Implementation of the Typed Call-by-Value λ-calculus using a Stack of Regions

Mads Tofte, University of Copenhagen[*]

Jean-Pierre Talpin, European Computer-Industry Research Center [†]

Abstract

We present a translation scheme for the polymorphically typed call-by-value λ-calculus. All runtime values, including function closures, are put into *regions*. The store consists of a stack of regions. Region inference and effect inference are used to infer where regions can be allocated and de-allocated. Recursive functions are handled using a limited form of polymorphic recursion. The translation is proved correct with respect to a store semantics, which models a region-based run-time system. Experimental results suggest that regions tend to be small, that region allocation is frequent and that overall memory demands are usually modest, even without garbage collection.

1 Introduction

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

In heap-based storage management schemes[4,19, 18], allocation is separate from de-allocation, the latter being handled by garbage collection. This separation is useful when the lifetime of values is not apparent from the source program. Heap-based schemes are less eager to reuse memory. Generational garbage collection collects young objects when the allocation space is used up. Hayes[11] discusses how to reclaim large, old objects.

Garbage collection can be very fast. Indeed, there is a much quoted argument that the amortized cost of copying garbage collection tends to zero, as memory tends to infinity[2, page 206]. Novice functional programmers often report that on their machines, memory is a constant, not a variable, and that this constant has to be uncomfortably large for their programs to run well. The practical ambition of our work is to reduce the required size of this constant significantly. We shall present measurements that indicate that our translation scheme holds some promise in this respect.

In this paper, we propose a translation scheme for Milner's call-by-value λ-calculus with recursive functions and polymorphic `let`[22,7]. The key features of our scheme are:

1. It determines lexically scoped lifetimes for all runtime values, including function closures, base values and records;
2. It is provably safe;
3. It is able to distinguish the lifetimes of different invocations of the same recursive function;

This last feature is essential for obtaining good memory usage (see Section 5).

Our model of the runtime system involves a stack of *regions*, see Figure 1. We do not expect always to be able to determine the size of a region when we allocate it. Part of the reason for this is that we consider recursive datatypes, such as lists, a must; the size of a region which is supposed to hold the spine of a list, say, cannot in general be determined when the region is allocated. Therefore, not all regions can be allocated on a hardware stack, although regions of known size can.

Our allocation scheme differs from the classical stack allocation scheme in that it admits functions as first-class values and is intended to work for recursive datatypes. (So far, the only recursive datatype we have dealt with is lists.)

[*] Postal address: Department of Computer Science (DIKU), University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark; email: `tofte@diku.dk`.

[†] Work done while at Ecole des Mines de Paris. Current address: European Computer-Industry Research Center (ECRC GmbH), Arabella Straße 17, D-81925 München; email: `jp@ecrc.de`

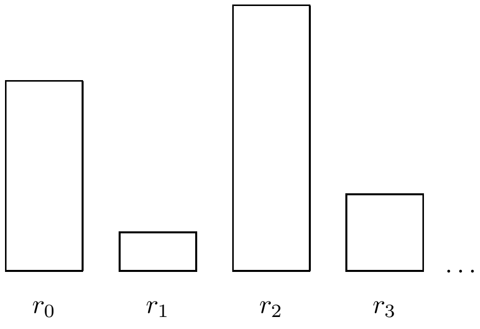

Figure 1: The store is a stack of regions; a region is a box in the picture.

Ruggieri and Murtagh[28] propose a stack of regions in conjunction with a traditional heap. Each region is associated with an activation record (this is not necessarily the case in our scheme). They use a combination of interprocedural and intraprocedural data-flow analysis to find suitable regions to put values in. We use a type-inference based analysis. They consider updates, which we do not. However, we deal with polymorphism and higher-order functions, which they do not.

Inoue *et al.*[15] present an interesting technique for compile-time analysis of runtime garbage cells in lists. Their method inserts pairs of HOLD and RECLAIMη instructions in the target language. HOLD holds on to a pointer, p say, to the root cell of its argument and RECLAIMη collects those cells that are reachable from p and fit the path description η. HOLD and RECLAIM pairs are nested, so the HOLD pointers can be held in a stack, not entirely unlike our stack of regions. In our scheme, however, the unit of collection is one entire region, i.e., there is no traversal of values in connection with region collection. The path descriptions of Inoue *et al.* make it possible to distinguish between the individual members of a list. This is not possible in our scheme, as we treat all the elements of the same list as equal. Inoue *et al.* report a 100% reclamation rate for garbage *list* cells produced by Quicksort[15, page 575]. We obtain a 100% reclamation rate (but for 1 word) for *all* garbage produced by Quicksort, without garbage collection (see Section 5).

Hudak[13] describes a reference counting scheme for a first-order call-by-value functional language. Reference counting may give more precise use information, than our scheme, as we only distinguish between "no use" and "perhaps some use."

Georgeff[10] describes an implementation scheme for typed lambda expressions in so-called simple form together with a transformation of expressions into simple form. The transformation can result in an increase in the number of evaluation steps by an arbitrarily large factor[10, page 618]. Georgeff also presents an implementation scheme which does not involve translation, although this relies on not using call-by-value reduction, when actual parameters are functions.

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

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

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

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

The device we use for grouping values according to regions is unification of region variables, using essentially the idea of Baker[3], namely that two value-producing expressions e_1 and e_2 should be given the same "`at` ρ" annotation, if and only if type checking, directly or indirectly, unifies the type of e_1 and e_2. Baker does not prove safety, however, nor does he deal with polymorphism.

To obtain good separation of lifetimes, we introduce *explicit region polymorphism*, by which we mean that regions can be given as arguments to functions at runtime. For example, the successor function $\texttt{succ} = \lambda x.x + 1$ is compiled into

$$\begin{array}{l}\Lambda\texttt{[}\rho\texttt{,}\rho'\texttt{]}.\lambda x.\texttt{letregion}\ \rho''\\ \qquad\qquad \texttt{in}\ (x + (1\ \texttt{at}\ \rho''))\ \texttt{at}\ \rho'\\ \qquad\qquad \texttt{end}\end{array}$$

which has the type scheme

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

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

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

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

We use effect inference[20,21,14] to find out where to wrap `letregion` ρ `in` ... `end` around an expression. Most work on effect inference uses the word "effect" as a short-hand for "side-effect". We have no side-effects in our source language — our effects are side-effects relative to an underlying region-based store model.

The idea that effect inference makes it possible to delimit regions of memory and delimit their lifetimes goes back to early work on effect systems[6]. Lucassen and Gifford[21] call it *effect masking*; they prove that effect masking is sound with respect to a store semantics where regions are not reused. Talpin[29] and Talpin and Jouvelot[30] present a polymorphic effect system with effect masking and prove that it is sound, with respect to a store semantics where regions are not reused.

We have found the notion of memory reuse surprisingly subtle, due to, among other things, pointers into de-allocated regions. Since memory reuse is at the heart of our translation scheme, we prove that our translation rules are sound with respect to a region-based operational semantics, where regions explicitly are allocated and de-allocated. This is the main technical contribution of this paper.

The rest of this paper is organised as follows. The source and target languages are presented in Section 2. The translation scheme is presented in Section 3. The correctness proof is in Section 4. In Section 5 we discuss strengths and weaknesses of the translation and give experimental results.

Due to space limitations, most proofs have been omitted. Detailed proofs (and an inference algorithm) are available in a technical report[31].

2 Source and target languages

2.1 Source language

We assume a denumerably infinite set Var of *variables*. Each variable is either an *ordinary* variable, x, or a *letrec* variable, f. The grammar for the source language is[1]

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

A *finite* map is a map with finite domain. The domain and range of a finite map f are denoted Dom(f) and Rng(f), respectively. When f and g are finite maps, $f + g$ is the finite map whose domain is $\mathrm{Dom}(f) \cup \mathrm{Dom}(g)$ and whose value is $g(x)$, if $x \in \mathrm{Dom}(g)$, and $f(x)$ otherwise. $f \downarrow A$ means the restriction of f to A.

A *(non-recursive) closure* is a triple $\langle x, e, E\rangle$, where E is an *environment*, i.e. a finite map from variables to values. A *(recursive) closure* takes the form $\langle x, e, E, f\rangle$ where f is the name of the function in question. A value is either an integer or a closure. Evaluation rules appear below.

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

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

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

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

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

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

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

2.2 Target language

Let ρ range over a denumerably infinite set RegVar of *region variables*. Let r range over a denumerably infinite set RegName = $\{\texttt{r1}, \texttt{r2}, \ldots\}$ of *region names*. Region names serve to identify regions uniquely at runtime, i.e. a *store*, s, is a finite map from region names to regions. Let p range over the set of *places*, a place being either a region variable or a region name. The grammar for the target language is:

$$\begin{array}{lcl} p & ::= & \rho \mid r \\ e & ::= & x \mid \lambda x.e\ \texttt{at}\ p \mid e_1 e_2 \\ & \mid & \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end} \\ & \mid & \texttt{letrec}\ f\texttt{[}\vec{\rho}\texttt{]}\texttt{(}x\texttt{)}\ \texttt{at}\ p\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end} \\ & \mid & f\ \texttt{[}\vec{p}\texttt{]}\ \texttt{at}\ p \\ & \mid & \texttt{letregion}\ \rho\ \texttt{in}\ e\ \texttt{end} \end{array}$$

[1] For brevity, we omit pairs and projections from this paper. They are treated in [31].

where $\vec{\rho}$ ranges over finite sequences $\rho_1, \cdots, \rho_k$ of region variables and $\vec{p}$ ranges over finite sequences $p_1, \cdots, p_k$ of places ($k \geq 0$). We write $|\vec{p}|$ for the length of a sequence $\vec{p}$. For any finite set $\{\rho_1, \ldots, \rho_k\}$ of region variables ($k \geq 0$), we write `letregion` $\vec{\rho}$ `in` e `end` for

```
letregion $\rho_1$
in $\cdots$ letregion $\rho_k$ in $e$ end $\cdots$
end
```

A *region* is a finite map from *offsets*, o, to storable values. A *storable value*, *sv*, is either an integer or a closure. A *(plain) closure* is a triple $\langle x, e, VE \rangle$, where e is a target expression and VE is a *variable environment*, i.e. a finite map from variables to addresses. A *region closure* takes the form $\langle \vec{\rho}, x, e, VE \rangle$ where $\vec{\rho}$ is a (possible empty) sequence $\rho_1, \cdots, \rho_k$ of distinct region variables, called the *formal region parameters* of the closure. Region closures represent region-polymorphic functions. For any sequence $\vec{p} = p_1, \cdots, p_k$, the simultaneous substitution of p_i for free occurrences of ρ_i in e ($i = 1 \ldots k$), is written $e[\vec{p}/\vec{\rho}]$.

For simplicity, we assume that all values are boxed. Hence a value v is an *address* $a = (r, o)$, where r is a region name and o is an offset.

A region-based operational semantics appears below. We are brief about indirect addressing. Thus, whenever a is an address (r, o), we write $s(a)$ to mean $s(r)(o)$ and we write $a \in \mathrm{Dom}(s)$ as a shorthand for $r \in \mathrm{Dom}(s)$ and $o \in \mathrm{Dom}(s(r))$. Similarly, when s is a store and *sv* is a storable value, we write $s + \{(r, o) \mapsto sv\}$ as a shorthand for $s + \{r \mapsto (s(r) + \{o \mapsto sv\})\}$. We express the popping of an entire region r from a store s by writing "$s \backslash\!\backslash \{r\}$", which formally means the restriction of s to $\mathrm{Dom}(s) \setminus \{r\}$.

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

$$\frac{VE(x) = v}{s, VE \vdash x \rightarrow v, s} \tag{7}$$

$$\frac{\begin{array}{c} VE(f) = a, \quad s(a) = \langle \vec{\rho}, x, e, VE_0 \rangle \quad |\vec{\rho}| = |\vec{p}| \\ o \notin \mathrm{Dom}(s(r)) \quad sv = \langle x, e[\vec{p}/\vec{\rho}], VE_0 \rangle \end{array}}{s, VE \vdash f\,\texttt{[}\vec{p}\texttt{]}\ \texttt{at}\ r \rightarrow (r, o), s + \{(r, o) \mapsto sv\}} \tag{8}$$

$$\frac{o \notin \mathrm{Dom}(s(r)) \quad a = (r, o)}{s, VE \vdash \lambda x.e\ \texttt{at}\ r \rightarrow a, s + \{a \mapsto \langle x, e, VE \rangle\}} \tag{9}$$

$$\frac{\begin{array}{c} s, VE \vdash e_1 \rightarrow a_1, s_1 \quad s_1(a_1) = \langle x_0, e_0, VE_0 \rangle \\ s_1, VE \vdash e_2 \rightarrow v_2, s_2 \\ s_2, VE_0 + \{x_0 \mapsto v_2\} \vdash e_0 \rightarrow v, s' \end{array}}{s, VE \vdash e_1\ e_2 \rightarrow v, s'} \tag{10}$$

$$\frac{s, VE \vdash e_1 \rightarrow v_1, s_1 \qquad s_1, VE + \{x \mapsto v_1\} \vdash e_2 \rightarrow v, s'}{s, VE \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end} \rightarrow v, s'} \tag{11}$$

$$\frac{\begin{array}{c} o \notin \mathrm{Dom}(s(r)) \qquad VE' = VE + \{f \mapsto (r, o)\} \\ s + \{(r, o) \mapsto \langle \vec{\rho}, x, e_1, VE' \rangle\}, VE' \vdash e_2 \rightarrow v, s' \end{array}}{s, VE \vdash \texttt{letrec f[}\vec{\rho}\texttt{](}x\texttt{) at}\ r\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end} \rightarrow v, s'} \tag{12}$$

$$\frac{r \notin \mathrm{Dom}(s) \quad s + \{r \mapsto \{\}\}, VE \vdash e[r/\rho] \rightarrow v, s_1}{s, VE \vdash \texttt{letregion}\ \rho\ \texttt{in}\ e\ \texttt{end} \rightarrow v, s_1 \backslash\!\backslash \{r\}} \tag{13}$$

For arbitrary finite maps f_1 and f_2, we say that f_2 *extends* f_1, written $f_1 \subseteq f_2$, if $\mathrm{Dom}(f_1) \subseteq \mathrm{Dom}(f_2)$ and for all $x \in \mathrm{Dom}(f_1)$, $f_1(x) = f_2(x)$. We then say that s_2 *succeeds* s_1, written $s_2 \sqsupseteq s_1$ (or $s_1 \sqsubseteq s_2$), if $\mathrm{Dom}(s_1) \subseteq \mathrm{Dom}(s_2)$ and $s_1(r) \subseteq s_2(r)$, for all $r \in \mathrm{Dom}(s_1)$.

Lemma 2.1 *If* $s, VE \vdash e \rightarrow v, s'$ *then* $\mathrm{Dom}(s) = \mathrm{Dom}(s')$ *and* $s \sqsubseteq s'$.

The proof is a straightforward induction on the depth of inference of $s, VE \vdash e \rightarrow v, s'$.

Example The source expression

```
let $x$ = (2,3) in $\lambda\,y$.(fst $x$, $y$)end 5
```

translates into

```
$e'$ $\equiv$letregion $\rho_4, \rho_5$
     in letregion $\rho_6$
        in let $x$ = (2 at $\rho_2$, 3 at $\rho_6$) at $\rho_4$
           in ($\lambda y$.(fst $x$, $y$) at $\rho_1$) at $\rho_5$
           end
        end
        5 at $\rho_3$
     end
```

Notice that ρ_1, ρ_2 and ρ_3 occur free in this expression. That is because they will hold the final result (a fact which the translation infers). To start the evaluation of e', we first allocate three regions, say `r1`, `r2` and `r3`. Then we substitute `r`i for ρ_i in e' ($i = 1..3$). Figure 2 shows three snapshots from the evaluation that follows, namely (a) just after the closure has been allocated; (b) just before the closure is applied and (c) at the end. The maximal depth of the region stack is 6 regions and the final depth is 3 regions. Notice the dangling, but harmless, pointer at (b).

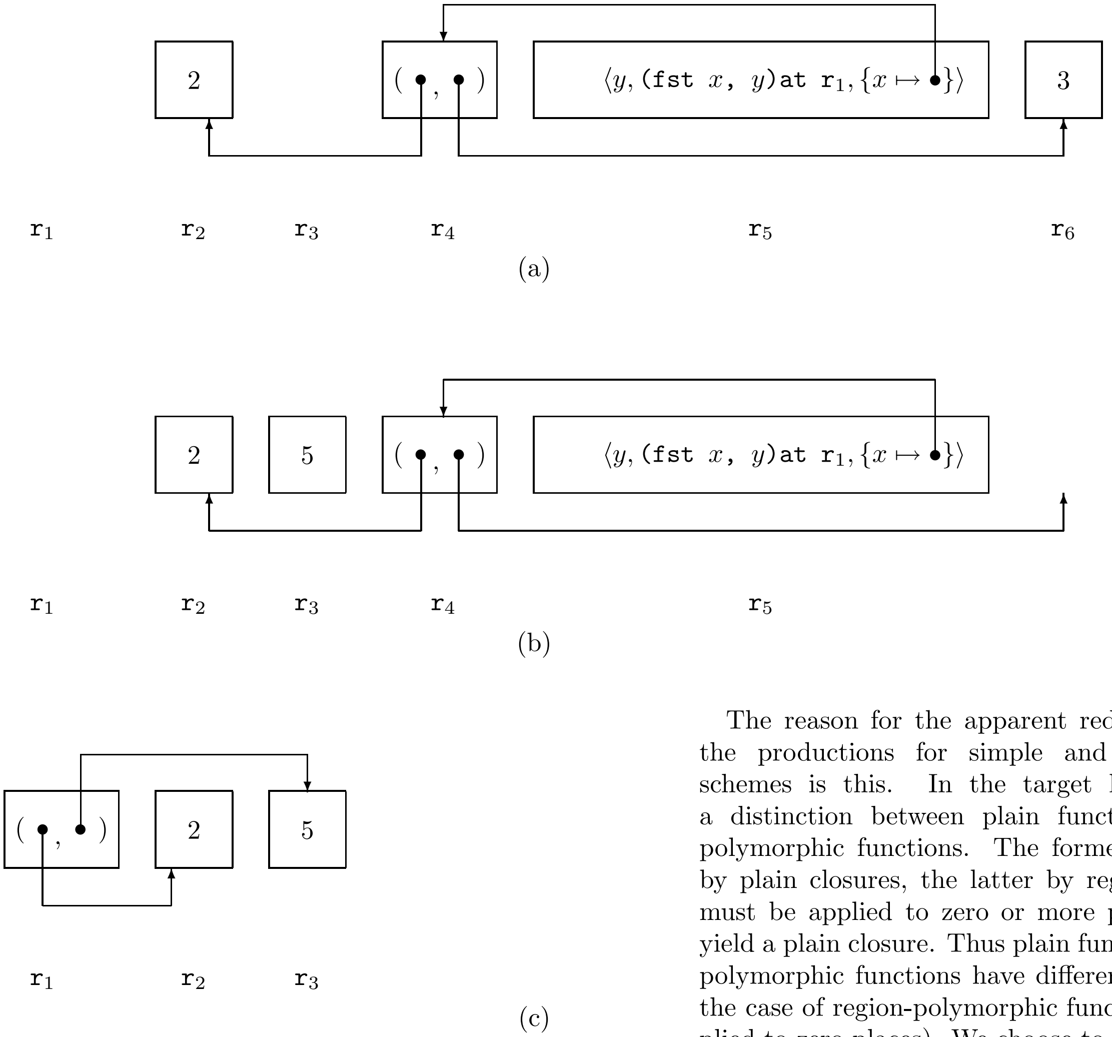

Figure 2: Three snapshots of an evaluation

3 The Translation

Let α and ϵ range over denumerably infinite sets of *type variables* and *effect variables*, respectively. We assume that the sets of type variables, effect variables, region variables and region names are all pairwise disjoint. An *effect*, φ, is a finite set of atomic effects. An *atomic effect* is either a token of the form $\mathbf{get}(\rho)$ or $\mathbf{put}(\rho)$, or it is an effect variable. *Types*, τ, *decorated types*, μ, *simple type schemes*, σ, and *compound type schemes*, π, take the form:

$$
\begin{array}{lcl}
\tau & ::= & \texttt{int} \mid \mu \xrightarrow{\epsilon.\varphi} \mu \mid \alpha \\
\mu & ::= & (\tau, p) \\
\sigma & ::= & \tau \mid \forall\alpha.\sigma \mid \forall\epsilon.\sigma \\
\pi & ::= & \underline{\tau} \mid \forall\alpha.\pi \mid \forall\epsilon.\pi \mid \forall\rho.\pi
\end{array}
$$

The reason for the apparent redundancy between the productions for simple and compound type schemes is this. In the target language there is a distinction between plain functions and region-polymorphic functions. The former are represented by plain closures, the latter by region closures that must be applied to zero or more places in order to yield a plain closure. Thus plain functions and region-polymorphic functions have different effects (even in the case of region-polymorphic functions that are applied to zero places). We choose to represent this distinction in the type system by a complete separation of simple and compound type schemes. Only region-polymorphic functions have compound type schemes. (The underlining in the production $\pi ::= \underline{\tau}$ makes it possible always to tell which kind a type scheme is.) When a type τ is regarded as a type scheme, it is always regarded as a simple type scheme.

An object of the form $\epsilon.\varphi$ (formally a pair (ϵ, φ)) is called an *arrow effect*. Here φ is the effect of calling the function. The "ϵ." is useful for type-checking purposes, as explained in more detail in Appendix A. A *type environment*, *TE*, is a finite map from ordinary variables to pairs of the form (σ, p) and from letrec variables to pairs of the form (π, p).

A *substitution* S is a triple (S_r, S_t, S_e), where S_r is a finite map from region variables to places, S_t is a finite map from type variables to types and S_e is a finite map from effect variables to arrow effects. Its effect is to carry out the three substitutions simultaneously on the three kinds of variables.

For any compound type scheme

$$\pi = \forall\rho_1\cdots\rho_k\forall\alpha_1\cdots\alpha_n\forall\epsilon_1\cdots\epsilon_m.\underline{\tau}$$

and type τ', we say that τ' is *an instance of π (via S)*, written $\pi \geq \tau'$, if there exists a substitution $S = (\{\rho_1 \mapsto p_1, \ldots, \rho_k \mapsto p_k\}, \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}, \{\epsilon_1 \mapsto \epsilon'_1.\varphi_1, \ldots, \epsilon_m \mapsto \epsilon'_m.\varphi_m\})$ such that $S(\tau) = \tau'$. Similarly for simple type schemes.

We infer sentences $TE \vdash e \Rightarrow e' : \mu, \varphi$, read: in TE, e translates to e', which has type and place μ and effect φ. In the example in Section 2.2, μ is $((\mathtt{int}, \rho_2) * (\mathtt{int}, \rho_3), \rho_1)$, φ is $\{\mathbf{put}(\rho_1), \mathbf{put}(\rho_2), \mathbf{put}(\rho_3)\}$ and TE is empty.

Translation $\boxed{TE \vdash e \Rightarrow e' : \mu, \varphi}$

$$\frac{TE(x) = (\sigma, p) \quad \sigma \geq \tau}{TE \vdash x \Rightarrow x : (\tau, p), \emptyset} \tag{14}$$

$$\frac{TE + \{x \mapsto \mu_1\} \vdash e \Rightarrow e' : \mu_2, \varphi \qquad \varphi \subseteq \varphi'}{TE \vdash \lambda x.e \Rightarrow \lambda x.e' \;\mathtt{at}\; p : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, p), \{\mathbf{put}(p)\}} \tag{15}$$

$$\frac{\begin{array}{c} TE \vdash e_1 \Rightarrow e'_1 : (\mu' \xrightarrow{\epsilon.\varphi_0} \mu, p), \varphi_1 \\ TE \vdash e_2 \Rightarrow e'_2 : \mu', \varphi_2 \\ \varphi = \varphi_0 \cup \varphi_1 \cup \varphi_2 \cup \{\epsilon\} \cup \{\mathbf{get}(p)\} \end{array}}{TE \vdash e_1\; e_2 \Rightarrow e'_1\; e'_2 : \mu, \varphi} \tag{16}$$

$$\frac{\begin{array}{c} TE \vdash e_1 \Rightarrow e'_1 : (\tau_1, p_1), \varphi_1 \\ \sigma_1 = \forall\vec{\alpha}\forall\vec{\epsilon}.\tau_1 \quad \mathrm{fv}(\vec{\alpha}, \vec{\epsilon}) \cap \mathrm{fv}(TE, \varphi_1) = \emptyset \\ TE + \{x \mapsto (\sigma_1, p_1)\} \vdash e_2 \Rightarrow e'_2 : \mu, \varphi_2 \end{array}}{\begin{array}{l} TE \vdash \;\mathtt{let}\; x \;\mathtt{=}\; e_1 \;\mathtt{in}\; e_2 \;\mathtt{end} \Rightarrow \\ \qquad \mathtt{let}\; x \;\mathtt{=}\; e'_1 \;\mathtt{in}\; e'_2 \;\mathtt{end} : \mu, \varphi_1 \cup \varphi_2 \end{array}} \tag{17}$$

$$\frac{\begin{array}{c} \pi = \forall\vec{\rho}\forall\vec{\epsilon}.\underline{\tau} \quad \mathrm{fv}(\vec{\rho}, \vec{\epsilon}) \cap \mathrm{fv}(TE, \varphi_1) = \emptyset \\ TE + \{f \mapsto (\pi, p)\} \vdash \lambda\, x.e_1 \Rightarrow \lambda\, x.e'_1 \;\mathtt{at}\; p : (\tau, p), \varphi_1 \\ \pi' = \forall\vec{\alpha}.\pi \quad \mathrm{fv}(\vec{\alpha}) \cap \mathrm{fv}(TE, \varphi_1) = \emptyset \\ TE + \{f \mapsto (\pi', p)\} \vdash e_2 \Rightarrow e'_2 : \mu, \varphi_2 \end{array}}{\begin{array}{l} TE \vdash \;\mathtt{letrec}\; f\mathtt{(}x\mathtt{)} \;\mathtt{=}\; e_1 \;\mathtt{in}\; e_2 \;\mathtt{end} \Rightarrow \\ \qquad \mathtt{letrec}\; f\mathtt{[}\vec{\rho}\mathtt{](}x\mathtt{)at}p \;\mathtt{=}\; e'_1 \;\mathtt{in}\; e'_2 \;\mathtt{end} : \mu, \varphi_1 \cup \varphi_2 \end{array}} \tag{18}$$

$$\frac{\begin{array}{c} TE(f) = (\pi, p') \quad \pi = \forall\rho_1\cdots\rho_k.\forall\vec{\alpha}.\forall\vec{\epsilon}.\underline{\tau'} \\ \pi \geq \tau \text{ via } S \quad \varphi = \{\mathbf{get}(p'), \mathbf{put}(p)\} \end{array}}{TE \vdash f \Rightarrow f\,\mathtt{[}S(\rho_1)\mathtt{,}\ldots\mathtt{,}S(\rho_k)\mathtt{]} \;\mathtt{at}\; p : (\tau, p), \varphi} \tag{19}$$

$$\frac{\begin{array}{c} TE \vdash e \Rightarrow e' : \mu, \varphi \quad \varphi' = \mathrm{Observe}(TE, \mu)(\varphi) \\ \{\rho_1, \ldots, \rho_k\} = \mathrm{frv}(\varphi \setminus \varphi') \end{array}}{TE \vdash e \Rightarrow \mathtt{letregion}\; \rho_1\cdots\rho_k \;\mathtt{in}\; e' \;\mathtt{end} : \mu, \varphi'} \tag{20}$$

For any semantic object A, $\mathrm{frv}(A)$ denotes the set of region variables that occur free in A, $\mathrm{frn}(A)$ denotes the set of region names that occur free in A, $\mathrm{ftv}(A)$ denotes the set of type variables that occur free in A, $\mathrm{fev}(A)$ denotes the set of effect variables that occur free in A and $\mathrm{fv}(A)$ denotes the union of all of the above.

Rule 14 is essentially the usual instantiation rule for polymorphic variables in Milner's type discipline[22, 7]. In rule 18, notice that f can be used region-polymorphically, but not type-polymorphically, inside e'_1.

As for rule 20, assume $TE \vdash e \Rightarrow e' : \mu, \varphi$ and that ρ is a region variable which occurs free in φ, but does not occur free in TE or in μ. *Then ρ is purely local to the evaluation of e', in the sense that the rest of the computation will not access any value stored in ρ.* Since ρ is no longer needed, we can introduce a "$\mathtt{letregion}\;\rho\;\mathtt{in}\;\cdots\;\mathtt{end}$" around e' and discharge any effect containing ρ from φ. Thus, following Talpin and Jouvelot [30], for every effect φ and semantic object A, we define *the observable part of φ with respect to A*, written $\mathrm{Observe}(A)(\varphi)$, to be the following subset of φ:

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

Every expression which is typable in Milner's type system[7] can be translated using our rules, by refraining from using region polymorphism, abstaining from rule 20, choosing a fixed region ρ_0 everywhere and choosing the arrow effect $\epsilon_0.\{\mathbf{get}(\rho_0), \mathbf{put}(\rho_0), \epsilon_0\}$ everywhere. (A good region inference algorithm naturally does the exact opposite and makes all regions as distinct and local as possible.)

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

An example of a translation where region polymorphism comes into play is found in Appendix B.

4 Correctness

In an attempt to prove that the translation rules are sound, one might try to define a "consistency relation" between values in the source language semantics and values in the target language semantics and prove a theorem to the effect that consistency is *preserved*: if E and VE are consistent in store s and e translates to e' and $E \vdash e \rightarrow v$ then there exists a store s' and a value v' such that $s, VE \vdash e' \rightarrow v', s'$ [25,8]. However,

Definition The relation Consistent is the largest relation satisfying:

- Consistent(R, μ, v, s, v') w.r.t. $\varphi \iff$ (writing μ as (τ, p) and v' as (r', o'))
  if $\mathbf{get}(p) \in \varphi$
  then $v' \in \mathrm{Dom}(s)$ and $r' = R(p)$ and
  1) If v is an integer i then $\tau = \mathtt{int}$ and $s(v') = i$;
  2) If v is a closure $\langle x, e, E\rangle$ then $s(v')$ is a closure $\langle x, e', VE\rangle$, for some e' and VE, and there exist TE, R' and e'' such that
     $TE \vdash \lambda x.e \Rightarrow \lambda x.e''$ $\mathtt{at}$ $p : \mu, \{\mathbf{put}(p)\}$
     and Consistent(R, TE, E, s, VE) w.r.t. φ
     and R' and R agree on φ
     and $R'(e'') = e'$;
  3) If v is a closure $\langle x, e, E, f\rangle$ then $s(v') = \langle x, e', VE\rangle$, for some e' and VE, and there exist TE, σ, p', R' and e'' such that
     $TE + \{f \mapsto (\sigma, p')\} \vdash \lambda x.e \Rightarrow \lambda x.e''$ $\mathtt{at}$ $p : \mu, \{\mathbf{put}(p)\}$
     and Consistent$(R, TE + \{f \mapsto (\sigma, p')\}, E + \{f \mapsto v\}, s, VE)$ w.r.t. φ
     and R' and R agree on φ
     and $R'(e'') = e'$;

- Consistent$(R, (\sigma, p), v, s, v')$ w.r.t. $\varphi \iff$ (writing v' as (r', o'))
  if $\mathbf{get}(p) \in \varphi$
  then $v' \in \mathrm{Dom}(s)$ and $r' = R(p)$
  and for all τ, if $\sigma \geq \tau$ then Consistent$(R, (\tau, p), v, s, v')$ w.r.t. φ;

- Consistent$(R, (\pi, p), v, s, v')$ w.r.t. $\varphi \iff$ (writing v' as (r', o'))
  if $\mathbf{get}(p) \in \varphi$
  then v is a recursive closure $\langle x, e, E, f\rangle$
  and $s(v') = \langle \vec{\rho'}, x, e', VE\rangle$, for some $\vec{\rho'}$, e' and VE,
  and there exist TE, R' and e'' such that
  Consistent$(R, TE + \{f \mapsto (\pi, p)\}, E + \{f \mapsto v\}, s, VE)$ w.r.t. φ
  and π can be written in the form $\forall\vec{\rho}.\forall\vec{\alpha}\forall\vec{\epsilon}.\underline{\tau}$
  where none of the bound variables occur free in (TE, p),
  and $TE + \{f \mapsto (\pi, p)\} \vdash \lambda x.e \Rightarrow \lambda x.e''$ $\mathtt{at}$ $p : (\tau, p), \{\mathbf{put}(p)\}$
  and R' and R agree on φ and $R'\langle\vec{\rho}, x, e'', VE\rangle = \langle\vec{\rho'}, x, e', VE\rangle$

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

Figure 3: The definition of Consistent

no matter how we tried to define the notion of consistency, we were unable to prove such a preservation theorem.

The key observation is that consistency, in an absolute sense, is *not* preserved – rather it *decreases monotonically*. In the example in Section 2, the consistency that existed between the pair (2,3) and its representation in the store at point (a) is obviously only partially preserved at point (b). The saving fact is that there is always enough consistency left!

We therefore define consistency *with respect to an effect*, which, intuitively speaking, stands for the effect of executing the rest of the program. A *region environment*, R is a finite map from region variables to places. R *connects* φ *to* s, if $\mathrm{frv}(R(\varphi)) = \emptyset$ and $\mathrm{frn}(R(\varphi)) \subseteq \mathrm{Dom}(s)$. Two region environments R_1 and R_2 *agree on effect* φ, if $R_1(\rho) = R_2(\rho)$, for all $\rho \in \mathrm{frv}(\varphi)$.

Region environments can be applied to target expressions and even to region closures $\langle\vec{\rho}, x, e, VE\rangle$ provided one renames bound region variables, when necessary, to avoid capture.

The consistency relation is defined in Figure 3. It is the maximal fixed point of a monotonic operator on sets; properties about consistency can therefore be proved using co-induction [23]. For example, one can prove[31] that consistency is preserved under increasing stores, with respect to decreasing effects:

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

This lemma is a special case of a the following lemma, which we shall see an application of later:

Lemma 4.2 *If* Consistent(R_1, μ, v, s_1, v') w.r.t. φ_1 *and* $\varphi_2 \subseteq \varphi_1$ *and* R_2 *and* R_1 *agree on* φ_2 *and* $s_1 \downarrow \mathrm{frn}(R_2\varphi_2) \sqsubseteq s_2$ *then* Consistent(R_2, μ, v, s_2, v') w.r.t. φ_2.

The next lemma states that, in certain circumstances, consistency can even be preserved with respect to increasing effects!

Lemma 4.3 *If* Consistent(R, TE, E, s, VE) w.r.t. φ *and* $\rho \notin \mathrm{frv}(TE, \varphi)$*,* $r \notin \mathrm{Dom}(s)$ *and* $\varphi' \subseteq \{\mathbf{put}(\rho), \mathbf{get}(\rho)\} \cup \{\epsilon_1, \ldots, \epsilon_k\}$*, where* $\epsilon_1, \ldots, \epsilon_k$ *are effect variables* $(k \geq 0)$ *then* Consistent$(R + \{\rho \mapsto r\}, TE, E, s + \{r \mapsto \{\}\}, VE)$ w.r.t. $\varphi' \cup \varphi$.

The proof of Lemma 4.3 has to deal with the possibility that there are old pointers into r[31]. This lemma will be used in the proof of the soundness of rule 20.

Here is the main theorem:

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

Proof The proof is by depth of the derivation of $E \vdash e \to v$ with an inner induction on the number of rule 20 steps that terminate the proof of $TE \vdash e \Rightarrow e' : \mu, \varphi$. The inner inductive argument is independent of e. We show the (inner) inductive step concerning `letregion` and the (outer) case concerning function application, as examples. In both cases, we assume

$$TE \vdash e \Rightarrow e' : \mu, \varphi \tag{21}$$

$$\text{Consistent}(R, TE, E, s, VE) \text{ w.r.t. } \varphi \cup \varphi' \tag{22}$$

$$E \vdash e \to v \tag{23}$$

$$R \text{ connects } \varphi \cup \varphi' \text{ to } s \tag{24}$$

$$R' \text{ and } R \text{ agree on } \varphi \cup \varphi' \tag{25}$$

$\boxed{\text{Inner proof case: the } \texttt{letregion} \text{ rule was applied}}$

Assume (21) takes the form $TE \vdash e \Rightarrow$ `letregion` $\rho_1 \cdots \rho_k$ `in` $e'_1 : \mu, \varphi$ which must have been inferred by rule 20 on the premises

$$TE \vdash e \Rightarrow e'_1 : \mu, \varphi^+ \tag{26}$$

$$\varphi = \text{Observe}(TE, \mu)(\varphi^+) \tag{27}$$

$$\{\rho_1, \ldots, \rho_k\} = \mathrm{frv}(\varphi^+ \setminus \varphi) \tag{28}$$

Without loss of generality we can choose $\rho_1, \ldots, \rho_k$ such that $\{\rho_1, \ldots, \rho_k\} \cap \mathrm{frv}(\varphi') = \emptyset$ as well as (27)-(28). Thus $\{\rho_1, \ldots, \rho_k\} \cap \mathrm{frv}(TE, \varphi \cup \varphi') = \emptyset$. Let $r_1, \ldots, r_k$ be distinct addresses none of which are in Dom(s). Then by repeated application of Lemma 4.3 starting from (22) we get

$$\text{Consistent}(R^+, TE, E, s^+, VE) \text{ w.r.t. } \varphi^+ \cup \varphi' \tag{29}$$

where $R^+ = R + \{\rho_1 \mapsto r_1, \ldots, \rho_k \mapsto r_k\}$ and $s^+ = s + \{r_1 \mapsto \{\}, \ldots, r_k \mapsto \{\}\}$. Also by (24)

$$R^+ \text{ connects } \varphi^+ \cup \varphi' \text{ to } s^+ \tag{30}$$

and by (25)

$$R'^+ \text{ and } R^+ \text{ agree on } \varphi^+ \cup \varphi' \tag{31}$$

where $R'^+ = R' + \{\rho_1 \mapsto r_1, \ldots, \rho_k \mapsto r_k\}$. By induction on (26), (29), (23), (30) and (31) there exist s'_1 and v' such that

$$s^+, VE \vdash R'^+(e'_1) \to v', s'_1 \tag{32}$$

$$\text{Consistent}(R^+, \mu, v, s'_1, v') \text{ w.r.t. } \varphi' \tag{33}$$

Write R'(`letregion` $\rho_1 \cdots \rho_k$ `in` e'_1 `end`) in the form `letregion` $\rho'_1 \cdots \rho'_k$ `in` e''_1 `end`. Then

$$e''_1[r_1/\rho'_1, \ldots, r_k/\rho'_k] = R'^+ e'_1$$

Thus, repeated application of rule 13 starting from (32) gives

$$s, VE \vdash R'(\texttt{letregion } \rho_1 \cdots \rho_k \texttt{ in } e'_1 \texttt{ end}) \to v', s'$$

where $s' = s'_1 \mathbin{\backslash\!\!\backslash} \{r_1, \ldots, r_k\}$. Note that R^+ and R agree on φ' (as $\{\rho_1, \ldots, \rho_k\} \cap \mathrm{frv}(\varphi') = \emptyset$). Also, $s'_1 \downarrow \mathrm{frn}(R\varphi') \sqsubseteq s'$ by (24). Then by Lemma 4.2 on (33) we get Consistent(R, μ, v, s', v') w.r.t. φ', as required.

$\boxed{\text{Application of non-recursive closure, rule 3}}$ Here $e \equiv e_1 e_2$, for some e_1 and e_2. For the base case of the inner inductive proof we have that $e' \equiv e'_1 e'_2$, for some e'_1 and e'_2 and that (21) was inferred by rule 16 on premises

$$TE \vdash e_1 \Rightarrow e'_1 : (\mu' \xrightarrow{\epsilon.\varphi_0} \mu, p), \varphi_1 \tag{34}$$

$$TE \vdash e_2 \Rightarrow e'_2 : \mu', \varphi_2 \tag{35}$$

$$\varphi = \varphi_1 \cup \varphi_2 \cup \{\epsilon, \mathbf{get}(p)\} \cup \varphi_0 \tag{36}$$

Moreover, (23) was inferred by rule 3 on premises

$$E \vdash e_1 \rightarrow v_1, \quad v_1 = \langle x_0, e_0, E_0 \rangle \tag{37}$$

$$E \vdash e_2 \rightarrow v_2 \tag{38}$$

$$E_0 + \{x_0 \mapsto v_2\} \vdash e_0 \rightarrow v \tag{39}$$

Let $\varphi'_1 = \varphi_2 \cup \{\epsilon, \mathbf{get}(p)\} \cup \varphi_0 \cup \varphi'$, i.e. the effect that remains after the computation of e'_1. Note that $\varphi \cup \varphi' = \varphi_1 \cup \varphi'_1$ so from (22), (24) and (25) we get

$$\text{Consistent}(R, TE, E, s, VE) \text{ w.r.t. } \varphi_1 \cup \varphi'_1 \tag{40}$$

$$R \text{ connects } \varphi_1 \cup \varphi'_1 \text{ to } s \tag{41}$$

$$R' \text{ and } R \text{ agree on } \varphi_1 \cup \varphi'_1 \tag{42}$$

By induction on (34), (40), (37), (41) and (42) there exist s_1 and v'_1 such that

$$s, VE \vdash R'(e'_1) \rightarrow v'_1, s_1 \tag{43}$$

$$\text{Consistent}(R, (\mu' \xrightarrow{\epsilon.\varphi_0} \mu, p), v_1, s_1, v'_1)\text{w.r.t.}\varphi'_1 \tag{44}$$

Notice that $\mathbf{get}(p) \in \varphi'_1$. Thus, by the definition of Consistent, (44) tells us that $v'_1 \in \text{Dom}(s_1)$ and r of $v'_1 = R(p)$ and there exist e'_0, VE'_0, TE_0, e''_0 and R_0 such that

$$s_1(v'_1) = \langle x_0, e'_0, VE'_0 \rangle \tag{45}$$

$$TE_0 \vdash \lambda x_0.e_0 \Rightarrow \lambda x_0.e''_0 \,\texttt{at}\, p : (\mu' \xrightarrow{\epsilon.\varphi_0} \mu, p), \{\mathbf{put}(p)\} \tag{46}$$

$$\text{Consistent}(R, TE_0, E_0, s_1, VE_0) \text{ w.r.t. } \varphi'_1 \tag{47}$$

$$R_0 \text{ and } R \text{ agree on } \varphi'_1 \tag{48}$$

$$R_0(e''_0) = e'_0 \tag{49}$$

Let $\varphi'_2 = \{\epsilon, \mathbf{get}(p)\} \cup \varphi_0 \cup \varphi'$, i.e. the effect that remains after the computation of e'_2. Note that $\varphi_2 \cup \varphi'_2 \subseteq \varphi \cup \varphi'$ and $s \sqsubseteq s_1$, so by Lemma 4.1 on (22) we have

$$\text{Consistent}(R, TE, E, s_1, VE) \text{ w.r.t. } \varphi_2 \cup \varphi'_2 \tag{50}$$

Also, from (24) and (25) we get

$$R \text{ connects } \varphi_2 \cup \varphi'_2 \text{ to } s_1 \tag{51}$$

$$R' \text{ and } R \text{ agree on } \varphi_2 \cup \varphi'_2 \tag{52}$$

By induction on (35), (50), (38), (51) and (52) there exist s_2 and v'_2 such that

$$s_1, VE \vdash R'(e'_2) \rightarrow v'_2, s_2 \tag{53}$$

$$\text{Consistent}(R, \mu', v_2, s_2, v'_2) \text{ w.r.t. } \varphi'_2 \tag{54}$$

Let $TE_0^+ = TE_0 + \{x_0 \mapsto \mu'\}$. By (46) there exists a φ'_0 such that $\varphi'_0 \subseteq \varphi_0$ and

$$TE_0^+ \vdash e_0 \Rightarrow e''_0 : \mu, \varphi'_0 \tag{55}$$

By Lemma 4.1 on (47) and $\varphi'_0 \subseteq \varphi_0$ we have

$$\text{Consistent}(R, TE_0, E_0, s_2, VE_0) \text{ w.r.t. } \varphi'_0 \cup \varphi' \tag{56}$$

and by Lemma 4.1 on (54) and $\varphi'_0 \subseteq \varphi_0$ we get

$$\text{Consistent}(R, \mu', v_2, s_2, v'_2) \text{ w.r.t. } \varphi'_0 \cup \varphi' \tag{57}$$

Let $E_0^+ = E_0 + \{x_0 \mapsto v_2\}$ and let $VE_0^+ = VE_0 + \{x_0 \mapsto v'_2\}$. Combining (56) and (57) we get

$$\text{Consistent}(R, TE_0^+, E_0^+, s_2, VE_0^+) \text{ w.r.t. } \varphi'_0 \cup \varphi' \tag{58}$$

Also, by (24) and $s \sqsubseteq s_2$ we get

$$R \text{ connects } \varphi'_0 \cup \varphi' \text{ to } s_2 \tag{59}$$

and by (48)

$$R_0 \text{ and } R \text{ agree on } \varphi'_0 \cup \varphi' \tag{60}$$

Then by induction on (55), (58), (39), (59) and (60) there exist s' and v' such that

$$s_2, VE_0^+ \vdash R_0(e''_0) \rightarrow v', s' \tag{61}$$

$$\text{Consistent}(R, \mu, v, s', v') \text{ w.r.t. } \varphi' \tag{62}$$

But by (49) we have $R_0(e''_0) = e'_0$ so (61) reads

$$s_2, VE_0 + \{x_0 \mapsto v_2\} \vdash e'_0 \rightarrow v', s' \tag{63}$$

From (43), (45), (53) and (63) we get

$$s, VE \vdash R'(e'_1\ e'_2) \rightarrow v', s' \tag{64}$$

which together with (62) is the desired result.

5 Strengths and weaknesses

In Standard ML[24], recursive functions cannot be used polymorphically within their own declaration. At first sight, our translation rules resemble the Milner/Mycroft calculus[26], in which recursive functions can be used polymorphically within their own body. The type-checking problem for the Milner/Mycroft is equivalent to the semi-unification problem[12,1], and semi-unification is unfortunately undecidable[17].

<table>
<tr><td><code>fib(15)</code></td><td>The computation of the 15th Fibonacci number by the “naïve” method (e.g. [16, page 235]). The computation of <code>fib</code>(n) requires number of function calls which is exponential in n.</td></tr>
<tr><td><code>sum(100)</code></td><td>Here <code>sum</code>(n) computes the sum $\Sigma_{i=1}^{n} i$, by n recursive calls, none of which are tail recursive.</td></tr>
<tr><td><code>sumit(100)</code></td><td>As above, but the sum is computed iteratively, by n tail recursive calls.</td></tr>
<tr><td><code>hsumit(100)</code></td><td>As above, but computed by folding the plus operator over the list $[1, .., 100]$: <code>foldr (op +) 0 [1,..,100];</code></td></tr>
<tr><td><code>acker(3,6)</code></td><td>Ackermann’s function, as defined in [16, page 239], except that our version is not curried.</td></tr>
<tr><td><code>ackerMon(3,6)</code></td><td>As above, but with Ackermann’s function made region monomorphic.</td></tr>
<tr><td><code>appel1(100)</code></td><td>A program, which Appel discusses in his chapter on space complexity [2, page 134]:
<pre>
fun s(0) = nil | s(i) = 0 :: s(i-1)

fun f (n,x) =
  let val z = length(x)
      fun g() = f(n-1, s(100))
  in
      if n=0 then 0 else g()
  end

val result = f(100,nil);
</pre></td></tr>
<tr><td><code>appel2(100)</code></td><td>Same as <code>appel1</code>, but with <code>g()</code> replaced by <code>g() + 1</code>. (Discussed by Appel[2, page 135]</td></tr>
<tr><td><code>inline(100)</code></td><td>A variant of the <code>appel1</code> obtained by in-lining <code>g</code> and making <code>f</code> region monomorphic. (Not present in Appel’s book.)</td></tr>
<tr><td><code>quick</code>(n)</td><td>Generation of list of n random numbers, followed by Quicksort (from Paulson[27, pp. 96–98].)</td></tr>
</table>

Figure 4: Test programs

In a technical report[31] we describe an inference algorithm which appears to be sound with respect to the rules in this paper and appears always to terminate. (This rather weak statement due to the fact that we do not have a proof, as yet.) We know of examples (all of which involve subtle uses of region polymorphism) for which the algorithm finds a compound type scheme which is not the most general one permitted by the translation rules. The practical implications of this situation are discussed at the end of this section.

The algorithm has been implemented. The implementation also handles pairs, lists, and conditionals, so that one can write non-trivial programs. We wrote roughly 1500 lines of test programs. After translation, the target programs were run on an instrumented interpreter, written in Standard ML. No garbage collector was implemented. The purpose of the experiments was to understand memory behaviour, not to estimate execution times.

After translation, the system performs a simple *storage mode analysis* to discovers cases, where regions can be updated destructively. This helps to get a good handling of tail recursion. One more optimization is mentioned in Appendix B. These were the only optimizations performed.

The quantities measured were:

(1) Maximal depth of region stack (unit: one region)

(2) Total number of region allocations

(3) Total number of value allocations

(4) Maximum number of storable values held (unit: 1 *sv*)

(5) Number of values stored in the final memory (unit: 1 *sv*)

The test programs in Figure 4 are representative of best and worst behaviour. The results are shown in the tables below. The numbers in the first column always refer to the the quantities enumerated above.

	`fib(15)`	`sum(100)`	`sum(`n`)`
(1)	47	205	$2n+5$
(2)	15,030	606	$6n+6$
(3)	15,030	606	$6n+6$
(4)	32	104	$n+4$
(5)	1	1	1

Notice that `fib` and `sum` used very little memory, at the expense of very frequent region allocation and deallocation. From lines (2) and (3) we see that the region analysis can be so fine-grained that there is a one-to-one correspondence between values and regions. In the above examples, this is due to region polymorphism. The third column gives the exact figures, as a function of n. These are obtained by inspection of the target program, which appears in Appendix B.

	`sumit(100)`	`hsumit(100)`
(1)	6	12
(2)	406	715
(3)	707	1,214
(4)	6	507
(5)	1	101

The results for `sumit` illustrate the behaviour on tight, tail-recursive loops. When computing `sumit(`n`)`, the number of the highest region, namely 6, and the maximum memory size, also 6, are both independent of n.

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

	`acker(3,6)`	`ackerMon(3,6)`
(1)	3.058	514
(2)	1,378,366	1,119,767
(3)	1,378,367	1,550,599
(4)	2,043	86,880
(5)	1	1

The strength of region polymorphism is illustrated by the differences observed between `acker` and `ackerMon`. The latter, where region polymorphism has been disabled, has a much larger maximal memory size, 86,880, than the former, 2,043.

	`quick(50)`	`quick(500)`
(1)	170	1,520
(2)	2,729	45,691
(3)	3,684	65,266
(4)	603	8,078
(5)	152	1,502

	`quick(1000)`	`quick(5000)`
(1)	3,020	15,020
(2)	86,915	556,369
(3)	122,793	795,376
(4)	10,525	61,909
(5)	3,002	15,002

A list occupies three regions: one for the elements, one for the constructors (cons and nil) and one for the pairs, to which cons is applied. Thus, a list with n different integers is represented by $3n+1$ values.

We see that, apart from one word, the final results are the only values left at the end of the computation.[2] Also, the maximal number of regions allocated (line 1) is roughly the same as the number of values in the final result. The ratio between the maximal memory size and the final memory size varies between roughly 4.0 and 5.5.

	`appel1(100)`	`appel2(100)`	`inline(100)`
(1)	911	1,111	311
(2)	81,714	81,914	81,113
(3)	101,614	101,814	101,413
(4)	20,709	20,709	411
(5)	1	1	1

The programs `appel1` and `appel2` use $\Theta(N^2)$ space (line 4), although $\Theta(N)$ ought to be enough. This is an example of a deficiency which our scheme has in common with the classical stack discipline: creating a large argument for a function which only uses it for a small part of its activation leads to waste of memory (see also Chase [5]). `inline(100)` uses only $\Theta(N)$ space, as the storage mode analysis discovers that updates are possible.

In cases where the algorithm infers a compound type scheme which is not the most general one permitted by the translation rules, the implementation detects the situation, prints a warning and completes the translation using a less general type scheme. Amongst our total supply of test programs, only those specifically designed to provoke the warning provoked it.

Garbage collection

If one wants to combine region allocation with garbage collection, dangling pointers are a worry. They can be avoided by adding the extra side-condition $\forall y \in \mathrm{FV}(\lambda x.e).\,\mathrm{frv}(TE(y)) \subseteq \mathrm{frv}(\varphi')$ to rule 15. This affects the precision of the inference rules, but obviously not their soundness.

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

6 Conclusion

The experiments presented in this report suggest that the scheme in many cases leads to very economical use of memory resources, even without the use of garbage collection. They also reveal that region allocation and de-allocation are very frequent and that many regions are small and short-lived. Regions with statically known, finite size can be allocated on a hardware stack; small regions can in principle even be held in machine registers. Magnus Vejlstrup is working on inferring the sizes of regions. Lars Birkedal is writing a compiler from region-annotated programs to C, together with a runtime system in C. In this system, a variable-sized region is represented by a linked list of fixed-size pages, which are taken from and returned to a free-list.

Acknowledgments

Fritz Henglein's expertise on semi-unification was most helpful. Peter Sestoft contributed in essential ways to the storage mode analysis mentioned in Section 5. Lars Birkedal wrote parts of the current implementation. Andrew Appel, David MacQueen, Flemming Nielson, Hanne Riis Nielson, David Schmidt and David N. Turner contributed with discussions and valuable criticism.

This work is supported by Danish Research Council grant number 5.21.08.03 under the DART project.

References

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

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

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

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

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

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

[7] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proc. 9th Annual ACM Symp. on Principles of Programming Languages*, pages 207–212, Jan. 1982.

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

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

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

[11] Barry Hayes. Using key object opportunism to collect old objects. In *Proceedings: Conference on Object-Oriented Programming Systems, Languages and Applications, Sigplan Notices, Vol 26, Number 11*, 1991.

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

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

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

[15] Hiroyuki Seki Katsuro Inoue and Hikaru Yagi. Analysis of functional programs to detect runtime garbage cells. *ACM Transactions on Programming Languages and Systems*, 10(4):555–578, 1988.

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

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

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

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

[20] J. M. Lucassen. *Types and Effects, towards the integration of functional and imperative programming.* PhD thesis, MIT Laboratory for Computer Science, 1987. MIT/LCS/TR-408.

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

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

[23] Robin Milner and Mads Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87:209–220, 1991.

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

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

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

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

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

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

[30] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3), 1992.

[31] Mads Tofte and Jean-Pierre Talpin. *A Theory of Stack Allocation in Polymorphically Typed Languages.* Technical Report DIKU-report 93/15, Department of Computer Science, University of Copenhagen, 1993.

Appendix A. Arrow effects

The purpose of this appendix is to motivate the use of arrow effects of the special form $\epsilon.\varphi$. The "ϵ." has to do with type checking. Effects are sets. Having sets on the function arrow[14,30] forces one to re-think the use of unification as the basic mechanism for type checking.

Milner's algorithm W[22] works on the following principle. Let e be an expression and let TE be a type environment giving the types of the free variables of e. Then $W(TE, e)$ attempts to find not just a type τ for e, but also a substitution S, such that $S(TE) \vdash e : \tau$. Informally speaking, the substitution says how the types in the type environment must be "refined" in order to make e typable.

In effect systems an important form of "type refinement" is that of increasing an effect (under the ordering of set inclusion). For example, consider the type checking of the expression

$$\lambda h.\ \mathtt{if}\ e\ \mathtt{then}\ h\ \mathtt{else}\ (\lambda x.x+1)$$

where we assume that e is an expression which contains an application of h. Assume for the moment that arrow effects are just effects. After the `then` branch has been analysed, the type environment might contain the binding: $\{h \mapsto ((\alpha,\rho_1) \xrightarrow{\emptyset} (\beta,\rho_2),\rho_3)\}$. Next, the type of $(\lambda x.x+1)$ might be inferred to be $((\mathtt{int},\rho_1') \xrightarrow{\{\mathbf{get}(\rho_1'),\mathbf{put}(\rho_2')\}} (\mathtt{int},\rho_2'),\rho_3')$. We want the unification of these types to refine the type of h to have the effect $\{\mathbf{get}(\rho_1'),\mathbf{put}(\rho_2')\}$ on the function arrow.

Talpin and Jouvelot[30] introduce *effect variables* to achieve this. In their algorithm, one always has just an effect variable on every function arrow. In addition, their algorithm infers a set of constraints of the form $\epsilon \supseteq \varphi$. Their algorithm then alternates between solving constraint sets and inferring types. Our arrow effects give a variation of their scheme which allows substitution to do all "refinement" without constraint sets. In the present system, under the assumption $\{h \mapsto ((\alpha,\rho_1) \xrightarrow{\epsilon_1.\emptyset} (\beta,\rho_2),\rho_3)\}$ the type of the `then` branch above is $((\alpha,\rho_1) \xrightarrow{\epsilon_1.\emptyset} (\beta,\rho_2),\rho_3)$ and the type of the `else` branch is

$$((\mathtt{int},\rho_1') \xrightarrow{\epsilon_2.\{\mathbf{get}(\rho_1'),\mathbf{put}(\rho_2')\}} (\mathtt{int},\rho_2'),\rho_3')$$

Unification then gives a substitution on region and type variables, but it also produces the *effect substitution*

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

Thus the resulting type of h in the type environment is

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

Appendix B. A larger example

The source program sum mentioned in Section 5 is

```
letrec $sum$ = $\lambda$ $x$ .
  if $x$=0 then 1 else $x$+$sum$($x$-1)
in $sum$(100)
end
```

It translates into the target program:

```
letregion $\rho_1$
in
  letrec $sum$[$\rho_2$,$\rho_3$] at $\rho_1$ =
    ($\lambda$ $x$:(int,$\rho_2$).
      if letregion $\rho_4$
           in letregion $\rho_5$
                in ($x$=(0 at $\rho_5$)) at $\rho_4$
                end
           end
      then 1 at $\rho_3$
      else
        letregion $\rho_6$
        in
          ($x$+
            letregion $\rho_7$,$\rho_8$
            in
              $sum$[$\rho_8$,$\rho_6$] at $\rho_7$
              letregion $\rho_9$
              in ($x$-(1 at $\rho_9$)) at $\rho_8$
              end
            end
          ) at $\rho_3$
        end
    ) at $\rho_1$
  in letregion $\rho_{10}$,$\rho_{11}$
       in $sum$[$\rho_{11}$,$\rho_0$] at $\rho_{10}$
            (100 at $\rho_{11}$)
       end
  end
end
```

Note that *sum* becomes region-polymorphic and that this region polymorphism is exploited inside the body of *sum* itself: regions ρ_6 and ρ_8 are local to the function body and are passed to the recursive call (ρ_8 holds the argument and ρ_6 is where the result is to be placed). Note that the regions are allocated and de-allocated much as they would be in the classical stack discipline. In fact, we can count how many regions will be allocated, as a function of n. For $n = 0$, the maximal stack depth is 6 (ρ_0, ρ_1, ρ_{10}, ρ_{11}, ρ_4 and ρ_5). For $n > 0$, the maximal stack depth is $6 + 3n$ (the factor 3 stems from regions ρ_6, ρ_7 and ρ_8; ρ_9 disappears before the recursive call). The optimization which we hinted at in Section 5 discovers that ρ_7 and ρ_{10} can be de-allocated during the calls they serve, bringing the maximal stack depth down to $2n + 5$.