 1\} \oplus \mathcal{S}(\psi_2 \oplus \mathcal{S}_2(\psi_1 \oplus \{\varepsilon \mapsto 1\} \oplus \psi)))
\end{array}
$$

Let us freeze the algorithm just before the call of Unify. If the algorithm works at all, it must certainly hold that

$$\mathcal{S}_1(TE) \vdash te \hookrightarrow e_1 : \left(\mu_1 \xrightarrow{\varepsilon.\psi} \mu_2, \rho\right), \psi_1$$

We can probably show that typings are preserved under multiplicity substitutions. So if $\mathcal{S}_2$ is a multiplicity substitution on $TE, \mu_1 \xrightarrow{\varepsilon.\psi} \mu_2$, and ψ_1, we get:

$$\mathcal{S}_2(\mathcal{S}_1(TE)) \vdash te \hookrightarrow e_1 : \mathcal{S}_2\left(\left(\mu_1 \xrightarrow{\varepsilon.\psi} \mu_2, \rho\right)\right), \mathcal{S}_2(\psi_1)$$

It must also hold that:

$$\mathcal{S}_2(\mathcal{S}_1(TE)) \vdash te' \hookrightarrow e_2 : \mu_1', \psi_2$$

As we are refining multiplicities only, we expect it to be the case that

$$\underline{\mathcal{S}_2(\mu_1)} = \underline{\mu_1'}$$

So the problem amounts to unifying arrow effects that are equal, except for their multiplicities. Remember that the effect of applying a multiplicity substitution is to add a constant to some of the multiplicities. But as $\mathcal{S}_2(\mu_1)$ and μ_1' are identical up to multiplicities and therefore also have the same effect variables, one cannot update the multiplicities of the one type without also updating those of the other. This means that the two types can only be unified by a substitution which sets all differing multiplicities to ∞. Therefore, this refinement mechanism will only work well in the case where we allow only few different multiplicities in types, best of all only 1 and ∞.

Example 4.2. Suppose

$$\begin{aligned}\mathcal{S}_2(\mu_1) &= (\alpha,\rho) \xrightarrow{\varepsilon_1.\{\mathrm{put}(\rho_1)\mapsto 1,\mathrm{put}(\rho_2)\mapsto 1\}} (\alpha,\rho)\\ \mu_1' &= (\alpha,\rho) \xrightarrow{\varepsilon_1.\{\mathrm{put}(\rho_1)\mapsto \infty,\mathrm{put}(\rho_2)\mapsto 1\}} (\alpha,\rho)\end{aligned}$$

Here the most general unifier would be $\{\varepsilon_1 \mapsto \varepsilon_1.\{\mathrm{put}(\rho_1)\mapsto\infty\}\}$.

This coarse-grainedness is not as critical as it may seem at first sight. As we will see in the chapter about experimental results, regions of sizes other that one and ∞ are very rare. However, it is not altogether clear, how it can be avoided for example to have the multiplicity zero.

But even if a solution to this problem was found, we are still left with the problem of finding fixed point type schemes for recursive functions. Following the idea of algorithm $\mathcal{RE}$ in [TT93a], we could have a separate procedure $\mathcal{R}ec$ to infer fixed point type schemes for recursive functions. $\mathcal{R}ec$ would be mutually recursive with $\mathcal{M}_W$ and be called in the letrec-case like this

$$\begin{array}{l}\mathcal{M}_W(TE, \texttt{letrec } f{:}\forall\vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau \texttt{ at } p = te_1 \texttt{ in } te_2) \Longrightarrow\\ \quad\textbf{let}\quad \sigma' = \text{decorate } \forall\vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau \text{ with minimal multiplicities}\\ \qquad\quad (\mathcal{S}, e_1, (\tau',\rho), \psi_1) = \mathcal{R}ec(TE, \sigma', f, te_1)\\ \qquad\quad (\mathcal{S}', e_2, \mu, \psi_2) = \mathcal{M}_W(TE) + \{f \mapsto \forall\vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau'\}, te_2)\\ \quad\textbf{in}\quad (\mathcal{S}'\circ\mathcal{S}, \mu, \texttt{letrec } f = e_1 \texttt{ in } e_2, \mathcal{S}'(\psi_1)\oplus\psi_2)\end{array}$$

$\mathcal{R}ec$ could be defined like this

$$\begin{array}{l}\mathcal{R}ec(TE, (\forall\vec{\rho}\vec{\varepsilon}.\tau, p), f, te) =\\ \quad\textbf{let}\quad (\mathcal{S}, e, \tau', \psi) = \mathcal{M}_W(TE + \{f \mapsto (\forall\vec{\rho}\vec{\varepsilon}.\tau, \rho)\}, te)\\ \quad\textbf{in}\\ \qquad\textbf{if}\ \ \mathcal{S}(\forall\vec{\rho}\vec{\varepsilon}.\tau) = \forall\vec{\rho}\vec{\varepsilon}.\tau'\\ \qquad\quad\textbf{then}\ (\mathcal{S}, e, \tau, \psi)\\ \qquad\quad\textbf{else}\\ \qquad\qquad\textbf{let}\quad (\mathcal{S}', e', \tau'', \psi') = \mathcal{R}ec(\mathcal{S}(TE), (\forall\vec{\rho}\vec{\varepsilon}.\tau', p), f, te)\\ \qquad\qquad\textbf{in}\quad (\mathcal{S}'\circ\mathcal{S}, e', \tau'', \psi')\end{array}$$

$\mathcal{R}ec$ searches for a fixed point type scheme for f by repeatedly calling $\mathcal{M}_W$. When it is found, $\mathcal{R}ec$ terminates.

The central question is whether $\mathcal{R}ec$ always terminates. If we could rely on $\mathcal{R}ec$ being monotonic in the sense that it would always hold that

$$\mathcal{S}(\forall\vec{\rho}\vec{\varepsilon}.\tau) \leq \forall\vec{\rho}\vec{\varepsilon}.\tau' \qquad (\dagger)$$

then $\mathcal{R}ec$ would indeed terminate, as there are only finitely many multiplicities (the relation $\geq$ was defined in the previous chapter). It may be the case that ($\dagger$) holds for every recursive call. However, there seems to be no obvious way of making sure.

We have therefore taken another approach to the problem, which allows us to avoid the problems of unification and which finds fixed point type schemes for recursive functions in a safe way.

4.2 Multiplicity consistency

The two main problems we discussed in the previous section, namely those of unification and the monotonicity of $\mathcal{R}ec$, both arose where one plain arrow effect occured with different multiplicities.

As type, region and effect inference is completed at the time our input is created, we know all the plain arrow effects of the inference tree in advance, before multiplicity inference begins. It is therefore possible to solve the problem of the same arrow effect occurring with different multiplicities by keeping one version of every arrow effect with multiplicities in a central table Ψ.

Algorithm $\mathcal{M}_W$ works by assuming minimal multiplicities to begin with when some type emerges, e.g. when the environment is extended with a new assumption or when a type scheme is instantiated, and then refining that information later by unification.

Our strategi is also based on gradual refinement, but of a more eager kind than that of $\mathcal{M}_W$. Whereas $\mathcal{M}_W$ allowed the same plain arrow effect to occur with different multiplicities, until the differences were resolved by unification, we have only one central version of every arrow effect in Ψ. Therefore, refinement must take place already at the time a new arrow effect is created. This happens when an arrow effect is put onto the arrow of a function type in the λ-case, and when type schemes are instantiated in the variable-cases. Refinement consists in updating Ψ so it contains the maximum of the newly created effect and the one already in Ψ.

Let us now proceed by specifying the concrete data structures the algorithm, which we call $\mathcal{M}$, should work on.

As input the algorithm, of course, takes an input expression te. In addition to this, as we argued above, it takes a table of all the arrow effects that will be encountered while processing te. We call this table Ψ. To begin with, this table will have zero multiplicities everywhere, but it will be updated continuously, while translating te. Furthermore, as we have polymorfism in multiplicity inference, we need to have an environment that maps effect variables to some kind of type scheme information. We will call this environment EE, the effect environment. We discuss the internal structure of EE below.

As output the algorithm must, of course, return a region program with multiplicities. As the multiplicities of the letregion-declarations, according to the LETREGION-rule, are always taken from the inferred effect.

The only component of the typing relation we have not yet included into data structures of the algorithm is TypeAndPlace. Types are important to multiplicity inference only to the extent they embed effect and region information. So we will not infer types explicitly. The type and place of te can always be reconstructed by taking the plain type inherently present in te and loading the multiplicities from the arrow effect table into it.

This argument also applies to the type scheme assumptions in EE: we do not need full-grown type schemes, it suffices to maintain the set of all arrow effects of the type scheme and the quantified effect and region variables.

This leads us to the following new semantic objects

$$\begin{array}{rrrl} \Psi & \in & \text{MulArrowEffSet} & = \text{fin}(\text{MulArrowEff}) \\ \forall\vec{\rho}\vec{\varepsilon}.\Psi \text{ or } \Xi & \in & \text{QuanMulArrowEffSet} & = \cup_{n\geq 0}\text{RegVar}^n \times \cup_{m\geq 0}\text{EffVar}^m \times \text{MulArrowEffSet} \\ EE & \in & \text{EffectEnvironment} & = \text{Var} \xrightarrow{\text{fin}} (\text{QuanMulArrowEffSet} \times \text{Place}) \end{array}$$

We have not yet specified how we intend to refine the arrow effect information in Ψ. We will do this by applying multiplicity substitutions to Ψ and thus increasing the multiplicities of the arrow effects.

All in all, this discussion gives us an inference algorithm of the following type

$$\mathcal{M}(\Psi, EE, te) = (\mathcal{S}, e, \psi)$$

Above we described some operations we want to perform on Ψ.

- We must be able to retrieve an effect given its principal effect variable. This operation only makes sense, if there is only one effect with which the effect variable occurs in a principal position.
- We want to update its multiplicities using multiplicity substitutions. It is hard to find out whether some substitution $\mathcal{S}$ is a multiplicity substitution on Ψ, if $\underline{\Psi}$ is not effect-consistent, i.e. if we are not certain that an effect variable is always surrounded by the same atomic effects, both in its principal and non-principal occurences.

The prerequisites for using these operations are summed up by the notion of *multiplicity consistency*:

Definition 4.3. *Let Ψ be a set of multiplicity arrow effects. We say that Ψ is* multiplicity-consistent, *if*

1) whenever $\varepsilon.\psi \in \Psi$ and $\varepsilon'.\psi' \in \Psi$, if $\varepsilon = \varepsilon'$ then $\psi = \psi'$
2) $\underline{\Psi}$ is effect-consistent.

If Ψ is multiplicity-consistent, and $\varepsilon \in \mathrm{fev}(\Psi)$, then by $\Psi[\varepsilon]$ we mean the effect ψ for which it holds that $\varepsilon.\psi \in \Psi$. By $\hat{\Psi}$ we mean the map

$$\hat{\Psi}(\varepsilon) = \Psi[\varepsilon], \qquad \textit{for all } \varepsilon \in \mathrm{fpev}(\Psi)$$

One might have expected the definition of multiplicity consistency to have some multiplicity version of requirement 2) from definition 3.2, e.g. that

**) Whenever $\varepsilon.\psi \in \Psi$ and $\varepsilon'.\psi' \in \Psi$, if $\psi(\varepsilon') = n$, then $n \otimes \psi' \sqsubseteq \psi$.*

where $\sqsubseteq$ would mean containment with multiplicities, or more formally:

$$\psi' \sqsubseteq \psi, \textit{ if } \underline{\psi'} \subseteq \underline{\psi} \textit{ and if for all } ae \in \underline{\psi'} \textit{ it holds that } \psi'(ae) \leq \psi(ae)$$

If *) is satisfied by Ψ to begin with, our algorithm will certainly preserve the property. However, we shall not need this as an explicit demand. In definition 4.3 requirement 1) is used for ensuring that a unique effect can be retrieved given some effect variable , and requirement 2) is necessary for making sure that the refinement substitions generated by the algorithm are multiplicity substitutions on the effect table. And this is all we need from this definition.

Example 4.4. Define

$$\begin{aligned} \Psi_1 = \{ \quad & \varepsilon_1.\{\mathrm{put}(\rho_1) \mapsto 1\}, \\ & \varepsilon_1.\{\mathrm{put}(\rho_1) \mapsto 2\} \ \} \\ \Psi_2 = \{ \quad & \varepsilon_1.\{\mathrm{put}(\rho_1) \mapsto 1\} \ \} \end{aligned}$$

Here, Ψ_2 is multiplicity-consistent, whereas Ψ_1 is not, as it violates 1) in the definition of multiplicity consistency.

We now extend the notion of multiplicity-consistency so that it covers types, type schemes, TypeAndPlace, and effects. This definition also makes it clear what we mean by loading multiplicities into plain types.

Definition 4.5. *We now introduce four relations* $\Psi \models \tau \Rightarrow \tau'$, $\Psi \models \mu \Rightarrow \mu'$, $\Psi \models \sigma \Rightarrow \sigma'$, *and* $\Psi \models \psi$. *Here* τ, μ, *and* σ *are without multiplicities, and* τ', μ', *and* σ' *are with multiplicities. The relations are defined by the following rules:*

$$\frac{\Psi \text{ is multiplicity-consistent}}{\Psi \models bool \Rightarrow bool} \qquad \frac{\Psi \text{ is multiplicity-consistent}}{\Psi \models \alpha \Rightarrow \alpha} \qquad \frac{\underline{\Psi} \models \underline{\psi}}{\Psi \models \psi}$$

$$\frac{\hat{\Psi}(\varepsilon) = \psi \quad \underline{\psi} = \varphi \qquad \Psi \models \mu_1 \Rightarrow \mu_1' \quad \Psi \models \mu_2 \Rightarrow \mu_2'}{\Psi \models \mu_1 \xrightarrow{\varepsilon.\varphi} \mu_2 \Rightarrow \mu_1' \xrightarrow{\varepsilon.\psi} \mu_2'} \qquad \frac{\begin{array}{c}\Psi \text{ is multiplicity-consistent} \\ \sigma = \forall \vec{\rho}\vec{\varepsilon}\vec{\alpha}.\tau \quad \Xi = \forall \vec{\rho}\vec{\varepsilon}.\Psi' \\ \mathrm{fev}(\sigma) \subseteq \mathrm{fev}(\Psi) \quad \Psi \cup \Psi' \models \tau \Rightarrow \tau' \\ \mathrm{fev}(\vec{\varepsilon}) \cap \mathrm{fev}(\Psi) = \emptyset \quad \mathrm{frv}(\vec{\rho}) \cap \mathrm{frv}(\Psi) = \emptyset\end{array}}{\Psi, \Xi \models \sigma \Rightarrow \forall \vec{\rho}\vec{\varepsilon}\vec{\alpha}.\tau'}$$

$$\frac{\Psi \models \tau \Rightarrow \tau'}{\Psi \models (\tau, \rho) \Rightarrow (\tau', \rho)} \qquad \frac{\Psi, \Xi \models \sigma \Rightarrow \sigma'}{\Psi, (\Xi, p) \models (\sigma, p) \Rightarrow (\sigma', p)}$$

$\Psi \models \tau \Rightarrow \tau'$ *can be read* τ' is multiplicity-consistent under Ψ. *In the same way we read* $\Psi \models \mu \Rightarrow \mu'$, $\Psi, \Xi \models \sigma \Rightarrow \sigma'$, *and* $\Psi \models \psi$.

Let TE *and* TE' *be type environments and EE and effect environment. We say that* TE *is multiplicity-consistent under* Ψ, *written* $\Psi, EE \models TE \Rightarrow TE'$, *if* $\mathbf{dom}(TE) = \mathbf{dom}(TE')$ *and* $\Psi, EE(x) \models TE(x) \Rightarrow TE'(x)$ *for all* $x \in \mathbf{dom}(TE)$.

Let A *be a set of types, type schemes, effects, and type environments. We say that* A *is multiplicity-consistent under* Ψ, *if all* $a \in A$ *are multiplicity-consistent under* Ψ.

Each of the rules has its counterpart in definition 3.4. Like in definition 3.4 the type scheme rule requires that all the free effect and region variables of the type scheme be contained in Ψ, and if a version of the type scheme is picked so that the bound effect and region variables are distinct from the ones in Ψ then it must be possible to extend Ψ with Ψ' so that the type of the type scheme is multiplicity-consistent under $\Psi \cup \Psi'$. And this Ψ' must be the arrow effects from Ξ, after renaming of bound variables.

Example 4.6. Define

$$\begin{array}{rl}\Psi = \{ & \varepsilon_0.\{\mathrm{put}(\rho_1) \mapsto 1\}, \\ & \varepsilon_1.\{\mathrm{put}(\rho_1) \mapsto 1\}, \\ & \varepsilon_2.\{\mathrm{put}(\rho_1) \mapsto 2, \varepsilon_0 \mapsto 1, \varepsilon_1 \mapsto 1\}\ \}\end{array}$$

Ψ is multiplicity-consistent. Define

$$\begin{aligned}\sigma &= \forall \varepsilon_2.\mu \xrightarrow{\varepsilon_2.\{\mathrm{put}(\rho_1), \varepsilon_1\}} \mu \\ \sigma' &= \forall \varepsilon_2.\mu \xrightarrow{\varepsilon_2.\{\mathrm{put}(\rho_1) \mapsto 3, \varepsilon_1 \mapsto 2\}} \mu \\ \Xi &= \forall \varepsilon_2.\{\varepsilon_2.\{\mathrm{put}(\rho_1) \mapsto 3, \varepsilon_1 \mapsto 2\}\}\end{aligned}$$

We have that $\Psi, \Xi \models \sigma \Rightarrow \sigma'$, because after renaming bound variables we get

$$\begin{aligned}\sigma &= \forall \varepsilon_4.\mu \xrightarrow{\varepsilon_4.\{\mathrm{put}(\rho_1), \varepsilon_1\}} \mu \\ \sigma' &= \forall \varepsilon_4.\mu \xrightarrow{\varepsilon_4.\{\mathrm{put}(\rho_1) \mapsto 3, \varepsilon_1 \mapsto 2\}} \mu \\ \Xi &= \forall \varepsilon_4.\{\varepsilon_4.\{\mathrm{put}(\rho_1) \mapsto 3, \varepsilon_1 \mapsto 2\}\}\end{aligned}$$

and

$$\mathrm{fev}(\sigma) = \{\varepsilon_1\} \subseteq \mathbf{dom}(\hat{\Psi}) \qquad \mathrm{fev}(\varepsilon_4) \cap \mathbf{dom}(\hat{\Psi}) = \emptyset$$

$$\Psi \cup \{\varepsilon_4.\{\mathrm{put}(\rho_1) \mapsto 3, \varepsilon_1 \mapsto 2\}\} \models \mu \xrightarrow{\varepsilon_4.\{\mathrm{put}(\rho_1), \varepsilon_1\}} \mu \Rightarrow \mu \xrightarrow{\varepsilon_4.\{\mathrm{put}(\rho_1) \mapsto 3, \varepsilon_1 \mapsto 2\}} \mu$$

Here are two useful lemmas about multiplicity consistency

Lemma 4.7. *Let Ψ be a multiplicity-consistent set of arrow effects.*

a) Suppose $\underline{\Psi} \models \tau$. Then there exists a unique τ' such that

$$\Psi \models \tau \Rightarrow \tau'$$

b) Suppose $\underline{\Psi} \models \mu$. Then there exists a unique μ' such that

$$\Psi \models \mu \Rightarrow \mu'$$

c) Suppose $\underline{\Psi} \models \sigma$. Then for given Ξ there is at most one σ' such that

$$\Psi, \Xi \models \sigma \Rightarrow \sigma'$$

d) Suppose $\underline{\Psi} \models (\sigma, p)$. Then for given Ξ there is at most one σ' such that

$$\Psi, \Xi \models (\sigma, p) \Rightarrow (\sigma', p)$$

e) Suppose $\underline{\Psi} \models TE$. Then for given EE there is at most one TE' such that

$$\Psi, \Xi \models TE \Rightarrow TE'$$

Proof.

a) τ' is the type obtained by exchanging every arrow effect $\varepsilon.\varphi$ in τ by $\varepsilon.\hat{\Psi}(\varepsilon)$.

b) Follows directly from a)

c) Suppose we can pick $\vec{\rho}$ and ε such that $\Xi = \forall\vec{\rho}\vec{\varepsilon}.\Psi'$ and $\sigma = \forall\vec{\rho}\vec{\varepsilon}\vec{\alpha}.\tau$ and

$$\mathrm{fev}(\vec{\varepsilon}) \cap \mathrm{fev}(\Psi) = \{\}$$

$$\mathrm{fev}(\sigma) \subseteq \mathrm{fev}(\Psi) \quad \mathrm{frv}(\sigma) \subseteq \mathrm{frv}(\Psi)$$

$$\Psi \cup \Psi' \text{ is multiplicity-consistent}$$

and

$$\underline{\Psi \cup \Psi'} \models \tau$$

then we know by a) that there exists only one τ' such that

$$\Psi \cup \Psi' \models \tau \Rightarrow \tau'$$

and by definition 4.5 we have that $\sigma' = \forall\vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau'$. So σ' is also unique.

On the other hand, if no $\vec{\rho}$ and $\vec{\varepsilon}$ could be picked such that the premises above could be satisfied, there would also exist no σ'.

d) Follows directly from c)

e) Is obtained from d) by pointwise extension

■

Lemma 4.8. *Suppose* $\Psi, \Xi \models \sigma \Rightarrow \sigma'$ *and* $\mathcal{S}$ *is a multiplicity substitution on* Ψ *then*

$$\mathcal{S}(\forall \vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau') = \forall \vec{\rho}\vec{\alpha}\vec{\varepsilon}.\mathcal{S} \setminus\setminus \{\vec{\varepsilon}\}(\tau')$$

where $\forall \vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau' = \sigma'$.

Proof.
What we must show is that there is no capture, i.e. that for all $\varepsilon \in \mathrm{fev}(\sigma') \cap \mathbf{dom}(\mathcal{S})$ it holds that there exists ψ such that $\varepsilon.\psi = \mathcal{S}(\varepsilon)$ and $\mathrm{fev}(\psi) \subseteq \mathrm{fev}(\sigma')$ and $\mathrm{frv}(\psi) \subseteq \mathrm{frv}(\sigma')$.

By the definition of multiplicity consistency it must be possible to pick $\vec{\varepsilon'}$ and $\vec{\rho'}$ such that the following premises hold

$$\mathrm{fev}(\sigma) \in \mathrm{fev}(\Psi) \tag{0}$$

$$\mathrm{fev}(\vec{\varepsilon'}) \cap \mathrm{fev}(\Psi) = \{\} \quad \mathrm{frv}(\vec{\rho'}) \cap \mathrm{frv}(\Psi) = \{\} \tag{1}$$

$$\Psi \cup \Psi' \models \tau \Rightarrow \mathcal{R}(\tau') \tag{2}$$

where

$$\forall \vec{\rho'}\vec{\alpha}\vec{\varepsilon'}.\tau = \sigma \quad \forall \vec{\rho'}\vec{\varepsilon'}.\Psi' = \Xi \quad \mathcal{R} = [\vec{\rho'}/\vec{\rho}, \vec{\varepsilon'}/\vec{\varepsilon}] \tag{3}$$

Notice that $\mathcal{R}$ is not a real substitution, but just a renaming of variables (effect variables are mapped to effect variables and not to arrow effects). Let ε be an arbitrary effect variable in $\mathbf{dom}(\mathcal{S}) \cap \mathrm{fev}(\sigma)$, and let $\varepsilon.\psi = \mathcal{S}(\varepsilon)$. By (0) we know that $\varepsilon \in \Psi$. And for $\mathcal{S}$ to be a multiplicity substitution on Ψ it must hold that

$$\underline{\psi} \subseteq \underline{\hat{\Psi}(\varepsilon)} \tag{4}$$

as $\mathcal{S}$ would otherwise introduce new atomic effects into Ψ.

All we have to show now is that

$$\forall \varepsilon'' \in \mathrm{fev}(\hat{\Psi}(\varepsilon)) \text{ it holds that } \varepsilon'' \notin \mathrm{fev}(\vec{\varepsilon}) \tag{5}$$

and that

$$\forall \rho \in \mathrm{frv}(\hat{\Psi}(\varepsilon)) \text{ it holds that } \rho \notin \mathrm{frv}(\vec{\rho}) \tag{6}$$

To show (5) (the argument for (6) is similar), let ε'' be an arbitrary effect variable in $\mathrm{fev}(\hat{\Psi}(\varepsilon))$. Suppose $\varepsilon'' \in \vec{\varepsilon}$. Let $\varepsilon''' = \mathcal{R}(\varepsilon'')$. By (1) we know that $\varepsilon''' \neq \varepsilon''$. But then (2) cannot hold, since ε will occur somewhere in $\mathcal{R}(\tau')$ without ε''. So we must conclude that $\varepsilon'' \notin \vec{\varepsilon}$.

■

As mentioned above, we update Ψ by applying substitutions to it. If we want the environment to be multiplicity-consistent, it is necessary to use *consistency preserving* substitutions:

Definition 4.9. *let* Ψ *be a multiplicity-consistent set of arrow effects, and let* S_e *be an effect substitution. We shall say that* S_e *is* a consistency preserving substitution on Ψ, *if* $S_e(\Psi)$ *is also multiplicity-consistent.*

Lemma 4.10. *Let Ψ be a multiplicity-consistent set of arrow effects, and let $\mathcal{S}$ be a multiplicity substitution on Ψ. Then $\mathcal{S}$ is a consistency preserving substitution on Ψ.*

Proof.
We will show that $\mathcal{S}(\Psi)$ is multiplicity-consistent. By the definition of multiplicity substitutions $\underline{\Psi} = \underline{\mathcal{S}(\Psi)}$. So $\underline{\mathcal{S}(\Psi)}$ is effect-consistent. So the second requirement of multiplicity consistency is satisfied. And as none of the principal effect variables are renamed by $\mathcal{S}$, the first requirement must also be satisfied.

■

The following lemma relates multiplicity tables, types, effects, and substitutions.

Lemma 4.11. *Let $\mathcal{S}$ be a multiplicity substitution on the set of arrow effects Ψ. We have that*

a) If $\Psi \models \tau \Rightarrow \tau'$ then $\mathcal{S}$ is a multiplicity substitution on τ' and $\mathcal{S}(\Psi) \models \tau \Rightarrow \mathcal{S}(\tau')$.
b) If $\Psi, \Xi \models \sigma \Rightarrow \sigma'$, then $\mathcal{S}$ is a multiplicity substitution on σ' and Ξ, and $\mathcal{S}(\Psi), \mathcal{S}(\Xi) \models \sigma \Rightarrow \mathcal{S}(\sigma')$.
c) If $\Psi, EE \models TE \Rightarrow TE'$, then $\mathcal{S}$ is a multiplicity substitution on TE' and EE, and $\mathcal{S}(\Psi), \mathcal{S}(EE) \models TE \Rightarrow \mathcal{S}(TE')$.
d) If $\Psi \models \psi$, then $\mathcal{S}$ is a multiplicity substitution on ψ and $\mathcal{S}(\Psi) \models \mathcal{S}(\psi)$.

Proof.
A proof of this lemmas can be found in appendix B.

■

Lemma 4.12. *If $\Psi \models \tau \Rightarrow \tau'$, and S_e is a consistency preserving substitution on Ψ, then $S_e(\Psi) \models \underline{S_e}(\tau) \Rightarrow S_e(\tau')$.*

Proof.
The proof resembles the one for lemma 4.11a).

■

When showing that typings are preserved under substitutions this lemma is useful

Lemma 4.13. *Let σ be a type scheme, τ a type, $\mathcal{I}$ an instantiation, and $\mathcal{S}$ a multiplicity substitution on τ and $\mathcal{I}$. Suppose that $\sigma \succ \tau$ via $\mathcal{I}$. Then it holds that*

$$\mathcal{S}(\sigma) \succ \mathcal{S}(\tau) \text{ via } \mathcal{S}(\mathcal{I})$$

Proof.
We assume

$$\mathcal{S} \text{ is a multiplicity substitution on } \tau \text{ and } \mathcal{I} \tag{0}$$

$$\sigma \succ \tau \text{ via } \mathcal{I} \tag{1}$$

We pick $\vec{\rho}$ and $\vec{\varepsilon'}$ such that

$$\forall \vec{\rho}\vec{\alpha}\vec{\varepsilon'}.\tau' = \sigma \tag{2}$$

$$\mathrm{fev}(\vec{\varepsilon'}) \cap (\mathrm{fev}(\mathbf{dom}(\mathcal{S})) \cup \mathrm{fev}(\mathbf{rng}(\mathcal{S}))) = \{\} \quad \mathrm{frv}(\vec{\rho}) \cap (\mathrm{frv}(\mathbf{dom}(\mathcal{S})) \cup \mathrm{frv}(\mathbf{rng}(\mathcal{S}))) = \{\} \tag{3}$$

(1) implies that there exists $\vec{\tau}$, $\vec{p}$, and $\vec{\varepsilon.\psi}$ such that

$$\mathcal{I} = [\vec{\tau}, \vec{\alpha}, \vec{\varepsilon.\psi}] \qquad S = [\vec{\tau}/\vec{\alpha}, \vec{p}/\vec{\rho}, \vec{\varepsilon.\psi}/\vec{\varepsilon'}] \tag{4}$$

$$\tau = S(\tau') \tag{5}$$

By (3) we have that

$$\mathcal{S}(\sigma) = \forall \vec{\rho}\vec{\alpha}\vec{\varepsilon'}.\mathcal{S}(\tau') \tag{6}$$

Now let

$$S' = [\mathcal{S}(\vec{\tau})/\vec{\alpha}, \vec{p}/\vec{\rho}, \mathcal{S}(\vec{\varepsilon.\psi})/\vec{\varepsilon'}] \tag{7}$$

We must show that

$$S'(\mathcal{S}(\tau')) = \mathcal{S}(\tau) \tag{8}$$

By (5) we know that

$$\mathcal{S}(\tau) = \mathcal{S}(S(\tau')) \tag{9}$$

So proving (8) amounts to showing that

$$S' \circ \mathcal{S} = \mathcal{S} \circ S \tag{10}$$

We know that the domain of both functions equals $\mathrm{frv}(\vec{\rho}) \cup \mathrm{ftv}(\vec{\alpha}) \cup \mathrm{fev}(\vec{\varepsilon'}) \cup \mathbf{dom}(\mathcal{S})$. That (10) holds for all region and type variables is obvious. Let ε be an arbitrary $\mathrm{fev}(\vec{\varepsilon'}) \cup \mathbf{dom}(\mathcal{S})$ effect variable in the domain. We distinguish between two cases

1) $\varepsilon \in \mathrm{fev}(\vec{\varepsilon'})$. We suppose $\varepsilon = \varepsilon'_i$ for some $i \in \{1, \ldots, \mid \vec{\varepsilon'} \mid\}$. Let $\varepsilon_i.\psi_i$ be the corresponding member of $\vec{\varepsilon.\psi}$ at the i'th position. We have

$$\begin{aligned} S' \circ \mathcal{S}(\varepsilon'_i) &= S'(\varepsilon'_i) && (3)\\ &= \mathcal{S}(\varepsilon_i.\psi_i) && (7)\\ &= \mathcal{S}(S(\varepsilon'_i)) && (4)\\ &= \mathcal{S} \circ S(\varepsilon'_i) \end{aligned}$$

2) $\varepsilon \notin \mathrm{fev}(\vec{\varepsilon'})$. In this case we have

$$\begin{aligned} S' \circ \mathcal{S}(\varepsilon) &= \mathcal{S}(\varepsilon) && (3)\\ &= \mathcal{S}(S(\varepsilon)) && (4)\ (S \text{ is the identity outside } \mathrm{fev}(\vec{\varepsilon'}))\\ &= \mathcal{S} \circ S(\varepsilon) \end{aligned}$$

■

Our substitution lemma looks like this

Lemma 4.14. *If* $\Psi, EE \models TE \Rightarrow TE'$, $\Psi \models \mu \Rightarrow \mu'$, $\Psi \models \psi$, $\mathcal{S}$ *is a multiplicity substitution on* Ψ, *and* $TE' \vdash te \hookrightarrow e : \mu', \psi$, *then*

$$\mathcal{S}(TE') \vdash te \hookrightarrow e : \mathcal{S}(\mu'), \mathcal{S}(\psi)$$

Proof.
By induction on the depth of inference. Uses lemma 4.11 and lemma 4.13. The only really unusual case is the IF case, where the conclusion is obtained by using the GE-rule after the IF-rule to solve the problem of the non-commutativity of substitutions and the maximum operation.

■

Before proceeding with the presentation the concrete algorithm, let us sum up the main points of this chapter

- We demand that the input expression be effect consistent under some type environment.
- We do not want to deal with types explicitly, but only with their arrow effects.
- As a refinement mechanism we use multiplicity substitutions.

These ideas are expressed formally by the following theorem, which will be proven correct later in this chapter:

Soundness and termination. *If* $\Psi, EE \models TE \Rightarrow TE'$, $\Psi \models \mu \Rightarrow \mu'$, *and* $T(\underline{\Psi}, TE, te) = (\mu, \varphi)$, *then there exist* $\mathcal{S}$, e, *and* ψ *such that* ■

$$(\mathcal{S}, e, \psi) = \mathcal{M}(\Psi, EE, te)$$

where

$$\mathcal{S} \text{ is a multiplicity substitution on } \Psi$$

and

$$\mathcal{S}(TE') \vdash te \hookrightarrow e : \mathcal{S}(\mu'), \psi$$

4.3 The algorithm

In this section we present and explain algorithm $\mathcal{M}$. The algorithm look like this

$$\mathcal{M}(\Psi, EE, \texttt{true at } p) \Longrightarrow (\{\}, \texttt{true at } \rho, \{\mathrm{put}(\rho) \mapsto 1\})$$

$$\mathcal{M}(\Psi, EE, \texttt{false at } p) \Longrightarrow (\{\}, \texttt{false at } \rho, \{\mathrm{put}(\rho) \mapsto 1\})$$

$$\begin{array}{l}\mathcal{M}(\Psi, EE, x[\vec{\tau}, \varepsilon.\vec{\varphi}]) \Longrightarrow \\ \quad \textbf{let} \quad (\forall \vec{\varepsilon}.\Psi', p') = EE(f) \\ \qquad\quad \mathcal{S} = \mathrm{Instantiate}(\varepsilon.\vec{\varphi}, \forall \vec{\varepsilon}.\Psi', \Psi) \\ \quad \textbf{in} \quad (\mathcal{S}, x, \{\})\end{array}$$

$$\begin{array}{l}\mathcal{M}(\Psi, EE, f[\vec{p}, \vec{\tau}, \varepsilon.\vec{\varphi}] \texttt{ at } p]) \Longrightarrow \\ \quad \textbf{let} \quad (\forall \vec{\rho}\vec{\varepsilon}.\Psi', p') = EE(f) \\ \qquad\quad \mathcal{S} = \mathrm{Instantiate}(\varepsilon.\vec{\varphi}, \forall \vec{\varepsilon}.[\vec{p}/\vec{\rho}](\Psi'), \Psi) \\ \quad \textbf{in} \quad (\mathcal{S}, f[\vec{p}] \texttt{ at } p, \{\mathrm{put}(p) \mapsto 1, \mathrm{get}(p') \mapsto 1\})\end{array}$$

$$\begin{array}{l}\mathcal{M}(\Psi, EE, \lambda : (\tau_1, p_1) \xrightarrow{\varepsilon.\varphi} \mu_2\ x.te \texttt{ at } p) \Longrightarrow \\ \quad \textbf{let} \quad (\mathcal{S}, e, \psi) = \mathcal{M}(\Psi, EE + \{x \mapsto (\{\}, p_1)\}, te) \\ \qquad\quad \mathcal{S}' = \textbf{if } \varepsilon \in \mathrm{fev}(\psi) \\ \qquad\qquad\qquad \textbf{then } \{\varepsilon \mapsto \varepsilon.\varphi^{\infty}\} \\ \qquad\qquad\qquad \textbf{else } \left\{\varepsilon \mapsto \varepsilon.(\psi \ominus \widehat{\mathcal{S}(\Psi)}(\varepsilon))\right\} \\ \quad \textbf{in} \quad (\mathcal{S}' \circ \mathcal{S}, \lambda x.e \texttt{ at } p, \{\mathrm{put}(p) \mapsto 1\})\end{array}$$

$$\begin{array}{l}\mathcal{M}(\Psi, EE, te_1\ te_2 : p, \varepsilon) \Longrightarrow \\ \quad \textbf{let} \quad (\mathcal{S}_1, e_1, \psi_1) = \mathcal{M}(\Psi, EE, te_1) \quad (\mathcal{S}_2, e_2, \psi_2) = \mathcal{M}(\mathcal{S}_1(\Psi), \mathcal{S}_1(EE), te_2) \\ \quad \textbf{in} \quad (\mathcal{S}_2 \circ \mathcal{S}_1, e_1\ e_2, \psi_2 \oplus \mathcal{S}_2(\psi_1) \oplus \{\varepsilon \mapsto 1\} \oplus \mathcal{S}_2(\widehat{\mathcal{S}_1(\Psi)})(\varepsilon) \oplus \{\mathrm{get}(p) \mapsto 1\})\end{array}$$

$$\begin{array}{l}\mathcal{M}(\Psi, EE, \texttt{let } x : \forall \vec{\varepsilon}\vec{\alpha}.\tau = te_1 \texttt{ in } te_2) \Longrightarrow \\ \quad \textbf{let} \quad (\mathcal{S}_1, e_1, \psi_1) = \mathcal{M}(\Psi, EE, te_1) \\ \qquad\quad (\mathcal{S}_2, e_2, \psi_2) = \mathcal{M}(\mathcal{S}_1(\Psi), \mathcal{S}_1(EE) + \left\{x \mapsto \left(\forall \vec{\varepsilon}.\mathrm{Lookup}_{\mathcal{S}_1(\Psi)}(\vec{\varepsilon}), p\right)\right\}, te_2) \\ \quad \textbf{in} \quad (\mathcal{S}_2 \circ \mathcal{S}_1, \texttt{let } x = e_1 \texttt{ in } e_2, \mathcal{S}_2(\psi_1) \oplus \psi_2)\end{array}$$

$$\begin{array}{l}\mathcal{M}(\Psi, EE, \texttt{letrec } f : \forall \vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau \texttt{ at } p = te_1 \texttt{ in } te_2) \Longrightarrow \\ \quad \textbf{let} \quad (\mathcal{S}_1, e_1, \psi_1) = \mathcal{R}ec(\Psi, EE, (\forall \vec{\rho}\vec{\varepsilon}.\mathrm{Lookup}_{\Psi}(\vec{\varepsilon}), p), f, te_1) \\ \qquad\quad (\mathcal{S}_2, e_2, \psi_2) = \mathcal{M}(\mathcal{S}_1(\Psi), \mathcal{S}_1(EE) + \left\{f \mapsto \left(\forall \vec{\rho}\vec{\varepsilon}.\mathrm{Lookup}_{\mathcal{S}_1(\Psi)}(\vec{\varepsilon}), p\right)\right\}, te_2) \\ \quad \textbf{in} \quad (\mathcal{S}_2 \circ \mathcal{S}_1, \texttt{letrec } f = e_1 \texttt{ in } e_2, \mathcal{S}_2(\psi_1) \oplus \psi_2)\end{array}$$

$$\begin{array}{l}\mathcal{M}(\Psi, EE, \texttt{letregion } \varphi \texttt{ in } te) \Longrightarrow \\ \quad \textbf{let} \quad (\mathcal{S}, e, \psi) = \mathcal{M}(\Psi, EE, te) \\ \qquad\quad \{\mathrm{put}(\rho_1), \ldots, \mathrm{put}(\rho_k)\} = \mathrm{PutEffects}(\varphi) \\ \quad \textbf{in} \quad (\mathcal{S}, \texttt{letregion } \rho_1 : \psi^{+}(\mathrm{put}(\rho_1)) \cdots \rho_k : \psi^{+}(\mathrm{put}(\rho_k)) \texttt{ in } e, \psi \setminus\setminus \varphi)\end{array}$$

$$\begin{array}{l}\mathcal{M}(\Psi, EE, \texttt{if } te_1 : p \texttt{ then } te_2 \texttt{ else } te_3) \Longrightarrow \\ \quad \textbf{let} \quad (\mathcal{S}_1, e_1, \psi_1) = \mathcal{M}(\Psi, EE, te_1) \\ \qquad\quad (\mathcal{S}_2, e_2, \psi_2) = \mathcal{M}(\mathcal{S}_1(\Psi), \mathcal{S}_1(EE), te_2) \\ \qquad\quad (\mathcal{S}_3, e_3, \psi_3) = \mathcal{M}(\mathcal{S}_2(\mathcal{S}_1(\Psi)), \mathcal{S}_2(\mathcal{S}_1(EE)), te_3) \\ \quad \textbf{in} \quad (\mathcal{S}_3 \circ \mathcal{S}_2 \circ \mathcal{S}_1, \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3, \{\mathrm{get}(p) \mapsto 1\} \oplus \mathcal{S}_3 \circ \mathcal{S}_2(\psi_1) \oplus \lceil \mathcal{S}_3(\psi_2), \psi_3 \rceil)\end{array}$$

$$
\begin{array}{l}
\mathcal{R}ec(\Psi, EE, (\forall\vec{\rho}\vec{\varepsilon}.\Psi', p), f, te) = \\
\quad \textbf{let} \quad (\mathcal{S}, e, \psi) = \mathcal{M}(\Psi, EE + \{f \mapsto (\forall\vec{\rho}\vec{\varepsilon}.\Psi', p)\}, te) \\
\quad \textbf{in} \\
\quad\quad \textbf{if} \ \mathcal{S}(\Psi) = \Psi \\
\quad\quad\quad \textbf{then} \ (\mathcal{S}, e, \psi) \\
\quad\quad\quad \textbf{else} \\
\quad\quad\quad\quad \textbf{let} \quad (\mathcal{S}', e, \psi) = \mathcal{R}ec(\mathcal{S}(\Psi), \mathcal{S}(EE), \forall\vec{\rho}\vec{\varepsilon}.\mathrm{Lookup}_{\mathcal{S}(\Psi)}(\vec{\varepsilon}), f, te) \\
\quad\quad\quad\quad \textbf{in} \ \ (\mathcal{S}' \circ \mathcal{S}, e, \psi)
\end{array}
$$

$$
\begin{array}{l}
\mathrm{Instantiate}((), \Xi, \Psi) = \{\} \\
\mathrm{Instantiate}((\varepsilon_1.\varphi_1, \ldots, \varepsilon_k.\varphi_k), \forall\varepsilon'_1 \cdots \varepsilon'_k.\Psi_\Xi, \Psi) = \\
\quad \textbf{let} \quad \Phi \text{ is a ground segment of } \{\varepsilon_1.\varphi_1, \ldots, \varepsilon_k.\varphi_k\} \\
\quad\quad\quad \{i_1, \ldots, i_j\} \text{ are the offsets of } \Phi \text{ in } (\varepsilon_1.\varphi_1, \ldots, \varepsilon_k.\varphi_k) \\
\quad\quad\quad \vec{\varepsilon} = \varepsilon'_1 \cdots \varepsilon'_k \setminus\setminus \{i_1, \ldots, i_j\} \quad \varepsilon.\vec{\varphi} = \varepsilon_1.\varphi_1 \cdots \varepsilon_k.\varphi_k \setminus\setminus \{i_1, \ldots, i_j\} \\
\quad \textbf{in} \\
\quad\quad \textbf{if} \ \text{the arrow effects of } \Phi \text{ are cyclic} \\
\quad\quad\quad \textbf{then} \\
\quad\quad\quad\quad \textbf{let} \quad \mathcal{S} = \bigcup_{1 \le p \le j} \left\{ \varepsilon_{i_p} \mapsto \varepsilon_{i_p}.\varphi_{i_p}^{\infty} \right\} \\
\quad\quad\quad\quad\quad\quad S_e = \{\varepsilon'_{i_1} \mapsto \varepsilon_{i_1}.\varphi_{i_1}^{\infty}, \ldots, \varepsilon'_{i_j} \mapsto \varepsilon_{i_j}.\varphi_{i_j}^{\infty}\} \\
\quad\quad\quad\quad\quad \textbf{in} \quad \mathrm{Instantiate}(\varepsilon.\vec{\varphi}, \forall\vec{\varepsilon}.S_e(\Psi_\Xi), \Psi) \circ \mathcal{S} \\
\quad\quad\quad \textbf{else} \\
\quad\quad\quad\quad \textbf{let} \quad \{\varepsilon.\varphi\} = \Phi \quad \{\varepsilon'_{i_1}.\psi'_{i_1}, \ldots, \varepsilon'_{i_j}.\psi'_{i_j}\} \subseteq \Psi_\Xi \\
\quad\quad\quad\quad\quad\quad \psi = \left\lceil \hat{\Psi}(\varepsilon), \psi'_{i_1}, \ldots, \psi'_{i_j} \right\rceil \quad \mathcal{S} = \{\varepsilon \mapsto \varepsilon.\psi \ominus \Psi(\varepsilon)\} \\
\quad\quad\quad\quad\quad\quad S_e = \{\varepsilon'_{i_1} \mapsto \varepsilon.\psi \ominus \psi'_{i_1}, \ldots, \varepsilon'_{i_j} \mapsto \varepsilon.\psi \ominus \psi'_{i_j}\} \\
\quad\quad\quad\quad\quad \textbf{in} \quad \mathrm{Instantiate}(\varepsilon.\vec{\varphi}, \forall\vec{\varepsilon}.S_e(\mathcal{S}(\Psi_\Xi)), \mathcal{S}(\Psi)) \circ \mathcal{S}
\end{array}
$$

As can be seen, the algorithm consists of one case for every construct in the input expression. It is mutually recursive with $\mathcal{R}ec$ and also makes use of an auxiliary subroutine Instantiate.

As we mentioned above, we do not infer types, as the type of an expression can always be reconstructed by taking the plain type of te and loading multiplicities into it from the arrow effect table.

We will now proceed with describing every case of the algorithm separately.

The constant cases are pretty straightforward, and so is the letregion case.

In the application case, the effect on the arrow of the function is added to the inferred effect. But as we infer no types, this effect must be retrieved from $\mathcal{S}_2(\mathcal{S}_1(\Psi))$.

What may seem surprising in the application and if-then-else case is the fact that no unification is done. As we described in the previous section, this is because we need a more eager refinement mechanism, if we want always to have only one version of every arrow effect. In our algorithm, Ψ is updated in the lambda-case and the variable cases.

Example 4.15. Suppose

$$\Psi \models \mu \Rightarrow \mu' \qquad \Psi \models TE \Rightarrow TE'$$

Now consider this input expression

$$te \quad = \quad \text{if } \cdots \text{ then } \cdots \text{ else } \cdots$$

Suppose further that

$$\mathrm{T}(\underline{\Psi}, TE, te) = (\mu, \varphi)$$

An algorithm $\mathcal{M}_W$ would process te by first inferring the most general type for each of the branches and then unifying them.

The way the algorithm $\mathcal{M}$ works, we are sure that the effect table always contains the maximum of all occurences of every arrow effect, which have been processed. So after having processed the three subexpressions of the if-statement, which in addition to an effect and an expression has yielded a substitution $\mathcal{S} = \mathcal{S}_3 \circ \mathcal{S}_2 \circ \mathcal{S}_1$, we need do no unification, but can rely on $\mathcal{S}(\mu')$ being the type of the expression, since the arrow effects of both branches have been taken into account.

The let-case is pretty straightforward. The notation $\mathrm{Lookup}_\Psi(\vec{\varepsilon})$ is a shorthand for $\bigcup_{1\leq i\leq k}\{\varepsilon_i.\hat{\Psi}(\varepsilon_i)\}$. The correctness of the let- and letrec-cases hinges on the fact that the extended environment is also multiplicity-consistent under the effect table. This is what is expressed by the following lemma

Lemma 4.16. *If* $\Psi, EE \models TE \Rightarrow TE'$, $\underline{\Psi} \models \underline{\psi}$ *and* $\Psi \models (\tau, p) \Rightarrow (\tau', p)$ *and* $\mathrm{RegTyEffGen}(TE, \underline{\psi})(\tau) = \forall\vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau$. *then*

$$\Psi, EE + \left\{x \mapsto \left(\forall\vec{\rho}\vec{\varepsilon}.\hat{\Psi}(\vec{\varepsilon}), p\right)\right\} \models TE + \{x \mapsto (\forall\vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau, p)\} \Rightarrow TE' + \{x \mapsto (\forall\vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau', p)\}$$

as well as

$$\Psi, EE + \left\{x \mapsto \left(\forall\vec{\varepsilon}.\hat{\Psi}(\vec{\varepsilon}), p\right)\right\} \models TE + \{x \mapsto (\forall\vec{\alpha}\vec{\varepsilon}.\tau, p)\} \Rightarrow TE' + \{x \mapsto (\forall\vec{\alpha}\vec{\varepsilon}.\tau', p)\}$$

and

$$\Psi, EE + \left\{x \mapsto \left(\forall\vec{\rho}\vec{\varepsilon}.\hat{\Psi}(\vec{\varepsilon}), p\right)\right\} \models TE + \{x \mapsto (\forall\vec{\rho}\vec{\varepsilon}.\tau, p)\} \Rightarrow TE' + \{x \mapsto (\forall\vec{\rho}\vec{\varepsilon}.\tau', p)\}$$

Proof.
Goes along the same lines as that of lemma 3.9

■

The letrec-case is based on the same idea as that of $\mathcal{RE}$ [TT93a]. Here $\mathcal{M}$ delegates the task of finding a fixed point type scheme to $\mathcal{R}ec$, which calls $\mathcal{M}$ iteratively, making the assumption more precise at every iteration, till a fixed point is found. Notice the termination condition of $\mathcal{R}ec$: The function returns when the elaboration of te causes no further refinement.

The lambda-case is one of the interesting cases which can cause Ψ to be updated. First the body of the lambda-abstraction is elaborated. The $\{\}$ with which EE is extended is the empty quantified set of arrow effects: as lambda-bound variables are not polymorphic, all the arrow effects of the type scheme to which the variable is bound must be contained in Ψ.

When the effect is put onto the arrow, Ψ is updated so that it contains the maximum of the newly inferred arrow effect and the one it already had. To make sure that this is indeed the way $\mathcal{S}'$ works, the following lemma can be used

Lemma 4.17. *Let Ψ be a multiplicity-consistent set of arrow effects, and let ε be an effect variable in* $\mathrm{fev}(\Psi)$ *such that* $\underline{\psi} \subseteq \underline{\hat{\Psi}(\varepsilon)}$ *and* $\varepsilon \notin \hat{\Psi}(\varepsilon)$*. Let* $\mathcal{S} = \left\{\varepsilon \mapsto \varepsilon.\psi \ominus \hat{\Psi}(\varepsilon)\right\}$*. It then holds that $\mathcal{S}$ is a multiplicity substitution on Ψ and*

$$\widehat{\mathcal{S}(\Psi)}(\varepsilon) = \left\lceil \psi, \hat{\Psi}(\varepsilon) \right\rceil$$

Proof.
Let $\psi_0 = \hat{\Psi}(\varepsilon)$. We have that

$$\begin{array}{rll} \mathcal{S}(\varepsilon.\psi_0) & = \varepsilon.(\mathcal{S}(\psi_0) \oplus (\psi \ominus \psi_0)) & \\ & = \varepsilon.(\psi_0 \oplus (\psi \ominus \psi_0)) & \varepsilon \notin \underline{\psi_0} \\ & = \varepsilon.\lceil \psi, \psi_0 \rceil & \text{lemma } 3.13 \end{array}$$

That $\mathcal{S}$ is a multiplicity substitution on Ψ is obvious, since $\underline{\lceil \psi, \psi_0 \rceil} = \underline{\psi_0}$.

■

The lambda-case shows the presence of an irritating phenomenon: arrow effects whose principal effect variable occurs in the effect itself. We call these arrow effects *cyclic*, and as they are not covered by lemma 4.17, we just set them to ∞ whenever we encounter them.

Example 4.18. Consider the following source program

$$\begin{array}{l} \lambda f. \\ \quad \textbf{letrec } g = \lambda x. \\ \qquad (\textbf{if } n = 0 \textbf{ then } f \textbf{ else } g)\ x \\ \quad \textbf{in } 1 \end{array}$$

f must be a function. Let ε be the principal effect variable on its arrow. It is clear that ε must be part of the effect of g, since the body of g contains a potential application of f. But then ε must also be included in the effect of f, since it occurs in the same if-then-else expressions as an instance of g. So the arrow effect of the function f will inevitably be cyclic.

The most complicated part of the algorithm is the variable cases. The instantiation of the effect variables can result in the creation of arrow effects whoose multiplicities are greater than those of their counterparts in Ψ. The multiplicity substitution returned by the variable cases can be viewed as the refinement necessary to make room for an instantiation of the type scheme that is multiplicity-consistent under under the effect table. We first instantiate region variables and then the effect variables. The difficult part of instantiation is finding the right effect substitution.

Example 4.19. Define

$$\begin{array}{rl} TE'(x) & = \sigma = \forall \varepsilon_1.\mu \xrightarrow{\varepsilon_1.\{\mathrm{put}(\rho_2) \mapsto 2\}} \mu \\ \Psi & = \{\ \varepsilon_2.\{\mathrm{put}(\rho_2) \mapsto 1, \mathrm{put}(\rho_1) \mapsto 1\}\ \} \end{array}$$

Suppose we are translating the expression $x[\varepsilon_2.\{\mathrm{put}(\rho_2), \mathrm{put}(\rho_1)\}]$. Let us use the following instantiation substitution

$$[\varepsilon_2.\{\mathrm{put}(\rho_2) \mapsto 0, \mathrm{put}(\rho_1) \mapsto 1\}/e1]$$

which would give the following type and place

$$\left(\mu \xrightarrow{\varepsilon_2.\{\mathrm{put}(\rho_2) \mapsto 2, \mathrm{put}(\rho_1) \mapsto 1\}} \mu, p\right)$$

But if the arrow effect of this type and place is to be equal to its counterpart in Ψ, Ψ has to be updated by the following substitution

$$\mathcal{S} = \{\varepsilon_2 \mapsto \varepsilon_2.\{\mathrm{put}(\rho_2) \mapsto 1\}\}$$

As suggested by the example, each arrow effect in σ can be processed by first instantiating it via a substitution which makes it at least as great as its counterpart in Ψ, and then constructing a substitution, which applied to Ψ makes them equal.

This approach is complicated by the fact that the type scheme may contain many arrow effects which can be nested within each other. So if the scheme is to work we must process the arrow effects in a special order. In this connection we use the following definition

Definition 4.20. *Let Φ be a set of arrow effects. We call a subset Φ' of Φ a* ground segment *of Φ, if*

either

1) it consists of exactly one arrow effect $\varepsilon.\varphi$, for which it holds that there is no $\varepsilon'.\varphi' \in \Phi$ such that $\varepsilon' \in \underline{\varphi}$.

or

2) it consists of arrow effects $\{\varepsilon_1.\varphi_1, \ldots, \varepsilon_k.\varphi_k\}$ such that for all $i \in \{1, \ldots, k\}$ it holds that $\varepsilon_i.\varphi_i$ is cyclic,z and if there exists $\varepsilon'.\varphi' \in \Phi$ such that $\varepsilon' \in \underline{\varphi_i}$, then $\varepsilon'.\varphi' \in \Phi'$.

Lemma 4.21. *Every non-empty subset Φ of an effect-consistent set of arrow effects Φ' has a ground segment*

Proof.

The only thing which could prevent Φ from having a ground segment would be if it contained a cycle of arrow effects $\varepsilon_1.\varphi_1, \ldots, \varepsilon_k.\varphi_k, \varepsilon_1.\varphi_1$ such that $\varepsilon_i \in \varphi_{i+1}$, $1 \leq i \leq k$, and $\varepsilon_k \in \varphi_1$, and where there would exist $j \in \{1, \ldots, k\}$ such that $\varepsilon_j \notin \varphi_j$. But this would violate requirement 2) of the definition of effect consistency.

■

Instantiate instantiates effect variables one segment at a time. And as the sub-instantiations it generates thus do not overlap, we get the same result as by instantiating all effect variables simultaneously.

The problem now is how to instantiate effect variables corresponding to one segment of arrow effects. We here distinguish between two cases: the case where the ground segment is one non-cyclic arrow effect, and the case where it is a collection of cyclic arrow effects. The former case is not essentially different from the situation in the lambda-case, except that we must take into account that several effect variables in the type scheme can be instantiated to the same arrow effect. In the latter case we will just set all the arrow effects to ∞, as we will not bother to find a better guess.

When making sure that this way of instantiating type schemes really works, the following lemmas will be necessary

Lemma 4.22. *if* $\Psi, \forall\vec{\rho}\vec{\alpha}\vec{\varepsilon}.\Psi' \models \forall\vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau \Rightarrow \forall\vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau'$ *then*

$$\Psi, \forall\vec{\varepsilon}.[\vec{p}/\vec{\rho}](\Psi') \models \forall\vec{\alpha}\vec{\varepsilon}.[\vec{p}/\vec{\rho}](\tau) \Rightarrow \forall\vec{\alpha}\vec{\varepsilon}.[\vec{p}/\vec{\rho}](\tau')$$

Proof.
One can convince oneself that this lemma holds by appreciating the fact that multiplicity consistency really is a statement about the effect variables in the arrow effects: every effect variable is associated with exactly one effect, and every time it occurs somewhere, it occurs together with that effect. Exchanging some region variables with other places in the effect does not change this property, as long as they are changed everywhere the effect variable occurs.

■

Lemma 4.23. *Suppose* $\Psi, \forall\vec{\varepsilon}.\Psi' \models \forall\vec{\alpha}\vec{\varepsilon}.\tau \Rightarrow \forall\vec{\alpha}\vec{\varepsilon}.\tau'$ *and let* $\varepsilon.\vec{\varphi}$ *be a vector of arrow effects such that* $\mid \varepsilon.\vec{\varphi} \mid = \mid \vec{\varepsilon} \mid$, $\underline{\Psi} \models [\varepsilon.\vec{\varphi}/\vec{\varepsilon}](\tau)$, *and* $\mathrm{aref}(\varepsilon.\vec{\varphi}) \subseteq \underline{\Psi}$. *Then there exist* $\mathcal{S}$ *and* S_e *such that*

$$\mathcal{S} = \mathrm{Instantiate}(\Psi, \forall\vec{\varepsilon}.\Psi', \varepsilon.\vec{\varphi})$$

$$\underline{S_e} = \varepsilon.\vec{\varphi}/\vec{\varepsilon}$$

$$\mathcal{S}(\Psi), \{\} \models \forall\vec{\alpha}.\underline{S_e}(\tau) \Rightarrow \forall\vec{\alpha}.S_e(\mathcal{S}(\tau'))$$

where

$$\mathcal{S} \textit{ is a multiplicity substitution on } \Psi$$

and

$$\mathbf{dom}(\mathcal{S}) = \mathrm{fpev}(\varepsilon.\vec{\varphi})$$

Proof.
se appendix C

■

Lemma 4.24. *if* $\Psi, \{\} \models \forall\vec{\alpha}.\tau \Rightarrow \forall\vec{\alpha}.\tau'$ *and* $\underline{\Psi} \models [\vec{\tau}/\vec{\alpha}](\tau)$ *then there exists* S_t *such that* $\underline{S_t} = [\vec{\tau}/\vec{\alpha}]$ *and*

$$\Psi \models \underline{S_t}(\tau) \Rightarrow S_t(\tau')$$

Proof.
Let $\vec{\tau} = \{\tau_1, \ldots, \tau_k\}$ and $\vec{\alpha} = \{\alpha_1, \ldots, \alpha_k\}$. By lemma 4.7 we know that there exist unique $\tau'_1, \ldots, \tau'_k$ such that

$$\Psi \models \tau_1 \Rightarrow \tau'_1$$
$$\Psi \models \tau_2 \Rightarrow \tau'_2$$
$$\vdots$$
$$\Psi \models \tau_k \Rightarrow \tau'_k$$

The substitution we are looking for is

$$S_t = \{\alpha_1 \mapsto \tau'_1, \ldots, \alpha_k \mapsto \tau'_k\}$$

■

4.4 Soundness

We now present the proof of the soundness of algorithm $\mathcal{M}$:

Soundness and termination. *If* $\Psi, EE \models TE \Rightarrow TE'$, $\Psi \models \mu \Rightarrow \mu'$, *and* $T(\underline{\Psi}, TE, te) = (\mu, \varphi)$, *then there exist* $\mathcal{S}$, e, *and* ψ *such that*

$$(\mathcal{S}, e, \psi) = \mathcal{M}(\Psi, EE, te)$$

where

$$\mathcal{S} \textit{ is a multiplicity substitution on } \Psi$$

and

$$\mathcal{S}(TE') \vdash te \hookrightarrow e : \mathcal{S}(\mu'), \psi$$

Proof.
The proof is by structural induction on te. Suppose

$$\Psi, EE \models TE \Rightarrow TE' \tag{0}$$

$$\Psi \models \mu \Rightarrow \mu' \tag{1}$$

and

$$\mathrm{T}(\underline{\Psi}, TE, te) = (\mu, \varphi) \tag{2}$$

The proof now proceeds with a case analysis.

`true at` p

We have that $\mathcal{M}$ terminates and

$$(\{\}, \texttt{true at } \rho, \{\mathrm{put}(\rho) \mapsto 1\}) = \mathcal{M}(\Psi, EE, \texttt{true at } p) \quad \boxed{3}$$

It vacously holds that

$$\{\} \text{ is a multiplicity substitution on } \Psi \quad \boxed{4}$$

and the following typing holds

$$TE' \vdash \texttt{true at } p \hookrightarrow \texttt{true at } p : (bool, p), \{\mathrm{put}(p) \mapsto 1\} \quad \boxed{5}$$

`false at` p

This case is similar to `true at` p.

$f[\vec{p}, \vec{\tau}, \varepsilon.\vec{\varphi}]$ `at` p

Here (2) looks like this

$$\mathrm{T}(\underline{\Psi}, TE, f[\vec{p},\vec{\tau},\varepsilon.\vec{\varphi}]\ \texttt{at}\ p) = ((\tau_{\mathrm{T}}, p)\,, \{\mathrm{put}(p), \mathrm{get}(p')\}) \tag{3}$$

where

$$\tau_{\mathrm{T}} = (\vec{p}/\vec{\rho}, \vec{\tau}/\vec{\alpha}, \varepsilon.\vec{\varphi}/\vec{\varepsilon})(\tau) \tag{4}$$

By the definition of T we know that the following premises must hold

$$(\forall\vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau, p') = TE(f) \tag{5}$$

$$|\,\vec{p}\,| = |\,\vec{\rho}\,| \quad |\,\vec{\tau}\,| = |\,\vec{\alpha}\,| \quad |\,\varepsilon.\vec{\varphi}\,| = |\,\vec{\varepsilon}\,| \tag{6}$$

$$\mathrm{aref}(\varepsilon.\vec{\varphi}) \in \underline{\Psi} \tag{7}$$

$$\underline{\Psi} \models \tau_{\mathrm{T}} \tag{8}$$

In this case (1) looks like this

$$\Psi \models (\tau_{\mathrm{T}}, p) \Rightarrow (\tau'_{\mathrm{T}}, p) \tag{9}$$

(0) implies that

$$\Psi, \forall\vec{\rho}\vec{\varepsilon}.\Psi' \models \forall\vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau \Rightarrow \forall\vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau' \tag{10}$$

where

$$(\forall\vec{\rho}\vec{\varepsilon}.\Psi', p') = EE(f) \quad (\forall\vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau', p') = TE'(f) \tag{11}$$

We pick $\vec{\varepsilon}$ and $\vec{\rho}$ so that

$$\mathrm{fev}(\forall\vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau) \subseteq \mathrm{fev}(\Psi) \tag{12}$$

$$\mathrm{fev}(\vec{\varepsilon}) \cap \mathrm{fev}(\Psi) = \{\} \quad \mathrm{frv}(\vec{\rho}) \cap \mathrm{frv}(\Psi) = \{\} \tag{13}$$

By lemma 4.22 on (10) we get

$$\Psi, \forall\vec{\varepsilon}.\ [\vec{p}/\vec{\rho}](\Psi') \models \forall\vec{\alpha}\vec{\varepsilon}.\ [\vec{p}/\vec{\rho}](\tau) \Rightarrow \forall\vec{\alpha}\vec{\varepsilon}.\ [\vec{p}/\vec{\rho}](\tau') \tag{14}$$

By lemma 3.17 on (8) and (4) we get that

$$\Phi \models [\vec{\tau}/\vec{\alpha}]([\varepsilon.\vec{\varphi}/\vec{\varepsilon}]([\vec{p}/\vec{\rho}](\tau))) \tag{15}$$

And by lemma 3.8 on this we get that

$$\Phi \models [\varepsilon.\vec{\varphi}/\vec{\varepsilon}]([\vec{p}/\vec{\rho}](\tau)) \tag{16}$$

Now by lemma 4.23 on (14), (6), (16), and (7) we get that there exist $\mathcal{S}$ and S_e such that

$$\mathcal{S} = \mathrm{Instantiate}(\Psi, \forall\vec{\varepsilon}.[\vec{p}/\vec{\rho}](\Psi'), \varepsilon.\vec{\varphi}) \tag{17}$$

$$\underline{S_e} = \varepsilon.\vec{\varphi}/\vec{\varepsilon} \tag{18}$$

$$\mathcal{S}(\Psi), \{\} \models \forall\vec{\alpha}.\underline{S_e}([\vec{p}/\vec{\rho}](\tau)) \Rightarrow \forall\vec{\alpha}.S_e(\mathcal{S}([\vec{p}/\vec{\rho}](\tau'))) \tag{19}$$

where

$$\mathcal{S} \text{ is a multiplicity substitution on } \Psi \qquad \boxed{20}$$

and

$$\mathbf{dom}(\mathcal{S}) = \mathrm{fpev}(\varepsilon.\vec{\varphi}) \tag{21}$$

From (11) and (12) we get that $\mathcal{M}$ terminates and

$$\mathcal{M}(\Psi, EE, f[\vec{p},\vec{\tau},\varepsilon.\vec{\varphi}]\ \texttt{at}\ p) = (\mathcal{S}, f[\vec{p}]\ \texttt{at}\ p, \{\mathrm{put}(p) \mapsto 1, \mathrm{get}(p') \mapsto 1\}) \qquad \boxed{17}$$

We have now only left to prove that $\mathcal{M}$ has found a real typing. By lemma 4.24 on (14) and (15) we get that there exists S_t such that

$$\mathcal{S}(\Psi) \models \underline{S_t}(\underline{S_e}([\vec{p}/\vec{\rho}](\tau))) \Rightarrow S_t(S_e(\mathcal{S}([\vec{p}/\vec{\rho}](\tau')))) \tag{18}$$

and

$$\underline{S_t} = [\vec{\tau}/\vec{\alpha}] \tag{19}$$

By (13), (7), and (16) we know that

$$\mathbf{dom}(\mathcal{S}) \cap \vec{\varepsilon} = \{\} \tag{20}$$

Furthermore, as $\mathcal{S}$ is a multiplicity substitution on Ψ, and as it, by (7) and (16), holds that $\mathbf{dom}(\mathcal{S}) \subseteq \mathrm{fev}(\Psi)$, we have that $\mathrm{fev}(\mathbf{rng}(\mathcal{S})) \subseteq \mathrm{fev}(\Psi)$ and $\mathrm{frv}(\mathbf{rng}(\mathcal{S})) \subseteq \mathrm{frv}(\Psi)$, as $\mathcal{S}$ would otherwise introduce new atomic effects to Ψ. Thus, by (13) it holds that

$$\mathrm{fev}(\mathbf{rng}(\mathcal{S})) \cap \vec{\varepsilon} = \{\} \qquad \mathrm{frv}(\mathbf{rng}(\mathcal{S})) \cap \vec{\rho} = \{\} \tag{21}$$

We have that

$$\begin{aligned} \mathcal{S}(TE')(x) &= \mathcal{S}(\forall \vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau') \\ &= \forall \vec{\rho}\vec{\alpha}\vec{\varepsilon}.\mathcal{S}(\tau') \quad (20) \text{ and } (21) \end{aligned} \tag{22}$$

Now set $S = ([\vec{p}/\vec{\rho}], S_t, S_e)$. By (13) and (19) we get that

$$\underline{S} = ([\vec{p}/\vec{\rho}], [\vec{\tau}/\vec{\alpha}], [\varepsilon.\vec{\varphi}/\vec{\varepsilon}]) \tag{23}$$

By the VARCOMPOUND-rule on (22), (23) we get that

$$\begin{aligned} S(TE') \vdash f[\vec{p}, \vec{\tau}, \varepsilon.\vec{\varphi}] \ \texttt{at} \ p \hookrightarrow \ & f[\vec{p}] \ \texttt{at} \ p : \\ & (S(\mathcal{S}(\tau')), p), \\ & \{\mathrm{put}(p) \mapsto 1, \mathrm{get}(p') \mapsto 1\} \end{aligned} \tag{24}$$

We have now only left to show that $(S(\mathcal{S}(\tau')), p) = \mathcal{S}((\tau'_{\mathrm{T}}, p))$.

First, we have that

$$\begin{aligned} \underline{S_t}(\underline{S_e}([\vec{p}/\vec{\rho}](\tau))) &= ([\vec{p}/\vec{\rho}], \underline{S_t}, \underline{S_e})(\tau) && \text{lemma 3.17} \\ &= ([\vec{p}/\vec{\rho}], [\vec{\tau}/\vec{\alpha}], [\varepsilon.\vec{\varphi}/\vec{\varepsilon}])(\tau) && (13) \text{ and } (19) \\ &= \tau_{\mathrm{T}} \end{aligned} \tag{25}$$

So (18) can be rewritten like this

$$\mathcal{S}(\Psi) \models \tau_{\mathrm{T}} \Rightarrow S_t(S_e(\mathcal{S}([\vec{p}/\vec{\rho}](\tau')))) \tag{26}$$

By lemma 4.11 on (15) and (9) we get

$$\mathcal{S}(\Psi) \models \tau_{\mathrm{T}} \Rightarrow \mathcal{S}(\tau'_{\mathrm{T}}) \tag{27}$$

By lemma 4.7 on (26) and (27) we get

$$\mathcal{S}(\tau'_{\mathrm{T}}) = S_t(S_e(\mathcal{S}([\vec{p}/\vec{\rho}](\tau')))) \tag{28}$$

Now it holds that

$$\begin{aligned} S(\mathcal{S}(\tau')) &= ([\vec{p}/\vec{\rho}], S_t, S_e)(\mathcal{S}(\tau')) \\ &= S_t(S_e([\vec{p}/\vec{\rho}](\mathcal{S}(\tau')))) && \text{lemma 3.17} \\ &= S_t(S_e(([\vec{p}/\vec{\rho}] \mid \mathcal{S})(\tau'))) && \text{lemma 3.16 on (16) and (21)} \\ &= S_t(S_e((\mathcal{S} \mid [\vec{p}/\vec{\rho}])(\tau'))) \\ &= S_t(S_e(\mathcal{S}([\vec{p}/\vec{\rho}](\tau')))) && \text{lemma 3.16 on (16)} \\ &= \mathcal{S}(\tau'_{\mathrm{T}}) && (28) \end{aligned} \tag{29}$$

So by (29) and the fact that $\mathcal{S}(p) = p$, we can rewrite (24) to get the desired

$$\begin{array}{l} \mathcal{S}(TE') \vdash f[\vec{p},\vec{\tau},\varepsilon.\vec{\varphi}] \texttt{ at } p \hookrightarrow f[\vec{p}] \texttt{ at } p : \\ \qquad\qquad\qquad\qquad\quad \mathcal{S}((\backslash taut', p)), \\ \qquad\qquad\qquad\qquad\quad \{\mathrm{put}(p) \mapsto 1, \mathrm{get}(p') \mapsto 1\} \end{array} \qquad \boxed{30}$$

$\boxed{x[\vec{\tau},\varepsilon.\vec{\varphi}]}$

This case is identical to $\boxed{f[\vec{p},\vec{\tau},\varepsilon.\vec{\varphi}] \texttt{ at } p}$, except that there is no quantified region variables, which only makes matters simpler.

$\boxed{\lambda : \mu_1 \xrightarrow{\varepsilon.\varphi} \mu_2 \;\; x.te \texttt{ at } p}$

Here (2) looks like this

$$\mathrm{T}(\underline{\Psi}, TE, \lambda : \mu_1 \xrightarrow{\varepsilon.\varphi} \mu_2 \; x.te \texttt{ at } p) = (\left(\mu_1 \xrightarrow{\varepsilon.\varphi} \mu_2, p\right), \{\mathrm{put}(p)\}) \tag{3}$$

This be returned on the following premises

$$\underline{\Psi} \models \mu_1 \tag{4}$$

$$\varepsilon.\varphi \in \underline{\Psi} \tag{5}$$

$$\mathrm{T}(\underline{\Psi}, TE + \{x \mapsto \mu_1\}, te) = (\mu_2, \varphi_1) \tag{6}$$

$$\varphi_1 \subseteq \varphi \tag{7}$$

In this case (1) looks like this

$$\Psi \models \left(\mu_1 \xrightarrow{\varepsilon.\varphi} \mu_2, p\right) \Rightarrow \left(\mu_1' \xrightarrow{\varepsilon.\psi_0} \mu_2', p\right) \tag{8}$$

By the definition 4.5 this implies that

$$\Psi \models \mu_2 \Rightarrow \mu_2' \tag{9}$$

and

$$\Psi \models \mu_1 \Rightarrow \mu_1' \tag{10}$$

Using empty quantifications and the empty quantified set of arrow effects, (10) gives us

$$\Psi, (\{\}, p) \models \mu_1 \Rightarrow \mu_1' \tag{11}$$

where μ_1 and μ_1' here are type schemes with empty quantifiers. This and (0) give us that

$$\Psi, EE + \{x \mapsto (\{\}, p)\} \models TE + \{x \mapsto \mu_1\} \Rightarrow TE' + \{x \mapsto \mu_1'\} \tag{12}$$

So by induction of (12), (9), and (6) we get that there exist $\mathcal{S}$, e, and ψ such that

$$(\mathcal{S}, e, \psi) = \mathcal{M}(\Psi, EE + \{x \mapsto (\{\}, p)\}, te) \tag{13}$$

where

$$\mathcal{S} \text{ is a multiplicity substitution on } \Psi \tag{14}$$

and

$$\mathcal{S}(TE' + \{x \mapsto \mu_1'\}) \vdash te \hookrightarrow e : \mathcal{S}(\mu_2'), \psi \tag{15}$$

Let

$$\mathcal{S}' = \quad \textbf{if} \ \ \varepsilon \in \mathrm{fev}(\psi) \ \textbf{then} \ \{\varepsilon \mapsto \varepsilon.\varphi^{\infty}\} \ \textbf{else} \ \left\{\varepsilon \mapsto \varepsilon.(\psi \ominus \widehat{\mathcal{S}(\Psi)}(\varepsilon))\right\} \tag{13}$$

By (10) and (13) we have that $\mathcal{M}$ terminates and that

$$(\mathcal{S}' \circ \mathcal{S}, \lambda x.e \ \texttt{at} \ p, \{\mathrm{put}(\rho) \mapsto 1\}) = \mathcal{M}(\Psi, EE, \lambda : \mu_1 \xrightarrow{\varepsilon.\varphi} \mu_2 \ x.te \ \texttt{at} \ p) \qquad \boxed{14}$$

Now we have that

$$\mathcal{S}' \text{ is a multiplicity substitution on } \mathcal{S}(\Psi) \tag{15}$$

and

$$\begin{aligned} \mathcal{S}'(\widehat{\mathcal{S}(\Psi)})(\varepsilon) \ &= \left\lceil \psi, \widehat{\mathcal{S}(\Psi)}(\varepsilon) \right\rceil \qquad \text{if } \varepsilon \notin \mathrm{fev}(\psi) \\ \mathcal{S}'(\widehat{\mathcal{S}(\Psi)})(\varepsilon) \ &= \varphi^{\infty} \qquad\qquad\qquad\quad \text{if } \varepsilon \in \mathrm{fev}(\psi) \end{aligned} \tag{16}$$

To convince ourselves that this is true, let us distinguish between two cases

$\varepsilon \notin \mathrm{fev}(\psi)$. In this case we get (15) and (16) by lemma 4.17 on (5), (7), and (11).

$\varepsilon \in \mathrm{fev}(\psi)$. Here (15) follows from the fact that $\underline{\varphi^{\infty}} = \varphi = \underline{\Phi(\varepsilon)} = \underline{\mathcal{S}(\Phi(\varepsilon))}$. That (16) holds is obvious.

By (11) and (15) we get

$$\mathcal{S}' \circ \mathcal{S} \text{ is a multiplicity substitution on } \Psi \qquad \boxed{17}$$

By lemma 4.11 on (0) and (11) we get

$$\mathcal{S}(\Psi) \models TE \Rightarrow \mathcal{S}(TE') \tag{18}$$

By lemma 4.11 on (9) and (11) we get

$$\mathcal{S}(\Psi) \models \mu_2 \Rightarrow \mathcal{S}(\mu_2') \tag{19}$$

By lemma 3.11 on 6 we get that

$$\underline{\Psi} \models \varphi_1 \tag{20}$$

By theorem 3.26 on 6 and (12) we get

$$\underline{\psi} = \varphi_1 \tag{21}$$

So from (21) and (20) we get

$$\Psi \models \psi \tag{22}$$

By lemma 4.14 on (18), (19), (22), (11), and (12) we get that

$$\mathcal{S}'(\mathcal{S}(TE' + \{x \mapsto \mu_1'\})) \vdash e \hookrightarrow te : \mathcal{S}'(\mathcal{S}(\mu_2)), \mathcal{S}'(\psi) \tag{23}$$

Now let

$$\psi'' = \mathcal{S}'(\widehat{\mathcal{S}(\Psi)})(\varepsilon) \qquad \psi' = \psi'' \ominus \mathcal{S}'(\psi) \tag{24}$$

We have that

$$\psi'' = \mathcal{S}'(\psi) \oplus \psi' \tag{25}$$

To convince ourselves that this is so, let us distinguish between two cases

$\varepsilon \in \mathrm{fev}(\psi)$. In this case $\psi'' = \varphi^{\infty}$, so (25) obviously holds.

$\varepsilon \notin \mathrm{fev}(\psi)$. In this case we have

$$\begin{aligned} \mathcal{S}'(\psi) \oplus \psi' \ &= \mathcal{S}'(\psi) \oplus (\psi'' \ominus \mathcal{S}'(\psi)) \\ &= \psi \oplus (\psi'' \ominus \psi) && \varepsilon \notin \mathrm{fev}(\psi) \\ &= \lceil \psi, \psi'' \rceil && \text{lemma 3.13} \\ &= \psi'' && (16) \end{aligned} \tag{26}$$

We have that

$$\begin{aligned}\underline{\psi''} &= \underline{\mathcal{S}'(\widehat{\mathcal{S}(\Psi)})(\varepsilon)} \\ &= \underline{\hat{\Psi}(\varepsilon)} \qquad (17) \\ &= \varphi \qquad (5)\end{aligned} \tag{27}$$

It is easy (using lemma 4.11 a number of times and the uniqueness part of lemma 4.7) to show that

$$\mathcal{S}'(\mathcal{S}(\mu_1')) \xrightarrow{\varepsilon.\psi''} \mathcal{S}'(\mathcal{S}(\mu_2')) = \mathcal{S}'(\mathcal{S}(\mu_1' \xrightarrow{\varepsilon.\psi_0} \mu_2')) \tag{28}$$

By the LAM-rule on (23), (25), (27), and (28) we get

$$\mathcal{S}'(\mathcal{S}(TE)) \vdash \lambda{:}\mu_1 \xrightarrow{\varepsilon.\varphi} \mu_2\ x.te\ \texttt{at}\ p \hookrightarrow \lambda x.e\ \texttt{at}\ p : \left(\mathcal{S}'(\mathcal{S}(\mu_1' \xrightarrow{\varepsilon.\psi_0} \mu_2')), p\right), \{\mathrm{put}(p) \mapsto 1\} \qquad \boxed{29}$$

$\boxed{te_1\ te_2 : p, \varepsilon}$

Here (2) looks like this

$$\mathrm{T}(\underline{\Psi}, TE, te_1\ te_2 : p, \varepsilon) = (\mu_1, \varphi \cup \varphi_1 \cup \varphi_2 \cup \{\mathrm{get}(p)\} \cup \{\varepsilon\}) \tag{3}$$

This be returned on the following premises

$$\mathrm{T}(\underline{\Psi}, TE, te_1) = \left(\left(\mu_2 \xrightarrow{\varepsilon.\varphi} \mu_1, p\right), \varphi_1\right) \tag{4}$$

$$\mathrm{T}(\underline{\Psi}, TE, te_2) = (\mu_2, \varphi_2) \tag{5}$$

By lemma 3.11 on (4) we get that

$$\underline{\Psi} \models \mu_2 \xrightarrow{\varepsilon.\varphi} \mu_1 \tag{6}$$

And by lemma 4.7 there exists $\mu_2' \xrightarrow{\varepsilon.\psi_0} \mu_1'$ such that

$$\Psi \models \left(\mu_2 \xrightarrow{\varepsilon.\varphi} \mu_1, p\right) \Rightarrow \left(\mu_2' \xrightarrow{\varepsilon.\psi_0} \mu_1', p\right) \tag{7}$$

Now by induction on (0), (7), and (4) we get that there exist $\mathcal{S}_1$, e_1, and ψ_1 such that

$$(\mathcal{S}_1, e_1, \psi_1) = \mathcal{M}(\Psi, EE, te_1) \tag{8}$$

where

$$\mathcal{S}_1 \text{ is a multiplicity substitution on } \Psi \tag{9}$$

and

$$\mathcal{S}_1(TE') \vdash te_1 \hookrightarrow e_1 : \mathcal{S}_1(\left(\mu_2' \xrightarrow{\varepsilon.\psi_0} \mu_1', p\right)), \psi_1 \tag{10}$$

By lemma 4.11 on (6) and (0) we get

$$\mathcal{S}_1(\Psi), \mathcal{S}_1(EE) \models TE \Rightarrow \mathcal{S}_1(TE') \tag{8}$$

By lemma 4.11 on (6) and (1) we get

$$\mathcal{S}_1(\Psi) \models \mu_2 \Rightarrow \mathcal{S}_1(\mu_2') \tag{9}$$

And by induction on (8), (9), and (5) we get that there exist $\mathcal{S}_2$, e_2, and ψ_2 such that

$$(\mathcal{S}_2, e_2, \psi_2) = \mathcal{M}(\mathcal{S}_1(\Psi), \mathcal{S}_1(EE), te_2) \tag{10}$$

where

$$\mathcal{S}_2 \text{ is a multiplicity substitution on } \mathcal{S}_1(\Psi) \tag{11}$$

and

$$\mathcal{S}_2(\mathcal{S}_1(TE')) \vdash te_2 \hookrightarrow e_2 : \mathcal{S}_2(\mathcal{S}_1(\mu_2')), \psi_2 \tag{12}$$

We know from (7) that $\varepsilon.\varphi \in \underline{\Psi}$. Let $\psi = \mathcal{S}_2(\widehat{\mathcal{S}_1}(\Psi))(\varepsilon)$. From (5) and (7) we get that $\mathcal{M}$ terminates and that

$$(\mathcal{S}_2 \circ \mathcal{S}_1, e_1\ e_2, \psi_2 \oplus \mathcal{S}_2(\psi_1) \oplus \{\varepsilon \mapsto 1\} \oplus \psi \oplus \{\mathrm{get}(\rho) \mapsto 1\}) = \mathcal{M}(\Psi, EE, te_1\ te_2 : p, \varepsilon) \qquad \boxed{10}$$

(6) and (8) give us that

$$\mathcal{S}_2 \circ \mathcal{S}_1 \text{ is a multiplicity substitution on } \Psi \tag{11}$$

By lemma 3.11 on 4 we get that

$$\underline{\Psi} \models \varphi_1 \tag{12}$$

By theorem 3.26 on 4 and (7) we get

$$\underline{\psi_1} = \varphi_1 \tag{13}$$

and from (13) and (12) we get

$$\Psi \models \psi_1 \tag{14}$$

It is straightforward to show that

$$\mathcal{S}_2\mathcal{S}_1(\mu_2' \xrightarrow{\varepsilon.\psi_0} \mu_1') = \mathcal{S}_2(\mathcal{S}_1(\mu_2')) \xrightarrow{\varepsilon.\psi} \mathcal{S}_2(\mathcal{S}_1(\mu_1')) \tag{15}$$

Now by this and lemma 4.14 (8), (9), (14) on (7) we get

$$\mathcal{S}_2(\mathcal{S}_1(TE')) \vdash te_1 \hookrightarrow e_1 : \mathcal{S}_2(\mathcal{S}_1(\mu_2')) \xrightarrow{\varepsilon.\psi} \mathcal{S}_2(\mathcal{S}_1(\mu_1')), \mathcal{S}_2(\psi_1) \tag{16}$$

Finally, by the APPL-rule on (16) and (9) we get the desired

$$\mathcal{S}_2(\mathcal{S}_1(TE')) \vdash te_1\ te_2 : p, \varepsilon \hookrightarrow e_1\ e_2 : \mathcal{S}_2(\mathcal{S}_1(\mu_2')), \psi_2 \oplus \mathcal{S}_2(\psi_1) \oplus \{\varepsilon \mapsto 1\} \oplus \psi \oplus \{\mathrm{get}(\rho) \mapsto 1\} \qquad \boxed{17}$$

`if` $te_1 : p$ `then` te_2 `else` te_3

The proof of this case goes along the same lines as that of $te_1\ te_2 : p, \varepsilon$.

`letrec` $f{:}\forall\vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau$ `at` $p = te_1$ `in` te_2

Here (2) looks like this

$$\mathrm{T}(\underline{\Psi}, TE, \texttt{letrec}\ f{:}\forall\vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau\ \texttt{at}\ p = te_1\ \texttt{in}\ te_2) = (\mu_2, \varphi_1 \cup \varphi_2) \tag{3}$$

This be returned on the following premises

$$\underline{\Psi} \models \forall\vec{\rho}\vec{\varepsilon}\vec{\alpha}.\tau \tag{4}$$

$$\mathrm{T}(\underline{\Psi}, TE + \{f \mapsto (\forall\vec{\rho}\vec{\varepsilon}.\tau, p)\}, te_1) = ((\tau, p), \varphi_1) \tag{5}$$

$$\forall\vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau = \mathrm{RegTyEffGen}(TE, \varphi_1)(\tau) \tag{6}$$

$$\mathrm{T}(\underline{\Psi}, TE + \{x \mapsto (\forall\vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau, p)\}, te_2) = (\mu_2, \varphi_2) \tag{7}$$

In this case (1) looks like this

$$\Psi \models \mu_2 \Rightarrow \mu_2' \tag{8}$$

By lemma 3.11 on (5) we know that

$$\underline{\Psi} \models \tau, \varphi_1 \tag{9}$$

By lemma 4.7 on (9) and the multiplicity consistency of Ψ we get that there exists exactly one τ' such that

$$\Psi \models \tau \Rightarrow \tau' \tag{10}$$

and by definition 4.5 it then also holds that

$$\Psi \models (\tau, p) \Rightarrow (\tau', p) \tag{11}$$

Now let us now study the following sequence of substitutions

$$\mathcal{S}^0, \mathcal{S}^1, \mathcal{S}^2, \ldots \tag{12}$$

which is defined by the following recursive equations

$$\begin{aligned} \mathcal{S}^{i+1} &= \mathcal{S} \circ \mathcal{S}^i \\ &\quad \text{where } (\mathcal{S}, \_, \_) = \mathcal{M}(\mathcal{S}^i(\Psi), \mathcal{S}^i(EE) + \left\{f \mapsto \left(\forall\vec{\rho}\vec{\varepsilon}.\mathrm{Lookup}_{\mathcal{S}^i(\Psi)}(\vec{\varepsilon}), p\right)\right\}, te_1) \\ \mathcal{S}^0 &= \{\} \end{aligned} \tag{13}$$

That the sequence is well-defined follows from the following fact:

For all $i \geq 0$ it holds that

$$\mathcal{S}^i \textit{ is defined} \tag{14}$$

$$\mathcal{S}^i \textit{ is a multiplicity substitution on } \Psi \tag{15}$$

This can be proven by an inner induction on i.

$i = 0$. $\mathcal{S}_0 = \{\}$, which is a multiplicity substitution on Ψ.
$i \geq 1$. By inner induction we know that

$$\mathcal{S}^{i-1} \text{ is defined} \tag{16}$$

$$\mathcal{S}^{i-1} \text{ is a multiplicity substitution on } \Psi \tag{17}$$

By lemma 4.11 on (17) and (11) we get that

$$\mathcal{S}^{i-1}(\Psi) \models (\tau, p) \Rightarrow \left(\mathcal{S}^{i-1}(\tau'), p\right) \tag{18}$$

By lemma 4.11 on (17) and (0) we get that

$$\mathcal{S}^{i-1}(\Psi), \mathcal{S}^{i-1}(EE) \models TE \Rightarrow \mathcal{S}^{i-1}(TE') \tag{19}$$

By (17) and (9) we have that

$$\underline{\mathcal{S}^{i-1}(\Psi)} \models \varphi_1 \tag{20}$$

Now by lemma 4.16 on (19), (20), (18), and (6) we get

$$\begin{aligned}\mathcal{S}^{i-1}(\Psi),\ \mathcal{S}^{i-1}(EE) + \left\{f \mapsto \left(\forall\vec{\rho}\vec{\varepsilon}.\mathrm{Lookup}_{\mathcal{S}^{i-1}(\Psi)}(\vec{\varepsilon}), p\right)\right\} \\ \models TE + \{f \mapsto (\forall\vec{\rho}\vec{\varepsilon}.\tau, p)\} \\ \Rightarrow \mathcal{S}^{i-1}(TE') + \left\{f \mapsto \left(\forall\vec{\rho}\vec{\varepsilon}.\mathcal{S}^{i-1}(\tau'), p\right)\right\}\end{aligned} \tag{21}$$

So by outer induction on (21), (18), and (5), we know that there exist $\mathcal{S}$, e, and ψ such that

$$(\mathcal{S}, e, \psi) = \mathcal{M}(\mathcal{S}^{i-1}(\Psi), \mathcal{S}^{i-1}(EE) + \left\{f \mapsto \left(\forall\vec{\rho}\vec{\varepsilon}.\mathrm{Lookup}_{\mathcal{S}^{i-1}(\Psi)}(\vec{\varepsilon}), p\right)\right\}, te_1) \tag{22}$$

where

$$\mathcal{S} \text{ is a multiplicity substitution on } \mathcal{S}^{i-1}(\Psi) \tag{23}$$

By (13), (16) and (22) we get that

$$\mathcal{S}^i \text{ is defined} \tag{24}$$

and, finally, by (17) and (23) we have that

$$\mathcal{S}^i \text{ is a multiplicity substitution on } \Psi \tag{25}$$

$\mathcal{S}^i$ is the substitution $\mathcal{R}ec(\Psi, EE, (\forall\vec{\rho}\vec{\varepsilon}.\mathrm{Lookup}_{\Psi}(\vec{\varepsilon}), p), f, te_1)$ gets from $\mathcal{M}$ at its i'th recursive call. We now need to convince ourselves that $\mathcal{R}ec$ terminates, i.e. that there exists k such that

$$\mathcal{S}^{k+1}(\Psi) = \mathcal{S}^k(\Psi) \tag{26}$$

By lemma 3.25 on (13) and (15) we have that

$$\Psi \leq \mathcal{S}^1(\Psi) \leq \mathcal{S}^2(\Psi) \leq \cdots \tag{27}$$

So by lemma 3.14 on this we get that there exists k such that (26) holds. Let p be the smallest number such that

$$\mathcal{S}^{p+1}(\Psi) = \mathcal{S}^p(\Psi) \tag{28}$$

We know that

$$\mathcal{S}^p \text{ is defined} \tag{29}$$

$$\mathcal{S}^p \text{ is a multiplicity substitution on } \Psi \tag{30}$$

By lemma 4.11 on (30) and (11) we get that

$$\mathcal{S}^p(\Psi) \models (\tau, p) \Rightarrow (\mathcal{S}^p(\tau'), p) \tag{31}$$

By lemma 4.11 on (30) and (0) we get that

$$\mathcal{S}^p(\Psi), \mathcal{S}^p(EE) \models TE \Rightarrow \mathcal{S}^p(TE') \tag{32}$$

By (30) and (9) we have that

$$\underline{\mathcal{S}^p(\Psi)} \models \varphi_1 \tag{33}$$

Now by lemma 4.16 on (32), (33), (31), and (6) we get

$$\begin{aligned}\mathcal{S}^p(\Psi),\ \mathcal{S}^p(EE) + \left\{f \mapsto \left(\forall\vec{\rho}\vec{\varepsilon}.\mathrm{Lookup}_{\mathcal{S}^p(\Psi)}(\vec{\varepsilon}), p\right)\right\} \\ \models TE + \{f \mapsto (\forall\vec{\rho}\vec{\varepsilon}.\tau, p)\} \\ \Rightarrow \mathcal{S}^p(TE') + \{f \mapsto (\forall\vec{\rho}\vec{\varepsilon}.\mathcal{S}^p(\tau'), p)\}\end{aligned} \tag{34}$$

We are now ready to use induction for $\mathcal{R}ec$'s last call of $\mathcal{M}$, and this we do on the premises (34), (31), and (5). This gives us that there exist $\mathcal{S}$, e_1, and ψ_1 such that

$$(\mathcal{S}, e_1, \psi_1) = \mathcal{M}(\mathcal{S}^p(\Psi), \mathcal{S}^p(EE) + \left\{f \mapsto \left(\forall\vec{\rho}\vec{\varepsilon}.\text{Lookup}_{\mathcal{S}^p(\Psi)}(\vec{\varepsilon}), p\right)\right\}, te_1) \tag{35}$$

where

$$\mathcal{S} \text{ is a multiplicity substitution on } \mathcal{S}^p(\Psi) \tag{36}$$

and

$$\mathcal{S}(\mathcal{S}^p(TE') + \{f \mapsto (\forall\vec{\rho}\vec{\varepsilon}.\mathcal{S}^p(\tau'), p)\}) \vdash te_1 \hookrightarrow e_1 : \mathcal{S}(\mathcal{S}^p(\tau'), p), \psi_1 \tag{37}$$

As we pointed out at (28), $\mathcal{R}ec(\Psi, EE, (\forall\vec{\rho}\vec{\varepsilon}.\text{Lookup}_\Psi(\vec{\varepsilon}), p), f, te_1)$ now terminates and

$$(\mathcal{S}_1, e_1, \psi_1) = \mathcal{R}ec(\Psi, EE, (\forall\vec{\rho}\vec{\varepsilon}.\text{Lookup}_\Psi(\vec{\varepsilon}), p), f, te_1) \tag{35}$$

where

$$\mathcal{S}_1 = \mathcal{S} \circ \mathcal{S}^p = \mathcal{S}^{p+1} \tag{36}$$

By (33) and (30) we have that

$$\mathcal{S}_1 \text{ is a multiplicity substitution on } \Psi \tag{37}$$

By lemma 4.11 on (31) and (33) we get that

$$\mathcal{S}(\mathcal{S}^p(\Psi)) \models (\tau, p) \Rightarrow (\mathcal{S}(\mathcal{S}^p(\tau')), p) \tag{38}$$

We have that

$$\begin{array}{lll} \mathcal{S}(\mathcal{S}^p(\Psi)) & = \mathcal{S}^{p+1}(\Psi) & (13), (32) \\ & = \mathcal{S}^p(\Psi) & (28) \end{array} \tag{39}$$

So by lemma 4.7 on this, (31), and (38) we get

$$\mathcal{S}(\mathcal{S}^p(\tau')) = \mathcal{S}^p(\tau') \tag{40}$$

We also have that

$$\begin{array}{lll} \mathcal{S}(\forall\vec{\rho}\vec{\varepsilon}.\mathcal{S}^p(\tau')) & = \forall\vec{\rho}\vec{\varepsilon}.\mathcal{S} \setminus\setminus \text{fev}(\vec{\varepsilon})(\mathcal{S}^p(\tau')) & \text{lemma 4.8 on (34) and (33)} \\ & = \forall\vec{\rho}\vec{\varepsilon}.\mathcal{S}^p(\tau') & (40) \end{array} \tag{41}$$

Now, by (40) and (41), we can rewrite (34) like this

$$\mathcal{S}_1(TE') + \{f \mapsto (\forall\vec{\rho}\vec{\varepsilon}.\tau_1', p)\} \vdash te_1 \hookrightarrow e_1 : (\tau_1', p), \psi_1 \tag{42}$$

where

$$\tau_1' = \mathcal{S}^p(\tau') \tag{43}$$

Now we have completed the first part of the proof of this case. Let us now provide the premises for using induction on the second subexpression. By an argument similar to that from (31) to (34) we get that

$$\begin{array}{l} \mathcal{S}_1(\Psi),\ \mathcal{S}_1(EE) + \left\{f \mapsto \left(\forall\vec{\rho}\vec{\alpha}\vec{\varepsilon}.\text{Lookup}_{\mathcal{S}_1(\Psi)}(\vec{\varepsilon}), p\right)\right\} \\ \quad \models TE + \{f \mapsto (\forall\vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau, p)\} \\ \quad \Rightarrow \mathcal{S}_1(TE') + \{f \mapsto (\forall\vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau_1', p)\} \end{array} \tag{44}$$

By lemma 4.11 on (37) and (8) we get

$$\mathcal{S}_1(\Psi) \models \mu_2 \Rightarrow \mathcal{S}_1(\mu_2') \tag{45}$$

By induction on (44), (45), and (7) we that there exist $\mathcal{S}_2$, e_2, and ψ_2 such that

$$(\mathcal{S}_2, e_2, \psi_2) = \mathcal{M}(\mathcal{S}_1(\Psi), \mathcal{S}_1(EE) + \left\{f \mapsto \left(\forall\vec{\rho}\vec{\alpha}\vec{\varepsilon}.\mathrm{Lookup}_{\mathcal{S}_1(\Psi)}(\vec{\varepsilon}), p\right)\right\}, te_2) \tag{46}$$

where

$$\mathcal{S}_2 \text{ is a multiplicity substitution on } \mathcal{S}_1(\Psi) \tag{47}$$

and

$$\mathcal{S}_2(\mathcal{S}_1(TE') + \{f \mapsto (\forall\vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau_1', p)\}) \vdash te_2 \hookrightarrow e_2 : \mathcal{S}_2(\mathcal{S}_1(\mu_2')), \psi_2 \tag{48}$$

By (35) and (43) we have that $\mathcal{M}$ terminates and

$$(\mathcal{S}_2 \circ \mathcal{S}_1, \texttt{letrec } f = e_1 \texttt{ in } e_2, \mathcal{S}_2(\psi_1) \oplus \psi_2) = \mathcal{M}(\Psi, EE, \texttt{letrec } f : \forall\vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau \texttt{ at } p = te_1 \texttt{ in } te_2) \quad \boxed{46}$$

By (37) and (44) we get that

$$\mathcal{S}_2 \circ \mathcal{S}_1 \text{ is a multiplicity substitution on } \Psi \quad \boxed{47}$$

Now let

$$\mathcal{S}_2' = \mathcal{S}_2 \downarrow_{\mathrm{fev}(\mathcal{S}_1(\mathrm{TE}') + \{\mathrm{f} \mapsto (\forall\vec{\rho}\vec{\varepsilon}.\tau_1', \mathrm{p})\}) \cup \mathrm{fev}(\psi_1)} \tag{48}$$

It clearly holds that

$$\mathcal{S}_2(\mathcal{S}_1(TE')) = \mathcal{S}_2'(\mathcal{S}_1(TE')) \quad \mathcal{S}_2(\forall\vec{\rho}\vec{\varepsilon}.\tau_1') = \mathcal{S}_2'(\forall\vec{\rho}\vec{\varepsilon}.\tau_1') \quad \mathcal{S}_2(\forall\vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau_1') = \mathcal{S}_2'(\forall\vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau_1') \quad \mathcal{S}_2(\psi_1) = \mathcal{S}_2'(\psi_1) \tag{49}$$

By lemma 3.24 on (44) we have that

$$\mathcal{S}_2' \text{ is a multiplicity substitution on } \Psi \tag{50}$$

And we have that

$$\begin{aligned} \mathcal{S}_2'(\forall\vec{\rho}\vec{\varepsilon}.\tau_1') &= \forall\vec{\rho}\vec{\varepsilon}.\mathcal{S}_2' \setminus\setminus \{\vec{\varepsilon}\}(\tau_1') && \text{lemma 4.8 on (50)} \\ &= \forall\vec{\rho}\vec{\varepsilon}.\mathcal{S}_2'(\tau_1') && \text{(48) and (6)} \end{aligned} \tag{51}$$

By lemma 3.11 on 5 we get that

$$\underline{\Psi} \models \varphi_1 \tag{52}$$

By theorem 3.26 on 5 and (34) we get

$$\underline{\psi_1} = \varphi_1 \tag{53}$$

and from (53) and (52) we get

$$\Psi \models \psi_1 \tag{54}$$

By lemma 4.14 on (44), (31), (54), (42) and (50) we get

$$\mathcal{S}_2'(\mathcal{S}_1(TE') + \{f \mapsto (\forall\vec{\rho}\vec{\varepsilon}.\tau_1', p)\}) \vdash te_1 \hookrightarrow e_1 : (\mathcal{S}_2'(\tau_1'), p), \mathcal{S}_2'(\psi_1) \tag{55}$$

which by (49) and (55) can be rewritten like this

$$(\mathcal{S}_2(\mathcal{S}_1(TE')) + \{f \mapsto (\forall\vec{\rho}\vec{\varepsilon}.\mathcal{S}_2'(\tau_1'), p)\}) \vdash te_1 \hookrightarrow e_1 : (\mathcal{S}_2'(\tau_1'), p), \mathcal{S}_2(\psi_1) \tag{56}$$

Similarly, (45) can be rewritten like this

$$(\mathcal{S}_2(\mathcal{S}_1(TE')) + \{f \mapsto (\forall\vec{\rho}\vec{\alpha}\vec{\varepsilon}.\mathcal{S}_2'(\tau_1'), p)\}) \vdash te_2 \hookrightarrow e_2 : \mathcal{S}_2(\mathcal{S}_1(\mu_2')), \psi_2 \tag{57}$$

By lemma 4.11 one (10) and (30), (43), (50) we get

$$(\mathcal{S}_2' \circ \mathcal{S}^p)(\Psi) \models \tau \Rightarrow \mathcal{S}_2'(\tau_1') \tag{58}$$

Now, a last, by the LETREC-rule on (56), (57), and (58) we get

$$\begin{array}{l}\mathcal{S}_2(\mathcal{S}_1(TE)) \vdash \texttt{letrec } f{:}\forall\vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau \texttt{ at } p = te_1 \texttt{ in } te_2 \\ \quad \hookrightarrow \texttt{letrec } f[\vec{\rho}] \texttt{ at } p = e_1 \texttt{ in } e_2 : \mathcal{S}_2(\mathcal{S}_1(\mu_2')), \mathcal{S}_2(\psi_1) \oplus \psi_2\end{array} \qquad \boxed{59}$$

$\boxed{\texttt{let } x{:}\forall\vec{\varepsilon}\vec{\alpha}.\tau = te_1 \texttt{ in } te_2}$

The proof of this case is just a simplified version of that of the letrec-case.

$\boxed{\texttt{letregion } \varphi \texttt{ in } te}$

Here (2) looks like this

$$\mathrm{T}(\underline{\Psi}, TE, \texttt{letregion } \varphi \texttt{ in } te) = (\mu, \varphi'') \tag{3}$$

This could have been returned on the following premises

$$\mathrm{T}(\underline{\Psi}, TE, te) = (\mu, \varphi') \tag{4}$$

$$\varphi'' = \mathrm{Observe}(TE, \mu)(\varphi') \tag{5}$$

$$\varphi = \varphi' \setminus \varphi'' \tag{6}$$

By induction on (0), (1), and (4) we get that there exist $\mathcal{S}$, e, and ψ such that

$$(\mathcal{S}, e, \psi) = \mathcal{M}(\Psi, EE, te) \tag{7}$$

where

$$\mathcal{S} \text{ is a multiplicity substitution on } \Psi \qquad \boxed{8}$$

and

$$\mathcal{S}(TE') \vdash te \hookrightarrow e : \mathcal{S}(\mu'), \psi \tag{9}$$

Now let

$$\mathrm{put}(\rho_1) \cdots \mathrm{put}(\rho_k) = \mathrm{PutEffects}(\varphi) \tag{7}$$

By (4) and (7) we have that $\mathcal{M}$ terminates and

$$(\mathcal{S}, \texttt{letregion } \rho_1{:}\psi^+(\mathrm{put}(\rho_1)) \cdots \rho_k{:}\psi^+(\mathrm{put}(\rho_k)) \texttt{ in } e, \psi \setminus\setminus \varphi) = \mathcal{M}(\Psi, EE, \texttt{letregion } \varphi \texttt{ in } te) \qquad \boxed{8}$$

By the LETREGION-rule on (6) and (7) we get

$$\mathcal{S}(TE') \vdash \texttt{letregion } \varphi \texttt{ in } te \hookrightarrow \texttt{letregion } \rho_1{:}\psi^+(\mathrm{put}(\rho_1)) \cdots \rho_k{:}\psi^+(\mathrm{put}(\rho_k)) \texttt{ in } e : \mathcal{S}(\mu'), \psi \setminus\setminus \varphi \qquad \boxed{9}$$

■

4.5 Incompleteness

In multiplicity inference typings can be defined as principal, if the multiplicities of their types, effects, and target expressions are as small as possible. As we shall see in this section, algorithm $\mathcal{M}$ is not complete.

One source of imprecision is the non-commutativity of substitutions and the maximum operation. As pointed out in the previous chapter, the maximum operation is imprecise in its treatment of effectvariables: it cannot predict with how much an effectvariable will contribute to the effect in the future, when it is hit by substitutions. Here is an example (this is a source program, i.e. it has to be submitted to region inference first before being run through multiplicity inference):

$$
\begin{array}{l}
\lambda f. \\
\quad \textbf{let } x = \textbf{ if } \texttt{true} \textbf{ then } 1 \textbf{ else } (f\ 1) \\
\quad \textbf{in } \textbf{ if } \texttt{true} \textbf{ then } \lambda x.1 \textbf{ else } f
\end{array}
$$

Let us focus on the effect of the let-part of this lambda expression, i.e. the effect of calculating x. If we know to begin with that the type of f is $(int, \rho_1) \xrightarrow{\varepsilon_1.\{\mathrm{put}(\rho_2)\mapsto 1\}} (int, \rho_2)$ we could see directly that the effect of calculating x is $\lceil\{\mathrm{put}(\rho_2) \mapsto 1\}, \{\varepsilon_1 \mapsto 1, \mathrm{put}(\rho_2) \mapsto 1\}\rceil = \{\varepsilon_1 \mapsto 1, \mathrm{put}(\rho_2) \mapsto 1\}$, and when elaborating the rest of the body of the lambda expression we would not have to revise our assumption for f.

The way the $\mathcal{M}$ elaborates the expression, it has an imprecise knowledge about the effect ε_1 when it performs the maximum operation: It assumes that the multiplicity of the put-effect to ρ_2 on the arrow of the type of f has multiplicity 0. However, the result of the maximum operation is the same, namely $\lceil\{\mathrm{put}(\rho_2) \mapsto 1\}, \{\varepsilon_1 \mapsto 1, \mathrm{put}(\rho_2) \mapsto 0\}\rceil = \{\varepsilon_1 \mapsto 1, \mathrm{put}(\rho_2) \mapsto 1\}$. When the algorithm then proceeds to elaborate the body of the lambda expression and encounters $\lambda x.1$, it finds out that the true effect ε_1 is $\{\mathrm{put}(\rho_2) \mapsto 1\}$. It therefore generates a substitution $\{\varepsilon_1 \mapsto \varepsilon_1.\{\mathrm{put}(\rho_2) \mapsto 1\}\}$, which turns the effect inferred for calculating x into $\{\varepsilon_1 \mapsto 1, \mathrm{put}(\rho_2) \mapsto 2\}$. So the put effect to ρ_2 becomes one larger than necessary, due to the fact that the maximum operation was used at a point in time where our knowledge about the effect of ε_1 was inadequate.

As a curiousity it should be mentioned that our policy of always setting the atomic effects of cyclic arrow effects to ∞ in rare cases gives rise to imprecision:

$$
\begin{array}{l}
(\lambda f. \\
\quad \textbf{let } \mathit{applyF} = \lambda x.(f\ x) \\
\quad \textbf{in } \textbf{ if } \texttt{true} \textbf{ then } \mathit{applyF} \textbf{ else } f) \\
\lambda x.1
\end{array}
$$

By using the inference rules intelligently, it is possible to infer the type $(\alpha, \rho_1) \xrightarrow{\varepsilon_1.\{\varepsilon_1\mapsto 1, \mathrm{put}(\rho_2)\mapsto 1\}} (int, \rho_2)$ for this expression. $\mathcal{M}$, however, will run into the cyclic arrow effect when instantiating applyF in the body of the first lambda abstraction and set the atomic effects to ∞.

As we shall se in the chapter about experimental results, the incompleteness of $\mathcal{M}$ appears to be of no practical importance at all. In fact, takes some effort to construct examples that reveal the incompleteness.

4.6 Implementation and efficiency

In this section we discuss the possibilities of implementing $\mathcal{M}$ efficiently.

When assessing $\mathcal{M}$ from the point of view of efficiency of implementation, the first question that attracts attention is, whether the algorithm really hinges on using substitutions for updating Ψ, or whether it could just as well use some more efficient means of refinement. An application of a substitution to a set of arrow effect is expensive, because you have to traverse the whole set, although the effect variables in the domain of the substitution typically will only occur at a few places.

It is clear that Ψ must be updated by assignment statements rather than substitutions. It would have been ideal just to update the ε-entry, whenever $\mathcal{M}$ generates a substitution $\{\varepsilon \mapsto \varepsilon.\psi\}$. But ε can also occur with its effect in non-prinipal positions elsewhere in Ψ, so updating only the ε-entry of Ψ will only work, if these occurences are also affected indirectly by the update proper entry of ε. So there must be some kind of link between the non-principal occurrences of an effect variable and its own entry in Ψ. This, however, is complicated by the fact is that the maximum operation destroys the internal structure of effects, it makes them shallow. To illustrate this point, let us look at an example. Define

$$\begin{aligned}\psi_1 &= \{\mathrm{put}(\rho_1) \mapsto 1\}\\ \psi_2 &= \{\mathrm{put}(\rho_1) \mapsto 2\}\end{aligned}$$

Let Ψ be a set of arrow effects which contains both $\varepsilon_1.\psi_1$ and $\varepsilon_2.\psi_2$. Now suppose that another arrow effect in Ψ is $\varepsilon_3.\psi_3 = \varepsilon_3.\lceil\{\varepsilon_1 \mapsto 1\} \cup \psi_1, \{\varepsilon_2 \mapsto 1\} \cup \psi_2\rceil$. The question now is how to represent ψ_3 so that it is possible to refine the information about ψ_1 without having to consult ψ_3 also.

If we just calculate ψ_3 directly and then update ψ_1, we will have to update ψ_3 at the same time, because you cannot decide later by looking at $\{\varepsilon_1 \mapsto 1, \varepsilon_2 \mapsto 1, \mathrm{put}(\rho_2) \mapsto 2\}$ how much ε_1 has contributed to this effect, so then it will be too late to update it.

The obvious way of avoiding these multiple updates would be to delay the concrete calculation of the maximum and just represent ψ_3 by two pointers to $\varepsilon_1.\psi_1$ and $\varepsilon_2.\psi_2$ and a tag signifying that it is the maximum of the two. We will call such an implementation of the maximum operation *lazy*. Thus ψ_3 would be affected automatically whenever either ψ_1 or ψ_2 changed. This, however, is not an implementation of substitutions, but a completely different approach to refinement. Suppose, for example, that ψ_1 changes to $\{\mathrm{put}(\rho_1) \mapsto 2\}$, and we then decide to calculate ψ_3. Then the result will have multiplicity 2 for ρ_1, as ψ_3 would discover that $\psi_1(\rho_1)$ is not yet greater than $\psi_2(\rho_1)$. This is better than using substitutions, where we would first calculate the maximum and then update $\varepsilon_1.\psi_1$, given ρ_1 in ψ_3 a final value of 3.

The must efficient solution to the problem is probably at every arrow effect in Ψ to maintain a list of pointers to the other arrow effects in which they occur. So after having updated the ε_1-entry in Ψ, we follow the pointer to ε_3. This scheme will be efficient, as non-principal occurences of effect variables are in fact comparatively rare in the effect tables for ordinary programs. So in most cases only a single entry in the table has to be. The dependency links between arrow effects can be calculated before multiplicity inference begins.

Another grossly inefficient point in the algorithm is the termination condition in the letrec-case: comparing two complete effect maps is an expensive operation. But this problem is easily solved by inferring a list of the effect variables that have been upgraded. We could have done the same in the substitution based version of $\mathcal{M}$ by restricting the substitutions generated in the λ and variable cases to the effect variables whose effects are really changed by the substitution. So both in an assignment and in a substitution based implementation of $\mathcal{M}$ we could instead of comparing two big sets of arrow effects just see directly, whether the elaboration of the expression had caused any further refinement.

The conclusion to this discussion must be that it is indeed possible to implement $\mathcal{M}$ efficiently. But we have no benchmarks to present, as we have only implemented a substitution based version of $\mathcal{M}$.

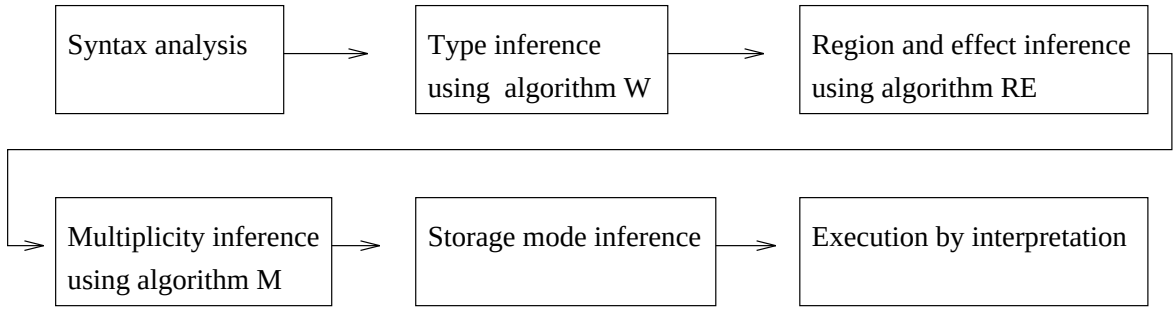
However, another multiplicity inference algorithm is being developed at DIKU by Mads Tofte. This algorithm represents effects as graphs, and sum and maximum are presented as nodes with two subgraphs as children, so they are lazy as in the example above. This algorithm promises both efficiency and a slightly improved precision of inference due to the lazyness of maximum.

5 Experimental results

In this section we investigate the performance of multiplicity inference by running target programs.

The results are from a prototype region inference compiler developed at DIKU by Mads Tofte and Lars Birkedal and extended with multiplicity inference by the author. The input language is the same as the one described in chapter 2, but it also has lists, pairs, references, case-statements, and various binary operators, such as +, -, *, = and :=. None of these extensions seem to cause any problems.

When given an input program this compiler goes through the following phases



Of these analyses only storage mode inference should be unfamiliar to the reader, and what it does will be explained later.

In this chapter we shall try to find out whether its is worthwile inferring sizes of regions, and, if it is, how good the analysis presented in the previous chapters is at doing it. We assume that memory is organized as described in the introduction: it is divided into two parts, a stack and a heap, where fixed-size regions are pushed directly onto the stack, and regions of unknown size are implemented as lists of blocks in the heap. Regions of size greater than one have a field containing the offset of the first free memory word in the region. We have two cardinal criterions, when assesing the results:

1) Speed.
   a) of region allocations.
      Regions of size one are allocated by incrementing the stack pointer. Regions of fixed size other than one are allocated by updating the stack pointer and initializing the offset. Infinite regions are allocated by first finding a free block in the heap and then initializing its offset. So its is very cheap to allocate a region of size one, fairly cheap to allocate a region of fixed size other than one, and expensive to allocate a region of unknown size.

   b) of value allocations
      Putting a value in a region of size one is done by writing directly to the address of the region. Writing to a region of fixed size other than one is done by reading and incrementing the region offset and putting the value in the first free place. Writing a value to a region of infinite size is done in the same way with the only difference that it may be necessary to traverse a list of blocks in the heap before finding the first free memory location. So it is cheap to write to a region of size one, expensive to write to a region of fixed size other than one, and it may be very expensive to put a value in a region of unknown size.

2) Memory usage - allocating too large regions can result in memory cells being reserved without ever being used for anything.

Our implementation has been run on a large corpus of programs. This corpus both contains ordinary programs taken from textbooks on functional programming and programs designed specifically to test region inference. In the following we discuss the results from a few of the programs in detail and then conclude the chapter by mentioning some general observations we made when studying the results from the whole corpus. We will first concentrate on the following programs

fib(15)

```
letrec fib = λn.
                if n = 0
                  then 1
                  else
                    if n = 1
                      then 1
                      else fib(n − 2) + fib(n − 1)
in fib(15)
```

fac(10)

```
letrec fac = λn.
                if n = 0
                  then 1
                  else n ∗ fac(n − 1)
in fac(10)
```

facacc((1, 10))

```
letrec facacc = λp.
      let n = snd(p)
      in
            let acc = fst(p)
            in
                if n = 0
                  then acc
                  else facacc(n ∗ acc, n − 1)
in facacc((1, 10))
```

quicksort(1000) The implementation of quicksort given in [Pau91], which uses lists. The program generates a list of 1000 random numbers and sorts them using quicksort.

When these program were run through the region inference system, the interpreter collected the following statistics

	fib(10)	*fac*(10)	*quicksort*(1000)	*facacc*((1, 10))	*facacc*(1, 10)*
maxregion	91	46	5021	27	27
region allocs	15030	66	65004	47	47
regions of size 1	15030	66	58924	44	44
regions of fixed size	0	0	0	0	0
regions of unknown size	0	0	6080	3	3
value allocs	15030	66	133750	77	77
to regions of size 1	15030	66	58924	44	44
to regions of fixed size	0	0	0	0	0
to regions of unknown size	0	0	74826	33	33
max. memory size	91	46	10567	27	57
max. stack height	91	46	5016	24	24
max. heap size	0	0	10479	3	33
final memory size	1	1	3002	1	11
final stack height	1	1	0	0	0
final heap size	0	0	3002	1	11

Here the height of the stack is the sum of the sizes of all live regions of known size. So the stack grows every time a region of known size is allocated. The heap size is the number of values put in live regions of unknown size. So the heap grows every time a value is put in a region of unknown size. The memory size is the sum of the stack height and the heap size. So the maximum memory size is less than or equal to the sum of the maximum stack height and the maximum heap size (the heap and the stack do not necessarily reach their peak at the same time).

The factorial and fibonacci functions behave ideally: all regions are of size one. So here multiplicity inference could not have been better.

To appreciate the *quicksort* example, it is necessary to know how lists are handled by region inference. A list is distributed over three regions, the elements are in one region, the constructors (cons and nil) in another, and the pairs to which cons is applied are in the third region. So a list of length n will consist of n pair values and $n + 1$ constructor values. The number of element values is at most n, but it can be less if the same value occurs at different positions in the list. The 3002 values the store contains after executing *quicksort*(1000) are 1001 constructors, 1000 pairs, 1000 list elements, plus an extra value from the last iteration of the random number generator put in the same region as the list elements.

If we try to relate this to multiplicity inference, a list which is constructed recursively by a function will typically have constructor and pair regions of unknown size. Only constant lists built by hand by the programmer have regions of fixed size.

The *quicksort* program works by first generaring a list of random numbers and then sorting them. The sorting consists in partitioning the input list by comparing the elements with the first element in the list, sorting them, and appending the sorted sublists together again. The lists created by the partitioning, sorting, and appending naturally have regions of unknown size. But even in this example multiplicity inference discovers a lot of small regions: only one tenth of the regions allocated are of unknown size, and almost half of the values (primarily booleans and closures) are written to regions of size one.

So we must conclude that also in this example we could not have expected multiplicity inference to perform better.

We shall now see how multiplictity inference interacts with storage mode inference. Storage mode inference brings a kind of tail recursion optimization to region inference. It determines when the values in a region can be soundly overwritten. It introduces different modes of writing the store: for every '`at` p' in the region program, the analysis decides whether it is an 'attop p' (the value is put on top of the other values in p) or an 'atbot p' (the region is reset, and the value is put at bottom of it). Consider the *facacc* example: here the

pair, the accumulator, and n can be overwritten at the recursive call. Therefore the expressions are set to overwrite the other values in the regions. This, of course, affects the size of the regions. But unfortunately multiplicity inference is farther down the food chain than storage mode inference, so it does not know that the values in *facacc* can be overwritten. As can be seen in the table, multiplicity inference infers unknown size for the regions of the pair, the accumulator, and n, although they never contain more than one value. In fact, the computation which multiplicity inference predicts, is the one in the table under *facacc*(1, 10)*, where we have run *facacc* without using storage mode inference, and as can be seen that in this case the regions are of indeterminable size.

It is, of course, a disadvantage that multiplicity inference does not take storage modes into account. The reason is historical: the work on multiplicity inference was begun before storage mode inference was conceived. However, it is no catastrophy. Storage mode inference only makes a difference in expressions with a low level of region polymorphism[TT93a, 44-45]. So the extra overhead connected with allocating regions of unknown size instead of regions of size one is minimal, as we are always dealing with a small number of region allocations (in *facacc* there are 3 such 'global' regions). The real problem is the value allocations: instead of doing cheap writes to regions of size one, the program does expensive writes to a region of unknown size.

It is not altogether obvious how multiplicity inference should be modified so that it could distinguish between different modes of writing values to the store, or whether the problem should be solved by a third analysis, which, based on the output from storage mode inference, could correct the sizes of the critical regions.

Among the general observations we made, when assessing the results from all the programs from the test corpus is the fact that regions of size other than 1 or ∞ virtually never occur. We ran the tests on a fine-grained version of the multiplicity inference implementation which distinguished between 25 multiplicities. In spites of this the only occurence of a region of size other than 1 or ∞ was a program which had a list of three elements built by hand by the programmer.

So there is no real time-precision trade-off in multiplicity inference. To distinguish between 0, 1, and ∞ is precise enough.

Another observation we made is that none of the programs managed to reveal the incompleteness of the algorithm. So given the variedness of the program corpus we must conclude that from a practical point of view the incompleteness constitutes no problem.

The general conclusion to the experiments must be that multiplicity inference performs excellently. No real problems or sources of imprecision were discovered. A remaining problem is to fill the gap between multiplicity inference and storage mode inference.

6 Conclusion

We have in this report presented an analysis for inferring sizes of regions in region inference, and we have developed an algorithm, which we proved sound with respect to the translation semantics of multiplicity inference. The important property which was used to show this was multiplicity consistency.

The precision of the analysis is excellent. Only a little fraction of the regions allocated are of unknown size. Many of our programs had no regions of unknown size at all. Programs that have regions of unknown sizes, however, tend to have a lot of value allocations to them. The incompleteness of our algorithm turned out to be of no practical importance. It was unproblematic to extend language with facilities corresponding roughly to the core language of ML. A remaining tasks is to modify the output from multiplicity inference so that storage modes are taken into account.

Multiplicity inference is part of region inference. So the significance of this work depends on the success of region inference. Although the results from the prototype compiler are very promising, we must wait for the benchmarks from the real compiler now being built at DIKU by Mads Tofte and Lars Birkedal, before we know how well region inference performs compared to the main implementations of ML.

Ackknowledgement

I thank Mads Tofte for his patience, support, and help with every part of this work.

Appendix A, proof of lemma 3.9

lemma 3.9. If $\Phi \models \{\tau, TE, \varphi\}$ then

$$\Phi \models \mathrm{RegTyEffGen}(TE, \varphi)(\tau)$$

$$\Phi \models \mathrm{RegEffGen}(TE, \varphi)(\tau)$$

and

$$\Phi \models \mathrm{TyEffGen}(TE, \varphi)(\tau)$$

Proof.
Assume

$$\Phi \models \{\tau, TE, \varphi\} \tag{0}$$

Let $\mathrm{RegTyEffGen}(TE, \varphi)(\tau) = \forall \vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau$. It obviously holds that

$$\mathrm{fev}(\mathrm{RegTyEffGen}(TE, \varphi)(\tau)) \subseteq \mathrm{fev}(\Phi) \tag{1}$$

And we can certainly pick $\vec{\rho'}$ and $\vec{\varepsilon'}$ such that

$$\mathrm{RegTyEffGen}(TE, \varphi)(\tau) = \forall \vec{\rho'}\vec{\alpha}\vec{\varepsilon'}.\mathcal{R}(\tau) \tag{2}$$

$$\mathrm{fev}(\vec{\varepsilon'}) \cap \mathrm{fev}(\Phi) = \{\} \quad \mathrm{frv}(\vec{\rho'}) \cap \mathrm{frv}(\Phi) = \{\} \tag{3}$$

where

$$\mathcal{R} = [\vec{\varepsilon'}/\vec{\varepsilon}, \vec{\rho'}/\vec{\rho}] \tag{4}$$

As we do not allow the same variable to be quantified more than once in a type scheme, we have that $\mathcal{R}$ is a bijection. By $\mathcal{R}^{-1}$ we mean the inverse of $\mathcal{R}$. Notice that $\mathcal{R}$ is not a real substitution, but just a renaming of variables (effect variables are mapped to effect variables and not to arrow effects). Now let

$$\Phi' = \{\varepsilon.\hat{\Phi}(\varepsilon) \mid \varepsilon \in \mathrm{fev}(\vec{\varepsilon})\} \tag{5}$$

We will now show

$$\Phi \cup \mathcal{R}(\Phi') \text{ is effect-consistent} \tag{6}$$

We check one requirement from definition 3.2 at a time.

req. 1) This is the requirement that an effect variable must only occur with one effect in a principal position. We know from (0) that this requirement is satisfied for Φ. It must also be satisfied for Φ', since it is a subset of Φ, and thus also for $\mathcal{R}(\Phi')$, since $\mathcal{R}$ is a total bijection on $\mathrm{fpev}(\Phi')$. Hence it also holds for $\Phi \cup \mathcal{R}(\Phi')$, as we, by (3), have that $\mathrm{fpev}(\Phi) \cap \mathrm{fpev}(\mathcal{R}(\Phi')) = \{\}$.

req. 3) This is the requirement that all effect variables must also occur in a principal position. Let ε be an arbitrary effect variable in $\mathrm{fev}(\Phi \cup \mathcal{R}(\Phi'))$. If $\varepsilon \in \mathrm{fev}(\Phi)$ then by (0) we have that $\varepsilon \in \mathrm{fpev}(\Phi) \subseteq \mathrm{fpev}(\Phi \cup \mathcal{R}(\Phi'))$. If $\varepsilon \notin \mathrm{fev}(\Phi)$, then $\varepsilon \in \mathbf{rng}(\mathcal{R})$, and it must be the case that $\varepsilon \in \mathrm{fpev}(\mathcal{R}(\Phi')) \subseteq \mathrm{fpev}(\Phi \cup \mathcal{R}(\Phi'))$.

Since we have now established that requirement 1) and 3) are satisfied and we thus know that every effect variable in $\Phi \cup \mathcal{R}(\Phi')$ occurs with exactly one effect in a principal position, we can define a function F on $\mathrm{fev}(\Phi \cup \mathcal{R}(\Phi'))$ such that if $\varepsilon.\varphi \in \Phi \cup \mathcal{R}(\Phi')$, then $F(\varepsilon) = \varphi$ (we cannot yet use the $\hat{}$-notation, as we have not yet proven effect consistency).

req. 2) This is the requirement that an effect variable always occur together with its effect in non-principal positions. We show that this is satisfied by contradiction: We suppose that there exist $\varepsilon', \varphi', \varepsilon''$ such that $\varepsilon'.\varphi' \in \Phi \cup \mathcal{R}(\Phi')$, $\varepsilon'' \in \varphi'$ and $F(\varepsilon'') \not\subseteq \varphi'$. We know that $\varepsilon'.\varphi' \in \mathcal{R}(\Phi')$, since requirement 2), by (0), is satisfied for all effect variables in Φ. We know by (5) that $\mathcal{R}^{-1}(\varepsilon'.\varphi') \in \Phi$. We now distinguish between two cases:

a $\varepsilon'' \in \mathrm{fev}(\vec{\varepsilon'})$. In this case there must exist ε such that

$$\varepsilon = \mathcal{R}^{-1}(\varepsilon'') \text{ and } \varepsilon''.\mathcal{R}(\hat{\Phi}(\varepsilon)) \in \mathcal{R}(\Phi')$$

By (0) it must hold that

$$\{\varepsilon\} \cup \hat{\Phi}(\varepsilon) \subseteq \mathcal{R}^{-1}(\varphi')$$

Applying $\mathcal{R}$ to both sides of this inequality gives

$$\mathcal{R}(\{\varepsilon\} \cup \hat{\Phi}(\varepsilon)) \subseteq \varphi'$$

But

$$\begin{aligned} \mathcal{R}(\{\varepsilon\} \cup \hat{\Phi}(\varepsilon)) &= \{\varepsilon''\} \cup \mathcal{R}(\hat{\Phi}(\varepsilon)) \\ &= \{\varepsilon''\} \cup F(\varepsilon'') \end{aligned}$$

So we have reached a contradiction.

b $\varepsilon'' \notin \mathrm{fev}(\vec{\varepsilon'})$. In this case $\varepsilon'' \in \mathrm{fev}(\Phi)$. By (0) we know that

$$\varepsilon'' \cup \hat{\Phi}(\varepsilon'') \subseteq \mathcal{R}^{-1}(\varphi')$$

So if

$$\varepsilon'' \cup \hat{\Phi}(\varepsilon'') \not\subseteq \varphi'$$

it must be because that there are effect and region variables in $\hat{\Phi}(\varepsilon'')$ that are in the domain of $\mathcal{R}$. But that means that these were bound when τ was closed against φ and TE. This again is implies that ε'' occurs free somewhere in TE or φ without all of the atomic effects in $\hat{\Phi}(\varepsilon'')$, which contradicts (0).

As all paths lead to contradictions we must conclude that also requirement 2) is satisfied for $\Phi \cup \mathcal{R}(\Phi')$.

We know only need to convince ourselves that

$$\Phi \cup \mathcal{R}(\Phi') \models \mathcal{R}(\tau) \tag{7}$$

We already know that $\Phi \cup \mathcal{R}(\Phi')$ is effect-consistent, so we only need to make sure that $\mathrm{aref}(\mathcal{R}(\tau)) \subseteq \Phi \cup \mathcal{R}(\Phi')$. Let $\varepsilon'.\varphi'$ be an arbitrary arrow effect in $\mathcal{R}(\tau)$). From (0) we know that $\mathrm{aref}(\tau) \subseteq \Phi$. So all arrow effects in $\mathcal{R}(\tau)$ that are left untouched by $\mathcal{R}$ will still be included in Φ. So it is only arrow effects in which some variable has been quantified that can be a problem. Of those the ones whose principal effect variable is quantified are included in $\mathcal{R}(\Phi')$. But what about arrow effects whose principal effect variable is not quantified but which nevertheless contain quantified variables in the effect. Such arrow cannot exist in $\mathcal{R}(\tau)$, as one can convince oneself by using an argument similar to the one for requirement 2) case **b** above. This establishes (7).

Finally, by definition 3.4 on (2), (6), (1), (3), and (7) we get the desired

$$\Phi \models \mathrm{RegTyEffGen}(TE, \varphi)(\tau)$$

That $\Phi \models \mathrm{TyEffGen}(TE, \varphi)(\tau)$ and $\Phi \models \mathrm{RegEffGen}(TE, \varphi)(\tau)$ hold is proven in the same way.

■

Appendix B, proof of lemma 4.23

In this appendix we present the proof of lemma 4.23. This is the lemmas that states that our way of instantiating effect variables is correct. Remember the way Instantiate works: at every recursive call the effect variables corresponding a ground segment of the instantiation vector of arrow effects are instantiated. In the proof of lemma 4.23 we shall therefore need a lemma stating that our way of instantiating ground segments works. That is what the following lemma does

Lemma 6.2. *Suppose that*

$$\Psi, \forall\vec{\varepsilon'}.\Psi' \models \forall\vec{\alpha}\vec{\varepsilon'}.\tau \Rightarrow \forall\vec{\alpha}\vec{\varepsilon'}.\tau' \tag{0}$$

Let $\{\varepsilon'_{i_1}.\psi'_{i_1}, \ldots, \varepsilon'_{i_k}.\psi'_{i_k}\}$ *be a subset of* Ψ' *such that*

$$\{\varepsilon'_1 \cdots \varepsilon'_k\} \subseteq \mathrm{fev}(\vec{\varepsilon'}) \tag{1}$$

and let $\{\varepsilon_{i_1}.\psi''_{i_1}, \ldots, \varepsilon_{i_k}.\psi''_{i_k}\}$ *be a vector of arrow effects such that*

$$S_e(\{\varepsilon'_{i_1}.\psi'_{i_1}, \ldots, \varepsilon'_{i_k}.\psi'_{i_k}\}) \subseteq \Psi \tag{2}$$

where

$$S_e = \{\varepsilon'_{i_1} \mapsto \varepsilon_{i_1}.\psi''_{i_1}, \ldots, \varepsilon'_{i_k} \mapsto \varepsilon_{i_k}.\psi''_{i_k}\} \tag{3}$$

It now holds that

$$\Psi, \forall\vec{\varepsilon'_0}.S_e(\Psi') \models \forall\vec{\alpha}\vec{\varepsilon'_0}.\underline{S_e}(\tau) \Rightarrow \forall\vec{\alpha}\vec{\varepsilon'_0}.S_e(\tau')$$

where $\vec{\varepsilon'_0} = \vec{\varepsilon'} \setminus\setminus \{\varepsilon'_1 \cdots \varepsilon'_k\}$.

Proof.
Define

$$\varepsilon_{i_j}.\psi_{i_j} = S_e(\varepsilon'_{i_j}.\psi'_{i_j}) \qquad \text{for } 1 \leq j \leq k \tag{4}$$

We suppose that $\vec{\varepsilon'}$ is chosen so that

$$\mathrm{fev}(\vec{\varepsilon'}) \cap \mathrm{fev}(\Psi) = \{\} \tag{5}$$

$$\mathrm{fev}(\forall\vec{\alpha}\vec{\varepsilon'}.\tau) \subseteq \mathrm{fev}(\Psi) \qquad \mathrm{frv}(\forall\vec{\alpha}\vec{\varepsilon'}.\tau) \subseteq \mathrm{frv}(\Psi) \tag{6}$$

$$\Psi \cup \Psi' \models \tau \Rightarrow \tau' \tag{7}$$

We will now show that S_e is a consistency preserving substitution on $\Psi \cup \Psi'$. By (1) and 5 we know that $S_e(\Psi) = \Psi$, so S_e affects only Ψ'. We will now convince ourselves that $\Psi \cup S_e(\Psi')$ is multiplicity-consistent.

The first requirement of multiplicity consistency, namely that an effect variable can occur in a principal position with at most one effect, is satisfied: From (2) we know that even though the application S_e results in 'name clashes'(it maps principal effect variables to arrow effects whose principal effect variables already exist in Ψ), the effects of the resulting arrow effects are also the same.

So we have only left to convince ourselves that $\underline{\Psi \cup S_e(\Psi')}$ is effect-consistent. We have already shown that effect variables occur with exactly one arrow effect in $\Psi \cup S_e(\Psi')$, so the same holds for $\underline{\Psi \cup S_e(\Psi')}$. Hence the first demand in the definition of effect consistency is satisfied. We can therefore define a function F on $\mathrm{fpev}(\Psi \cup S_e(\Psi'))$ such that if $\varepsilon.\varphi \in \underline{\Psi \cup S_e(\Psi')}$, then $F(\varepsilon) = \varphi$ (we cannot yet use the $\hat{}$-notation, as we have not yet proven effect consistency).

The third requirement in the definition of effect consistency, namely that all effect variables must occur in a principal position, is obviously satisfied, as

$$\begin{aligned}
\mathrm{fev}(\underline{\Psi \cup S_e(\Psi')}) &= \mathrm{fev}(\Psi \cup S_e(\Psi')) \\
&= \mathrm{fev}(\Psi) \cup (\mathrm{fev}(\Psi') \setminus \{\varepsilon'_{i_1}, \ldots, \varepsilon'_{i_k}\}) \cup \mathrm{fev}(S_e(\{\varepsilon'_{i_1}.\psi'_{i_1}, \ldots, \varepsilon'_{i_k}.\psi'_{i_k}\})) && (3) \\
&= \mathrm{fev}(\Psi) \cup \mathrm{fev}(\Psi') \setminus \{\varepsilon'_{i_1}, \ldots, \varepsilon'_{i_k}\} && (2) \\
&= (\mathrm{fev}(\Psi) \cup \mathrm{fev}(\Psi')) \setminus \{\varepsilon'_{i_1}, \ldots, \varepsilon'_{i_k}\} \\
&= \mathrm{fpev}(\Psi) \cup \mathrm{fpev}(\Psi') \setminus \{\varepsilon'_{i_1}, \ldots, \varepsilon'_{i_k}\} && \Psi \cup \Psi' \text{ mul. cons.} \\
&= \mathrm{fpev}(\Psi) \cup \mathrm{fpev}(S_e(\Psi')) && (2), (3), \text{and } (5)
\end{aligned}$$

We will now show that the second requirement from the definition of effect consistency is also satisfied. We already know that $\underline{\Psi}$ is effect-consistent, so we only have to check that the requirement is satisfied for all arrow effects in $\underline{S_e(\Psi')}$. Now let $\varepsilon'.\varphi'$ be an arbitrary arrow effect in $\underline{\Psi'}$, and let $\varepsilon.\varphi = \underline{S_e}(\varepsilon'.\varphi')$. We will now make sure that for all effect variables $\varepsilon'' \in \varphi$, it holds that $F(\varepsilon'') \subseteq \varphi$. We distringuish between two cases

a $\varepsilon.\varphi \in \underline{S_e(\{\varepsilon'_{i_1}.\psi'_{i_1}, \ldots, \varepsilon'_{i_k}.\psi'_{i_k}\})}$. In this case $\varepsilon.\varphi \in \underline{\Psi}$, and $\underline{\Psi}$ is effect-consistent. So for all $\varepsilon'' \in \varphi$, we know that $\hat{\Psi}(\varepsilon'') = F(\varepsilon'') \subseteq \varphi$.

b $\varepsilon.\varphi \notin \underline{S_e(\{\varepsilon'_{i_1}.\psi'_{i_1}, \ldots, \varepsilon'_{i_k}.\psi'_{i_k}\})}$. In this case $\varepsilon = \varepsilon'$ is not in the domain of $\underline{S_e}$. So we have that $\varepsilon.\varphi = \varepsilon.\underline{S_e}(\varphi')$ for some $\varepsilon.ef' \in \underline{\Psi'}$. Let ε'' be an arbitrary effect variable in $\varphi = \underline{se}(\varphi')$. ε'' must have originated in at least one of the following ways:

1 $\varepsilon'' \in \varphi'$. This is the case where ε'' has been left untouched by S_e. We have that

$$\begin{aligned}
\underline{S_e}(\varphi') &= \underline{S_e}(\{\varepsilon''\} \cup \underline{\widehat{\Psi \cup \Psi'}}(\varepsilon'') \cup \varphi_{rest}) && \underline{\Psi \cup \Psi'} \text{ effect-consistent} \\
&= \underline{S_e}(\{\varepsilon''\}) \cup \underline{S_e}(\underline{\widehat{\Psi \cup \Psi'}}(\varepsilon'')) \cup \underline{S_e}(\varphi_{rest}) && \text{subst. distr. over } \cup \\
&= \{\varepsilon''\} \cup F(\varepsilon'') \cup \underline{S_e}(\varphi_{rest}) && \varepsilon'' \notin \mathbf{dom}(S_e), \text{ by } (1)
\end{aligned}$$

2 $\varepsilon'' = \varepsilon_{i_j}$, for some $j \in \{1, \ldots, k\}$, and $\varepsilon'_{i_j} \in \varphi'$. This is the case where S_e has mapped the original ε'_{i_j} to ε''. We have that

$$\begin{aligned}
\underline{S_e}(\varphi') &= \underline{S_e}(\{\varepsilon'_{i_j}\} \cup \underline{\psi'_{i_j}} \cup \varphi_{rest}) && \underline{\Psi \cup \Psi'} \text{ effect-consistent} \\
&= \underline{S_e}(\{\varepsilon'_{i_j}\} \cup \underline{\psi'_{i_j}}) \cup \underline{S_e}(\varphi_{rest}) && \text{subst. distr. over } \cup \\
&= \{\varepsilon_{i_j}\} \cup \underline{\psi_{i_j}} \cup \underline{S_e}(\varphi_{rest}) && (2) \text{ and } (4) \\
&= \{\varepsilon_{i_j}\} \cup \underline{\hat{\Psi}}(\varepsilon_{i_j}) \cup \underline{S_e}(\varphi_{rest}) && (2) \text{ and } \underline{\Phi} \text{ effect-consistent} \\
&= \varepsilon'' \cup F(\varepsilon'') \cup \underline{S_e}(\varphi_{rest})
\end{aligned}$$

3 $\varepsilon'' \in \underline{\psi''_{i_j}}$ for some $j \in 1, \ldots, k$. This is the case where ε'' has been added from one of the effects in the instantiation vector. Then it must holds that $\varepsilon'_{i_j} \in \varphi'$. But then we have

$$\begin{aligned}
\underline{S_e}(\varphi') &= \underline{S_e}(\{\varepsilon'_{i_j}\} \cup \underline{\psi'_{i_j}} \cup \varphi_{rest}) && \underline{\Psi \cup \Psi'} \text{ effect-consistent} \\
&= \underline{S_e}(\{\varepsilon'_{i_j}\} \cup \underline{\psi'_{i_j}}) \cup \underline{S_e}(\varphi_{rest}) && \text{subst. distr. over } \cup \\
&= \{\varepsilon_{i_j}\} \cup \underline{\psi_{i_j}} \cup \underline{S_e}(\varphi_{rest}) && (2) \\
&= \{\varepsilon_{i_j}\} \cup \underline{\hat{\Psi}}(\varepsilon_{i_j}) \cup \underline{S_e}(\varphi_{rest}) \\
&= \{\varepsilon_{i_j}\} \cup \{\varepsilon''\} \cup \underline{\hat{\Psi}}(\varepsilon'') \cup \varphi'_{rest} \cup \underline{S_e}(\varphi_{rest}) && \varepsilon'' \in \underline{\psi''_{i_j}},\ \underline{\psi''_{i_j}} \subseteq \underline{\psi_{i_j}} \\
& && \varepsilon_{i_j}.\underline{\psi_{i_j}} \in \underline{\Psi}, \\
& && \underline{\Psi} \text{ effect-consistent} \\
&= \{\varepsilon''\} \cup F(\varepsilon'') \cup \varepsilon_{i_j} \cup \varphi'_{rest} \cup \underline{S_e}(\varphi_{rest})
\end{aligned}$$

We have now shown that S_e is a consistency preserving substitution on $\Psi \cup \Psi'$, so by lemma 4.12 on (7) we get

$$S_e(\Psi \cup \Psi') \models \underline{S_e}(\tau) \Rightarrow S_e(\tau') \tag{8}$$

Now it quite obviously still holds that

$$\mathrm{fev}(\vec{\varepsilon'_0}) \cap \mathrm{fev}(\Psi) = \{\} \tag{9}$$

And from (2) and (6) we get that

$$\mathrm{fev}(\forall\vec{\alpha}\vec{\varepsilon'}.\underline{S_e}(\tau)) \subseteq \mathrm{fev}(\Psi) \tag{10}$$

And so, by the definition of multiplicity consistency on (8), (9), and (10) we get the desired

$$\Psi, \forall\vec{\varepsilon'_0}.S_e(\Psi') \models \forall\vec{\alpha}\vec{\varepsilon'_0}.\underline{S_e}(\tau) \Rightarrow \forall\vec{\alpha}\vec{\varepsilon'_0}.S_e(\tau')$$

■

Now we can finally present the proof of lemma 4.23:

lemma 4.23.*Suppose*

$$\Psi, \forall\vec{\varepsilon'}.\Psi' \models \forall\vec{\alpha}\vec{\varepsilon'}.\tau \Rightarrow \forall\vec{\alpha}\vec{\varepsilon'}.\tau' \tag{0}$$

Let $\varepsilon.\vec{\varphi}$ *be a vector of arrow effects such that*

$$\mid \varepsilon.\vec{\varphi} \mid = \mid \vec{\varepsilon'} \mid \tag{1}$$

$$\underline{\Psi} \models [\varepsilon.\vec{\varphi}/\vec{\varepsilon'}](\tau) \tag{2}$$

and

$$\mathrm{aref}(\varepsilon.\vec{\varphi}) \subseteq \underline{\Psi} \tag{3}$$

Then there exist $\mathcal{S}$ *and* S_e *such that*

$$\mathcal{S} = \mathrm{Instantiate}(\Psi, \forall\vec{\varepsilon'}.\Psi', \varepsilon.\vec{\varphi})$$

$$\underline{S_e} = \varepsilon.\vec{\varphi}/\vec{\varepsilon'}$$

$$\mathcal{S}(\Psi), \{\} \models \forall\vec{\alpha}.\underline{S_e}(\tau) \Rightarrow \forall\vec{\alpha}.S_e(\mathcal{S}(\tau'))$$

where

$$\mathcal{S} \textit{ is a multiplicity substitution on } \Psi$$

and

$$\mathbf{dom}(\mathcal{S}) = \mathrm{fpev}(\varepsilon.\vec{\varphi})$$

Proof.
We suppose $\vec{\varepsilon'}$ is picked so that

$$\mathrm{fev}(\vec{\varepsilon'}) \cap \mathrm{fev}(\Psi) = \{\} \tag{4}$$

$$\mathrm{fev}(\forall\vec{\varepsilon'}\vec{\alpha}.\tau) \subseteq \mathrm{fev}(\Psi) \qquad \mathrm{frv}(\forall\vec{\varepsilon'}\vec{\alpha}.\tau) \subseteq \mathrm{frv}(\Psi) \tag{5}$$

and

$$\Psi \cup \Psi' \models \tau \Rightarrow \tau' \tag{6}$$

The proof now procedes by induction on $\mid \vec{\varepsilon'} \mid$.

1. $\mid \vec{\varepsilon'} \mid = 0$. By (1) we know that $\varepsilon.\vec{\varphi} = ()$. So in this case we have that

$$\mathcal{S} = \mathrm{Instantiate}(\Psi, \Psi', ())$$

where

$$\mathcal{S} = \{\} \tag{7}$$

Now pick $S_e = \{\}$. It holds that

$$\underline{S_e} = \{\} = ()/()$$

We have that Ψ' is unnecessary, as all the effect variables of the type scheme, by (5), are free and contained in Ψ. This means that none of the arrow effects of Ψ' are actually used when loading multiplicities into τ' at (6). Thus (6) can be rewritten like this

$$\Psi \cup \{\} \models \tau \Rightarrow \tau' \tag{8}$$

So by the definition of multiplicity consistency on (4), (5) and (8) we can conclude

$$\Psi, \{\} \models \forall \vec{\alpha}.\tau \Rightarrow \forall \vec{\alpha}.\tau' \tag{9}$$

And as $\mathcal{S}$ and S_e are empty substitutions, (9) can be written like this

$$\mathcal{S}(\Psi), \{\} \models \forall \vec{\alpha}.\underline{S_e}(\tau) \Rightarrow \forall \vec{\alpha}.S_e(\mathcal{S}(\tau'))$$

And the empty substitution $\mathcal{S}$ is naturally a multiplicity substitution on Ψ, and its domain is the empty set of effect variables (= fev(())).

2. $|\ \vec{\varepsilon'}\ | \geq 1$. Let $\varepsilon.\vec{\varphi} = (\varepsilon_1.\varphi_1, \ldots, \varepsilon_k.\varphi_k)$ and $\vec{\varepsilon'} = (\varepsilon'_1, \ldots, \varepsilon'_k)$. By (3) $\{\varepsilon_1.\varphi_1, \ldots, \varepsilon_k.\varphi_k\} \subseteq \underline{\Psi}$, so by lemma 4.21 $\{\varepsilon_1.\varphi_1, \ldots, \varepsilon_k.\varphi_k\}$ has a ground segment Φ. Let $\{i_1, \ldots, i_j\}$ be the offsets of the arrow effects of Φ in $\varepsilon.\vec{\varphi}$, i.e. all the positions in $\varepsilon.\vec{\varphi}$ where an arrow effect from Φ occurs. Just like Instantiate, we distinguish between two cases.

   a. $\Phi = \{\varepsilon.\varphi\}$, where $\varepsilon \notin \varphi$. Let $\{\varepsilon'_{i_1}.\psi'_{i_1}, \ldots, \varepsilon'_{i_j}.\psi'_{i_j}\} \subseteq \Psi'$.
   From (3) and (2) we know that

$$\text{for all } \varepsilon'.\psi' \in \{\varepsilon'_{i_1}.\psi'_{i_1}, \ldots, \varepsilon'_{i_j}.\psi'_{i_j}\} \text{ it holds that } [\varepsilon.\vec{\varphi}/\vec{\varepsilon'}](\varepsilon'.\underline{\psi'}) = \varepsilon.\varphi \tag{10}$$

   It also holds that

$$\neg\exists m \in \{1, \ldots, k\} : \exists n \in \{i_1, \ldots, i_j\} : \varepsilon'_m \in \text{fev}(\psi'_n) \tag{11}$$

   Otherwise, by (10), it would have been the case that $\varepsilon_m \in \varphi$, contradicting the fact that $\{\varepsilon.\varphi\}$ is a ground segment of $\{\varepsilon_1.\varphi_1, \ldots, \varepsilon_k.\varphi_k\}$ and $\varepsilon \notin \varphi$.
   Let $\varepsilon'.\psi'$ be an arbitrary arrow effect in $\{\varepsilon'_{i_1}.\psi'_{i_1}, \ldots, \varepsilon'_{i_j}.\psi'_{i_j}\} \subseteq \Psi'$. We have that

$$\begin{aligned} [\varepsilon.\vec{\varphi}/\vec{\varepsilon'}](\varepsilon'.\underline{\psi'}) &= \varepsilon.[\varepsilon.\vec{\varphi}/\vec{\varepsilon'}](\underline{\psi'}) \oplus \varphi \\ &= \varepsilon.\underline{\psi'} \oplus \varphi \qquad (11) \end{aligned} \tag{12}$$

   At the same (10) gives us that

$$[\varepsilon.\vec{\varphi}/\vec{\varepsilon'}](\varepsilon'.\underline{\psi'}) = \varepsilon.\varphi \tag{13}$$

   So we can conclude that $\underline{\psi'} \subseteq \varphi$. And as $\varepsilon'.\psi'$ was arbitrary, we can conclude that

$$\text{For all } \varepsilon'.\psi' \in \{\varepsilon'_{i_1}.\psi'_{i_1}, \ldots, \varepsilon'_{i_j}.\psi'_{i_j}\} \text{ it holds that } \underline{\psi'} \subseteq \varphi \tag{14}$$

   Now let

$$\psi = \left\lceil \hat{\Psi}(\varepsilon), \psi'_{i_1}, \ldots, \psi'_{i_j} \right\rceil \tag{15}$$

   We have that

$$\begin{aligned} \underline{\psi} &= \underline{\hat{\Psi}(\varepsilon)} \cup \underline{\psi'_{i_1}} \cup \cdots \cup \underline{\psi'_{i_j}} && \text{lemma 3.12 on (15)} \\ &= \varphi \cup \underline{\psi'_{i_1}} \cup \cdots \cup \underline{\psi'_{i_j}} && (3) \\ &= \varphi && (14) \\ &= \underline{\hat{\Psi}}(\varepsilon) && (3) \end{aligned} \tag{16}$$

Now by lemma 4.17 on (16) we have that

$$\mathcal{S}' = \left\{\varepsilon \mapsto \varepsilon.\psi \ominus \hat{\Psi}(\varepsilon)\right\} \text{ is a multiplicity substitution on } \Psi \tag{17}$$

and

$$\widehat{\mathcal{S}'(\Psi)}(\varepsilon) = \left\lceil \psi, \hat{\Psi}(\varepsilon) \right\rceil = \psi \tag{18}$$

and by lemma 4.11 on (0) we get

$$\mathcal{S}'(\Psi), \mathcal{S}'(\forall\vec{\varepsilon'}.\Psi') \models \forall\vec{\alpha}\vec{\varepsilon'}.\tau \Rightarrow \mathcal{S}'(\forall\vec{\alpha}\vec{\varepsilon'}.\tau') \tag{19}$$

Now (4) implies that $\text{fev}(\vec{\varepsilon'}) \cap \mathbf{dom}(\mathcal{S}') = \{\}$ and $\text{fev}(\vec{\varepsilon'}) \cap \text{fev}(\mathbf{rng}(\mathcal{S}')) = \{\}$. So $\mathcal{S}'$ can be moved inside the quantifier in (19), so that we get

$$\mathcal{S}'(\Psi), \forall\vec{\varepsilon'}.\mathcal{S}'(\Psi') \models \forall\vec{\alpha}\vec{\varepsilon'}.\tau \Rightarrow \forall\vec{\alpha}\vec{\varepsilon'}.\mathcal{S}'(\tau') \tag{20}$$

Now let

$$\varepsilon.\vec{\psi} = (\varepsilon.\psi \ominus \psi'_{i_1}, \ldots, \varepsilon.\psi \ominus \psi'_{i_j}) \tag{21}$$

and

$$S'_e = \varepsilon.\vec{\psi}/(\varepsilon'_{i_1}, \ldots, \varepsilon'_{i_j}) \tag{22}$$

We have that

$$\begin{aligned}
S'_e(\{\varepsilon'_{i_1}.\psi'_{i_1}, \ldots, \varepsilon'_{i_j}.\psi'_{i_j}\}) &= \{\varepsilon.S'_e(\psi'_{i_1}) \oplus (\psi \ominus \psi'_{i_1}), \ldots, \varepsilon.S'_e(\psi'_{i_j}) \oplus (\psi \ominus \psi'_{i_j})\} \\
&= \{\varepsilon.\psi'_{i_1} \oplus (\psi \ominus \psi'_{i_1}), \ldots, \varepsilon.\psi'_{i_j} \oplus (\psi \ominus \psi'_{i_j})\} && (11) \\
&= \{\varepsilon.\left\lceil \psi'_{i_1}, \psi \right\rceil, \ldots, \varepsilon.\left\lceil \psi'_{i_j}, \psi \right\rceil\} && \text{lemma 3.13} \\
&= \{\varepsilon.\psi\} && (15) \\
&\subseteq \mathcal{S}'(\Psi) && (18)
\end{aligned} \tag{23}$$

By lemma 6.2 on (20), (22), and (23) we get

$$\mathcal{S}'(\Psi), \forall\vec{\varepsilon'_0}.S'_e(\mathcal{S}'(\Psi')) \models \forall\vec{\alpha}\vec{\varepsilon'_0}.\underline{S'_e}(\tau) \Rightarrow \forall\vec{\alpha}\vec{\varepsilon'_0}.S'_e(\mathcal{S}'(\tau')) \tag{24}$$

where $\vec{\varepsilon'_0} = \vec{\varepsilon'} \setminus\setminus \{i_1, \ldots, i_j\}$.
Now let $\varepsilon.\vec{\varphi}_0 = \varepsilon.\vec{\varphi} \setminus\setminus \{i_1, \ldots, i_j\}$. Let us look at how the substitution $[\varepsilon.\vec{\varphi}_0/\vec{\varepsilon'_0}]$ interacts with $\underline{S'_e}$. From (22) we know that the domain of $\underline{S'_e}$ is disjoint from $\text{fev}(\vec{\varepsilon'_0})$. And as $\underline{\mathbf{rng}(S'_e)} = \{\varepsilon.\varphi\} \subseteq \underline{\Psi}$, (4) we gives us that $\text{fev}(\mathbf{rng}(\underline{S'_e}))$ is disjoint from $\text{fev}(\vec{\varepsilon'_0})$. Thus lemma 3.16 gives the following equation

$$[\varepsilon.\vec{\varphi}_0/\vec{\varepsilon'_0}] \mid \underline{S'_e} = [\varepsilon.\vec{\varphi}_0/\vec{\varepsilon'_0}] \circ \underline{S'_e} \tag{25}$$

And from the way S'_e, $\varepsilon.\vec{\varphi}_0$, and $\vec{\varepsilon'_0}$ were defined we have that

$$[\varepsilon.\vec{\varphi}_0/\vec{\varepsilon'_0}] \mid \underline{S'_e} = [\varepsilon.\vec{\varphi}/\vec{\varepsilon'}] \tag{26}$$

And as $\underline{\mathcal{S}'(\Psi)} = \underline{\Psi}$, (2) can be rewritten like this

$$\underline{\mathcal{S}'(\Psi)} \models [\varepsilon.\vec{\varphi}_0/\vec{\varepsilon'_0}](\underline{S'_e}(\tau)) \tag{27}$$

So by induction on (24), (27), and the fact that $\mid \varepsilon.\vec{\varphi}_0 \mid = \mid \vec{\varepsilon'_0} \mid$, $\text{aref}(\varepsilon.\vec{\varphi}_0) \subseteq \text{aref}(\varepsilon.\vec{\varphi}) \subseteq \underline{\mathcal{S}'(\Psi)}$ we get that there exist $\mathcal{S}''$ and S''_e such that

$$\mathcal{S}'' = \text{Instantiate}(\mathcal{S}'(\Psi), \forall\vec{\varepsilon'_0}.S'_e(\mathcal{S}'(\Psi')), \varepsilon.\vec{\varphi}_0) \tag{28}$$

$$\underline{S''_e} = \varepsilon.\vec{\varphi}_0/\vec{\varepsilon'_0} \tag{29}$$

$$\mathcal{S}''(\mathcal{S}'(\Psi)), \{\} \models \forall\vec{\alpha}.\underline{S''_e}(\underline{S'_e}(\tau)) \Rightarrow \forall\vec{\alpha}.S''_e(\mathcal{S}''(S'_e(\mathcal{S}'(\tau')))) \tag{30}$$

where

$$\mathcal{S}'' \text{ is a multiplicity substitution on } \Psi \tag{31}$$

and

$$\mathbf{dom}(\mathcal{S}'') = \mathrm{fpev}(\varepsilon.\vec{\varphi}_0) \tag{32}$$

Now from (15), (17), (22), (23), and the definition of Instantiate we get that

$$\mathcal{S} = \mathrm{Instantiate}(\Psi, \forall\vec{\varepsilon'}.\Psi', \varepsilon.\vec{\varphi}) \tag{28}$$

where

$$\mathcal{S} = \mathcal{S}'' \circ \mathcal{S}' \tag{29}$$

Now pick $S_e = S''_e \circ S'_e$. We have that

$$\begin{aligned}
\underline{S_e} &= \underline{S''_e \circ S'_e} \\
&= \underline{S''_e} \circ \underline{S'_e} && _ \text{ distr. over } \circ \\
&= [\varepsilon.\vec{\varphi}_0/\vec{\varepsilon'_0}] \circ \underline{S'_e} && (24) \\
&= [\varepsilon.\vec{\varphi}/\vec{\varepsilon'}] && (26)
\end{aligned} \tag{30}$$

We must show that

$$S''_e(\mathcal{S}''(S'_e(\mathcal{S}'(\tau')))) = S_e(\mathcal{S}(\tau')) \tag{31}$$

From (4) we have that $\mathbf{dom}(S'_e) \cap \mathbf{dom}(\mathcal{S}'') = \emptyset$, $\mathbf{dom}(S'_e) \cap \mathrm{fev}(\mathbf{rng}(\mathcal{S}'')) = \emptyset$. Moreover, from (27) and the fact that $\underline{\mathbf{rng}(S'_e)} = \{\varepsilon.\varphi\}$, which is a ground segment of the arrow effects in $\varepsilon.\vec{\varphi}$, we get that $\mathbf{dom}(\mathcal{S}'') \cap \mathrm{fev}(\underline{\mathbf{rng}(S'_e)}) = \emptyset$. So applying lemma 3.16 twice gives us $S'_e \circ \mathcal{S}'' = \mathcal{S}'' \circ S'_e$, which proves (31). Thus (25) can be rewritten like this

$$\mathcal{S}(\Psi), \{\} \models \forall\vec{\alpha}.\underline{S_e}(\tau) \Rightarrow \forall\vec{\alpha}.S_e(\mathcal{S}(\tau'))$$

It is follows directly from (17) and (26) that

$$\mathcal{S} \text{ is a multiplicity substitution on } \Psi$$

And from (17) and (27) it follows that

$$\mathbf{dom}(\mathcal{S}) = \vec{\varepsilon'}$$

b. Every $\varepsilon.\varphi \in \Phi$ is cyclic. Suppose Ψ' contains $\{\varepsilon'_{i_1}.\psi'_{i_1}, \ldots, \varepsilon'_{i_j}.\psi'_{i_j}\}$. Define

$$\mathcal{S}' = \bigcup_{1 \le p \le j} \left\{ \varepsilon_{i_p} \mapsto \varepsilon_{i_p}.\varphi_{i_p}^{\infty} \right\} \tag{32}$$

$\mathcal{S}'$ is clearly a multiplicity substitution on Ψ. For by (3) it holds for all $\varepsilon.\varphi \in \Phi$ that $\varphi = \underline{\hat{\Psi}}(\varepsilon)$, so $\mathcal{S}'$ cannot introduce new atomic effects. It simply sets all multiplicities in the effect to ∞. So by lemma 4.11 on (0) we get

$$\mathcal{S}'(\Psi), \mathcal{S}'(\forall\vec{\varepsilon'}.\Psi') \models \forall\vec{\alpha}\vec{\varepsilon'}.\tau \Rightarrow \mathcal{S}'(\forall\vec{\alpha}\vec{\varepsilon'}.\tau') \tag{33}$$

By (4) we have that $\mathrm{fev}(\vec{\varepsilon'}) \cap \mathbf{dom}(\mathcal{S}) = \{\}$ and by (3) we also get that $\mathrm{fev}(\vec{\varepsilon'}) \cap \mathrm{fev}(\mathbf{rng}(\mathcal{S})) = \{\}$. So $\mathcal{S}$ can be moved inside the quantifier in (33), which gives

$$\mathcal{S}'(\Psi), \forall\vec{\varepsilon'}.\mathcal{S}'(\Psi') \models \forall\vec{\alpha}\vec{\varepsilon'}.\tau \Rightarrow \forall\vec{\alpha}\vec{\varepsilon'}.\mathcal{S}'(\tau') \tag{34}$$

Define

$$S'_e = \{\varepsilon'_{i_1} \mapsto \varepsilon_{i_1}.\varphi^{\infty}_{i_1}, \ldots, \varepsilon'_{i_j} \mapsto \varepsilon_{i_j}.\varphi^{\infty}_{i_j}\} \tag{35}$$

From (3) and (2) we know that

$$[\varepsilon.\vec{\varphi}/\vec{\varepsilon'}](\{\varepsilon'_{i_1}.\underline{\psi_{i_1}}', \ldots, \varepsilon'_{i_j}.\underline{\psi_{i_j}}'\}) = \Phi \tag{36}$$

Now it holds that

$$\neg\exists m \in \{1, \ldots, k\} \setminus \{i_1, \ldots, i_j\} : \exists n \in \{i_1, \ldots, i_j\} : \varepsilon'_m \in \mathrm{fev}(\psi'_n) \tag{37}$$

Otherwise, by (36), it would have been the case that $\varepsilon_m \in \varphi$ for some $\varepsilon.\varphi \in \Phi$, contradicting the fact that Φ is a ground segment of $\{\varepsilon_1.\varphi_1, \ldots, \varepsilon_k.\varphi_k\}$. We have that

$$\begin{aligned}
[\varepsilon.\vec{\varphi}/\vec{\varepsilon'}](\{\varepsilon'_{i_1}.\underline{\psi_{i_1}}', \ldots, \varepsilon'_{i_j}.\underline{\psi_{i_j}}'\}) &= \{\varepsilon_{i_1}.[\varepsilon.\vec{\varphi}/\vec{\varepsilon'}](\underline{\psi_{i_1}}'), \ldots, \varepsilon_{i_j}.[\varepsilon.\vec{\varphi}/\vec{\varepsilon'}](\underline{\psi_{i_j}}')\} \\
&= \{\varepsilon_{i_1}.\underline{S'_e}(\underline{\psi_{i_1}}'), \ldots, \varepsilon_{i_j}.\underline{S'_e}(\underline{\psi_{i_j}}')\} \qquad (37) \\
&= \underline{S'_e}(\{\varepsilon'_{i_1}.\underline{\psi_{i_1}}', \ldots, \varepsilon'_{i_j}.\underline{\psi_{i_j}}'\}) \qquad (38)
\end{aligned}$$

Now it holds that

$$\begin{aligned}
S'_e(\{\varepsilon'_{i_1}.\psi'_{i_1}, \ldots, \varepsilon'_{i_j}.\psi'_{i_j}\}) &= \{\varepsilon.\varphi^{\infty}_{i_1}, \ldots, \varepsilon.\varphi^{\infty}_{i_j}\} \quad (38), (3) \text{ and } (2) \\
&\subseteq (\mathcal{S}'(\Psi)) \qquad (3) \text{ and } (32) \qquad (39)
\end{aligned}$$

The rest of the proof of this case is identical to case **a** from (24) on: first lemma 6.2 is used on (34) and (39), and then induction is used.

■

Appendix C, proof of lemma 4.11

lemma 4.11.*Let $\mathcal{S}$ be a multiplicity substitution on the set of arrow effects Ψ. We have that*

a) If $\Psi \models \tau \Rightarrow \tau'$ then $\mathcal{S}$ is a multiplicity substitution on τ' and $\mathcal{S}(\Psi) \models \tau \Rightarrow \mathcal{S}(\tau')$.

b) If $\Psi \models \mu \Rightarrow \mu'$ then $\mathcal{S}$ is a multiplicity substitution on μ' and $\mathcal{S}(\Psi) \models \mu \Rightarrow \mathcal{S}(\mu')$.

c) If $\Psi, \Xi \models \sigma \Rightarrow \sigma'$, then $\mathcal{S}$ is a multiplicity substitution on σ' and Ξ, and $\mathcal{S}(\Psi), \mathcal{S}(\Xi) \models \sigma \Rightarrow \mathcal{S}(\sigma')$.

d) If $\Psi, EE \models TE \Rightarrow TE'$, then $\mathcal{S}$ is a multiplicity substitution on TE' and EE, and $\mathcal{S}(\Psi), \mathcal{S}(EE) \models TE \Rightarrow \mathcal{S}(TE')$.

e) If $\Psi \models \psi$, then $\mathcal{S}$ is a multiplicity substitution on ψ and $\mathcal{S}(\Psi) \models \mathcal{S}(\psi)$.

Proof.
Assume

$$\mathcal{S} \text{ is a multiplicity substitution on } \Psi \tag{0}$$

a) Assume

$$\Psi \models \tau \Rightarrow \tau' \tag{1}$$

(1) implies $\underline{\Psi} \models \tau$, and by (0) this is the same as $\underline{\mathcal{S}(\Psi)} \models \tau$. So by lemma 4.7 there exists a type τ'' such that $\mathcal{S}(\Psi) \models \tau \Rightarrow \tau''$, and as we pointed out in the proof of lemma 4.7, that τ'' is obtained from τ by exchanging all plain arrow effects in τ by their corresponding multiplicity counterparts in $\mathcal{S}(\Psi)$. So in other words $\tau'' = \mathcal{S}(\tau')$, as it make no difference whether $\mathcal{S}$ is applied to the arrow effects before or after they are loaded into τ.

b) Follows directly from a).

c) Assume

$$\Psi, \Xi \models \sigma \Rightarrow \sigma' \tag{2}$$

(2) implies that we can pick $\vec{\varepsilon}$ and $\vec{\rho}$ such that

$$\Xi = \forall \vec{\rho}\vec{\varepsilon}.\Psi' \qquad \sigma = \forall \vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau \qquad \sigma' = \forall \vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau' \tag{3}$$

and

$$\mathrm{fev}(\vec{\varepsilon}) \cap \mathrm{fev}(\Psi) = \{\} \qquad \mathrm{frv}(\vec{\rho}) \cap \mathrm{fev}(\Psi) = \{\} \tag{4}$$

$$\mathrm{fev}(\sigma) \subseteq \mathrm{fev}(\Psi) \qquad \mathrm{frv}(\sigma) \subseteq \mathrm{frv}(\Psi) \tag{5}$$

$$\Psi \cup \Psi' \models \tau \Rightarrow \tau' \tag{6}$$

Suppose further that $\vec{\varepsilon}$ was chosen so that

$$\mathrm{fev}(\vec{\varepsilon}) \cap \mathbf{dom}(\mathcal{S}) = \{\} \quad \mathrm{fev}(\vec{\varepsilon}) \cap \mathrm{fev}(\mathbf{rng}(\mathcal{S})) = \{\} \quad \mathrm{frv}(\vec{\rho}) \cap \mathrm{frv}(\mathbf{rng}(\mathcal{S})) = \{\} \tag{7}$$

We will now show that

$$\mathcal{S} \text{ is a multiplicity substitution on } \Psi \cup \Psi' \tag{8}$$

By (0) we only have to show that $\mathcal{S}$ is a multiplicity substitution on Ψ'. Let $\varepsilon'.\psi'$ be an arbitrary arrow effect in Ψ'. We distinguish between two cases.

1. $\varepsilon' \in \text{fev}(\Psi)$. By the multiplicity consistency of $\Psi \cup \Psi'$ we have that $\varepsilon'.\psi' \in \Psi$, and so by (0) it holds that $\underline{\varepsilon'.\psi'} = \underline{\mathcal{S}(\varepsilon'.\psi')}$.
2. $\varepsilon' \notin \text{fev}(\Psi)$. From (5) we have that $\varepsilon' \in \text{fev}(\vec{\varepsilon})$ and therefore, by 7, $\varepsilon' \notin \mathbf{dom}(\mathcal{S})$. Thus $\mathcal{S}(\varepsilon'.\psi') = \varepsilon'.\mathcal{S}(\psi')$, so we only have to show that $\underline{\psi'} = \underline{\mathcal{S}(\psi')}$. To do this, we will show that $\mathcal{S}$ does not any new atomic effects to ψ'. More formally, we will show that for all effect variables $\varepsilon \in \mathbf{dom}(\mathcal{S}) \cap \text{fev}(\psi')$ there exists ψ such that $\varepsilon.\psi = \mathcal{S}(\varepsilon)$ and $\underline{\psi} \subseteq \underline{\psi'}$. Let ε be an arbitrary effect variable in $\mathbf{dom}(\mathcal{S}) \cap \text{fev}(\psi')$. By (7) and (5) it must hold that $\varepsilon \in \text{fev}(\Psi)$. From (0) we know that $\underline{\mathcal{S}(\varepsilon.\hat{\Psi}(\varepsilon))} = \underline{\varepsilon.\hat{\Psi}(\varepsilon)}$, so there exists ψ such that $\varepsilon.\psi = \mathcal{S}(\varepsilon)$ and $\underline{\psi} \subseteq \underline{\hat{\Psi}(\varepsilon)}$. But by the multiplicity consistency of $\Psi \cup \Psi'$ we know that $\underline{\hat{\Psi}(\varepsilon)} \subseteq \underline{\psi'}$, and so it also holds that $\underline{\psi} \subseteq \underline{\psi'}$.

This establishes (8).

Now by a) on (8) and (6) we get that

$$\mathcal{S}(\Psi) \cup \mathcal{S}(\Psi') \models \tau \Rightarrow \mathcal{S}(\tau') \tag{9}$$

By (3), (4), (9), and definition 4.5 we get

$$\mathcal{S}(\Psi), \forall \vec{\rho}\vec{\alpha}\vec{\varepsilon}.\mathcal{S}(\Psi') \models \forall \vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau \Rightarrow \forall \vec{\rho}\vec{\alpha}\vec{\varepsilon}.\mathcal{S}(\tau') \tag{10}$$

which by (7) is the same as

$$\mathcal{S}(\Psi), \mathcal{S}(\forall \vec{\rho}\vec{\varepsilon}.\Psi') \models \forall \vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau \Rightarrow \mathcal{S}(\forall \vec{\rho}\vec{\alpha}\vec{\varepsilon}.\tau') \tag{11}$$

d) This result is generalised from b) by pointwise extension.

e) Pick ε such that

$$\varepsilon \notin \text{fev}(\Psi) \cup \text{fev}(\psi) \cup \mathbf{dom}(\mathcal{S}) \tag{12}$$

We know, by the definition of $\models$, that

$$\Psi \cup \{\varepsilon.\psi\} \text{ is multiplicity-consistent} \tag{13}$$

From (12), (13), and the definition of multiplicity consistency we know that

$$\text{fev}(\psi) \subseteq \text{fev}(\Psi) \tag{14}$$

By lemma 4.10

$$\mathcal{S}(\Psi) \text{ is multiplicity-consistent} \tag{15}$$

We must now show that $\mathcal{S}(\Psi \cup \{\varepsilon.\psi\})$ is multiplicity-consistent. As $\varepsilon \notin \mathbf{dom}(\mathcal{S})$, the first requirement of the definition of multiplicity consistency is obviously satisfied. So we only need to show that $\underline{\mathcal{S}(\Psi \cup \{\varepsilon.\psi\})}$ is effect-consistent. This can be done by showing that $\mathcal{S}$ is a multiplicity substitution on $\Psi \cup \{\varepsilon.\psi\}$. We already know that $\mathcal{S}$ is a multiplicity substitution on Ψ and that $\varepsilon \notin \mathbf{dom}(\mathcal{S})$, so the problem only amounts to showing that $\underline{\mathcal{S}(\psi)} = \underline{\psi}$.

Now let ε' be an arbitrary effect variable in $\text{fev}(\psi)$. We will show the application of $\mathcal{S}$ to ε' will not contribute with any new atomic effects to $\underline{\psi}$. We have that

$$\begin{aligned}
\underline{\mathcal{S}(\psi)} &= \underline{\mathcal{S}(\underline{\psi})} && \text{lemma 3.18} \\
&= \underline{\mathcal{S}(\{\varepsilon'\} \cup \underline{\hat{\Psi}(\varepsilon')} \cup \varphi_{rest})} && \text{13 and 14} \\
&= \underline{\mathcal{S}(\{\varepsilon'\} \cup \underline{\hat{\Psi}(\varepsilon')})} \cup \underline{\mathcal{S}(\varphi_{rest})} \\
&= \{\varepsilon'\} \cup \underline{\hat{\Psi}(\varepsilon')} \cup \underline{\mathcal{S}(\varphi_{rest})} && \mathcal{S} \text{ is a multiplicity substitution on } \Psi
\end{aligned}$$

As ε' was arbitrary, we can conclude that $\mathcal{S}$ is a multiplicity substitution on ψ.

■

References

[Bak90] Henry G. Baker. "Unify and Conquer (Garbage, Updating, Aliasing, ...) in Functional Languages", *Proceedings of the 1990 ACM conference on Lisp and Functional Programming*, Nice, France, June 27-29, 1990.

[DM82] Luis Damas & Robin Milner. "Principal type-schemes for functional programs", *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages*, 1982, Albuquerque, New Mexico.

[LH83] Henry Lieberman & Carl Hewitt, "A Real-Time Garbage Collector Based on the Lifetimes of Objects", *CACM 26, 6*, June 1983, 419-429.

[Hay91] Barry Hayes. "Using Key Object Opportunism to Collect Old Objects", *Proceedings: Conference on Object-Oriented Programming Systems, Languages, and Applications*, 6-11, Oct. 1991, Phoenix, Arizona. *Sigplan Notices*, Vol. 26, Number 11, Nov 1991.

[Hen93] Fritz Henglein. "Type inference with polymorphic recursion", *ACM transactions on Programming Languages and Systems*, 15(2):253, April 1993.

[Pau91] Laurence C. Paulson. *ML for the Working Programmer*, Cambridge University Press, 1991.

[Tof88] Mads Tofte. *Operational Semantics and Polymorphic Type Inference*, PhD thesis, Department of Computer Science, Edinburgh University, ECS-LFCS-88-54, 1988.

[TT93a] Mads Tofte & Jean-Pierre Talpin. *A Theory of Stack Allocation in Polymorphically Typed Languages*, DIKU-rapport Nr.93/15, Department of Computer Science, University of Copenhagen, Universitetsparken 1, 2100 Copenhagen Ø, Denmark, 1993.

[TT93b] Mads Tofte & Jean-Pierre Talpin. "Implementation of the Call-by-Value λ-calculus using a Stack of Regions.", *POPL 93*, Portland, 1993.

[Wil92] Paul R. Wilson. "Uniprocessor Garbage Collection Techniques", *Proceedings: Memory Management, Intl. Workshop, IWMM92*, 1992, ed. Y. Dekkers and J. Cohen, pages 1-42, Springer Verlag.