Electronic Notes in Theoretical Computer Science 41 No. 3 (2000)
URL: http://www.elsevier.nl/locate/entcs/volume41.html pages 1–19

Syntactic Type Soundness for the Region Calculus

Simon Helsen and Peter Thiemann

Institut für Informatik, Universität Freiburg
Georges-Köhler-Allee 79
D-79110 Freiburg, Germany
Email: {helsen,thiemann}@informatik.uni-freiburg.de

Abstract

The region calculus of Tofte and Talpin is an annotated polymorphically typed lambda calculus which makes memory allocation and deallocation explicit. It is intended as an intermediate language in a compiler for ML-like languages. The region annotations are obtained by static region and effect inference, which makes it an attractive alternative for garbage collection. Soundness of the region and effect system is crucial to guarantee safe deallocation of regions, i.e. deallocation should only take place for objects which are provable dead.

Tofte and Talpin have proved type soundness of the region calculus using rule-based co-induction. This proof is quite complicated and not very intuitive. Much of the problem lies in the low-level big-step operational semantics which involves manipulations of an explicit store and which has a co-inductive definition. In this paper, we present a small-step operational semantics for the region calculus, based on syntactic rewriting. We prove type soundness following the approach of Wright and Felleisen, leading to very simple inductive proofs.

1 Introduction

Memory management for dynamic data structures is a problem in programming. While memory allocation is dictated by the problem at hand, there is considerable freedom in memory deallocation. If deallocation happens too late, the program suffers from memory bloat and space leaks, which impede performance. If deallocation happens too early, there might be dangling pointers into deallocated memory. Dereferencing a dangling pointer is unsafe and might lead to a crash, or worse, to wrong results.

Some languages (like C or Pascal) leave the deallocation problem entirely to the programmer, whereas others (like Smalltalk, Java, ML, and Haskell) perform automatic deallocation by incorporating garbage collection into the

runtime system. While the programmer-based solution is immensely error-prone, programs can in principle be tuned for optimal memory use. Garbage collection avoids a large class of errors, but it has some problems, too. Since the garbage collector is, in general, unaware of the semantics of the running program, it must preserve all pointers reachable from a given set of root pointers. This set is a conservative approximation of the set of pointers that will actually be used by the program. In consequence, deallocation might happen too late, which can lead to space leaks. In addition, garbage collection takes extra, non-productive time and can cause erratic pauses in the execution of programs. Finally, interoperability between garbage collected languages, like ML, and non-garbage collected languages, such as C, is difficult.

The region calculus of Tofte and Talpin [16, 15] provides an alternative method of memory management for the functional language ML. It is intended and used as an intermediate language in an ML compiler [2, 3, 14, 15, 16]. The basic idea is to split memory into regions that are allocated in a stack-like manner, directed by a construct of the language. Deallocation is instantaneous, it just pops the topmost region from the stack. Using this method, it is possible to implement ML without garbage collection (in principle), while guaranteeing safety. In some instances, the region calculus can even prove that a pointer is semantically dead, so that the region it points to can be safely deallocated. Standard garbage collectors cannot do this.

1.1 Related Work and Contribution

The proof of consistency, or type soundness, for the region calculus as it is given by Tofte and Talpin [16] is a complicated proof using rule-based co-induction. The source of the complication is the co-inductive definition of consistency caused by the explicit use of a store in their *big-step* semantics. Recently, alternative type-soundness proofs for the region calculus have been proposed.

(i) Crary, Walker, and Morrisett [6] provide an indirect soundness proof by translating the region calculus into their capability calculus. For the latter calculus, they provide a syntactic soundness proof.

(ii) Banerjee, Heintze, and Riecke [1] translate the region calculus into an extension of the polymorphic lambda calculus called $F_{\#}$. For the latter, they construct a semantic soundness proof, exploiting the properties of their original denotational model.

(iii) Dal Zilio and Gordon [18] modify the operational semantics of Tofte and Talpin so that it also keeps track of deallocated regions. Albeit artificial, this extra information allows an inductive definition of the consistency relation and an inductive correctness proof. Then they go on to show that this result is a consequence of a more general result for a typed π-calculus with name groups. This is shown using a translation from the region caculus to the typed π-calculus with name groups.

However, the question for a direct syntactic soundness proof for the region calculus is still open. The present paper provides such a soundness proof using the technique of Wright and Felleisen [17]. This approach requires a *small-step* operational semantics which is based on syntactic rewriting. The main challenge in the present work is the definition of a language of computational terms that captures the intuition of Tofte and Talpin's big-step semantics and is suitable to define a small-step operational semantics for the region calculus. Most striking, our rewrite semantics only uses region annotations on values and avoids the use of an explicit store.

The proofs themselves use routine inductive techniques and are therefore considerably easier than the co-inductive proofs of Tofte and Talpin.

In contrast to the other soundness proofs [6, 1, 18], we treat the complete *polymorphic* region calculus. As noted by Tofte and Talpin [16], type polymorphism does not add conceptual problems to the type soundness result, but polymorphic recursion gives rise to some subtle twists. We have included some illustrative cases in the paper.

After submission of this paper, we learned of another soundness proof by Calcagno [4]. He defines a high-level structural operational semantics and proves type soundness for it. The similarities between his big-step semantics and our small-step semantics are remarkable: the main difference is that our semantics is entirely based on syntactic rewriting, whereas his operational semantics propagates a set of live regions. Calcagno formally relates the high-level semantics to the original low-level semantics of Tofte and Talpin.

1.2 Overview

The rest of the paper is structured as follows: in Section 2 we introduce the region calculus. Section 3 presents the small-step operational semantics. Then, in Section 4, we recall the static semantics of Tofte and Talpin and provide the necessary extensions for our presentation. In Section 5, we prove syntactic type soundness. A small example is given in Section 6 and finally we conclude.

2 The region calculus

The region calculus, λ^{region}, is an explicitly typed polymorphic lambda calculus, which makes memory allocation and deallocation explicit. Figure 1 defines its syntactic categories. *Surface terms* are built up from variables, integer constants, lambda abstractions, applications, and recursive function definitions, as usual in applied lambda calculi. A `copy` operation serves as a prototypical primitive operation. In addition, there are terms particular to the region calculus, region introduction and application of a region abstraction.

Memory is divided in *regions* of unbounded size, which are allocated and deallocated in a stack-like manner. The term, `letregion` ϱ `in` e, allocates

$$
\begin{array}{lll}
\text{Surface Terms} & e ::= & x \mid c\ \texttt{at}\ \varrho \mid \lambda x.\, e\ \texttt{at}\ \varrho \mid e\ @\ e \mid \texttt{copy}\,[\varrho_1, \varrho_2]\, e \mid \\
& & \texttt{letrec}\ f = a\ \texttt{at}\ \varrho\ \texttt{in}\ e \mid \texttt{letregion}\ \varrho\ \texttt{in}\ e \mid \\
& & f\ [\varrho_1, \ldots, \varrho_n]\ \texttt{at}\ \varrho \\
\text{Region Abstractions} & a ::= & \Lambda[\varrho_1, \ldots, \varrho_n].\lambda x.\, e
\end{array}
$$

$$
\begin{array}{lll}
x & \in & \mathit{Vname} \quad \text{some infinite set of variable names} \\
f & \in & \mathit{Fname} \quad \text{some infinite set of function names} \\
& & \text{with } \mathit{Vname} \cap \mathit{Fname} = \emptyset
\end{array}
$$

Fig. 1. Syntax of λ^{region}

a new region, binds it to ϱ, evaluates e, and finally deallocates the region. Hence, the lifetime of a region corresponds with the lexical scope of ϱ. All memory-allocating constructs carry a region variable. It indicates the region in which the allocation must be performed. Constants have a boxed representation, hence c `at` ϱ allocates memory in ϱ and stores the integer c. A lambda abstraction $\lambda x.\, e$ `at` ϱ builds a closure in region ϱ. The term `letrec` $f = a$ `at` ϱ `in` e recursively binds the variable f to the closure of a *region abstraction* a, which is put into region ϱ. A *region abstraction* is a lambda term, which is abstracted over zero or more region variables. The term $f\ [\varrho_1, \ldots, \varrho_n]$ `at` ϱ is a *region function application*. It carries a region annotation because it copies the closure for the region abstraction $\Lambda[\varrho_1, \ldots, \varrho_n].\lambda x.\, e$ bound to f to an ordinary closure in region ϱ.

The term $\texttt{copy}\,[\varrho_1, \varrho_2]\, e$ denotes a copy operation which moves the (integer) value of e from region ϱ_1 to region ϱ_2.

The set, $\mathit{free}(e) \subseteq \mathit{Vname} \cup \mathit{Fname}$, contains all term variables occurring free in term e, with the usual definition. The set, $\mathit{frv}(e)$, contains the free region variables of e. There are two constructs that bind region variables: Region abstraction $\Lambda[\varrho_1, \ldots, \varrho_n].\lambda x.\, e$ binds region variables $\varrho_1, \ldots, \varrho_n$ and `letregion` ϱ `in` e binds ϱ. All other occurrences of region variables are free. The term e is *closed* if it contains no free term variables ($\mathit{free}(e) = \emptyset$). A closed term may contain free region variables.

3 Dynamic semantics of λ^{region}

Differing from Tofte and Talpin, we define a small-step operational semantics for λ^{region} by a set of rewrite rules on *computational terms*. Computational terms extend λ^{region}-terms by constructs that make intermediate computational states explicit. Figure 2 defines the extended syntax and the small-step

$$
\begin{array}{lll}
\text{Computational Terms} & e ::= & x \mid v \mid c \;\texttt{at}\; \rho \mid \lambda\, x.\, e \;\texttt{at}\; \rho \mid e \mathbin{@} e \mid \texttt{copy}\,[\rho_1, \rho_2]\, e \mid \\
& & \texttt{letrec}\; f = \gamma\; \texttt{in}\; e \mid \texttt{letregion}\; \varrho\; \texttt{in}\; e \mid \\
& & \gamma'\; [\rho_1, \ldots, \rho_n]\; \texttt{at}\; \rho \\
\text{Values} & v ::= & \langle c \rangle_\rho \mid \langle \lambda\, x.\, e \rangle_\rho \\
\text{Regions} & \rho ::= & \varrho \mid \bullet \\
\text{Region Abstractions} & a ::= & \Lambda[\varrho_1, \ldots, \varrho_n].\lambda\, x.\, e \\
& \gamma ::= & a \;\texttt{at}\; \rho \mid \langle a \rangle_\rho \\
& \gamma' ::= & f \mid \langle a \rangle_\rho \\
\\
\text{Evaluation Contexts} & \mathcal{E} ::= & [\,] \mid \mathcal{E} \mathbin{@} e \mid v \mathbin{@} \mathcal{E} \mid \texttt{copy}\,[\rho_1, \rho_2]\, \mathcal{E} \mid \\
& & \texttt{letregion}\; \varrho\; \texttt{in}\; \mathcal{E}
\end{array}
$$

$$
c \;\texttt{at}\; \varrho \to_r \langle c \rangle_\varrho \tag{1}
$$

$$
\lambda\, x.\, e \;\texttt{at}\; \varrho \to_r \langle \lambda\, x.\, e \rangle_\varrho \tag{2}
$$

$$
\texttt{letrec}\; f = a \;\texttt{at}\; \varrho\; \texttt{in}\; e \to_r \texttt{letrec}\; f = \langle a \rangle_\varrho\; \texttt{in}\; e \tag{3}
$$

$$
\texttt{letregion}\; \varrho\; \texttt{in}\; v \to_r v[\bullet/\varrho] \tag{4}
$$

$$
\texttt{copy}\,[\varrho_1, \varrho_2]\, \langle c \rangle_{\varrho_1} \to_r \langle c \rangle_{\varrho_2} \tag{5}
$$

$$
\langle \lambda\, x.\, e \rangle_\varrho \mathbin{@} v \to_r e[v/x] \tag{6}
$$

$$
\begin{array}{c}
\texttt{letrec}\; f = \gamma\; \texttt{in}\; e_2 \to_r e_2[\langle \Lambda[\varrho_1, \ldots, \varrho_n].\lambda\, x.\, \texttt{letrec}\; f = \gamma\; \texttt{in}\; e_1 \rangle_\rho / f] \\
\text{where} \quad \gamma = \langle \Lambda[\varrho_1, \ldots, \varrho_n].\lambda\, x.\, e_1 \rangle_\rho
\end{array} \tag{7}
$$

$$
\langle \Lambda[\varrho_1, \ldots, \varrho_n].\lambda\, x.\, e \rangle_\varrho\; [\rho'_1, \ldots, \rho'_n]\; \texttt{at}\; \varrho' \to_r \langle \lambda\, x.\, e[\rho'_1/\varrho_1, \ldots, \rho'_n/\varrho_n] \rangle_{\varrho'} \tag{8}
$$

$$
\frac{e \to_r e'}{\mathcal{E}[e] \to_r \mathcal{E}[e']} \tag{9}
$$

Fig. 2. Small-step Operational Semantics

transition relation $\to_r$ using the technique of evaluation contexts [9]. The relation $\twoheadrightarrow_r$ is its reflexive and transitive closure.

None of the original terms qualifies as a syntactic value because each of them performs a non-trivial computation step, even if it only allocates memory. Hence, we introduce three new value terms. The value $\langle c \rangle_\rho$ is a pointer

to an integer c in region ρ. The value $\langle \lambda\, x.\ e \rangle_\rho$ is a pointer to a closure in region ρ. The metavariable ρ ranges over region variables ϱ and a distinct *dead region* $\bullet$, which is the union of all deallocated regions. For example, $\langle c \rangle_\bullet$ is a dangling pointer to an integer in a deallocated region.

The third value term, $\langle a \rangle_\rho$, is a pointer to the closure of a region abstraction. It only occurs in conjunction with `letrec` and region function application. Evaluation allocates the closure when it enters a `letrec` term for the first time. Unfolding the recursion presumes that the closure is already allocated. Consequently, a region function application either starts with some function variable f or with a pointer to such a closure, after unfolding the recursion.

The rules (1), (2), and (3) deal with memory allocation of constants, lambda abstractions, and region abstractions, respectively. The rules (5), (6), and (7) are computation rules that define the `copy` operation, beta-value reduction, and unfolding of `letrec`. The notation, $e[e'/x]$, stands for the term e with e' substituted for each free occurrence of x (and analogously for $e[\langle a \rangle_\rho / f]$). Substitution avoids capture of term and region variables by renaming. Rule (4) deallocates a region of memory by substituting $\bullet$ for the `letregion`-bound region variable, once the body has turned into a syntactic value. The substitution $v[\bullet/\varrho]$ replaces all free occurrences of ϱ in v with $\bullet$. Rule (8) defines region function application, which is just beta reduction for region abstractions. Finally, Rule (9) is a context rule, which specifies a call-by-value semantics through the set $\mathcal{E}$ of evaluation contexts. All rules require that the regions involved in the reduction step are not dead, as indicated by the use of ϱ.

4 Static semantics of λ^{region}

This section first summarizes the semantic objects of the static semantics, as defined by Tofte and Talpin [16]. Then, it discusses the type rules for surface terms and the extensions for computational terms.

4.1 Semantic objects

An *effect* φ is a finite set of regions, ρ, and *effect variables*, ϵ. The effect of a term, e, contains the set of regions that may be affected by evaluation of e. A *type*, τ, and a *type with place*, μ, are defined by

$$\tau ::= \alpha \mid \texttt{int} \mid \mu \xrightarrow{\epsilon.\varphi} \mu \qquad\qquad \mu ::= (\tau, \rho)$$

A type is either a type variable, α, an integer type, or a function type. Function types carry an *arrow effect* $\epsilon.\varphi$. An arrow effect is a pair of an effect variable and an effect. Arrow effects are a technical device for type reconstruction of λ^{region} [13]. The effect, φ, is the latent effect that happens on application of

the function. The effect variable identifies a group of functions that share an application. The use of effect variables corresponds to a simple flow analysis.

A type with place, μ, is a pair of a type, τ, and a region, ρ. The region specifies where an object of type with place μ is stored. If the region is dead then the object cannot be accessed anymore.

A substitution, $S_s = (S_t, S_r, S_e)$, is a triple of

- a type substitution, S_t, which maps type variables to types,
- a region substitution, S_r, which maps region variables to regions, and
- an effect substitution, S_e, which maps effect variables to arrow effects.

Application of S_s to effects, types, and types with places is defined as follows:

$$S_s(\varphi) = \{S_s(\rho) \mid \rho \in \varphi\} \cup \{\eta \mid \epsilon \in \varphi \wedge S_e(\epsilon) = \epsilon'.\varphi' \wedge \eta \in \{\epsilon'\} \cup \varphi'\}$$

$$S_s(\texttt{int}) = \texttt{int} \quad S_s(\alpha) = S_t(\alpha) \quad S_s(\varrho) = S_r(\varrho) \quad S_s(\tau, \rho) = (S_s(\tau), S_s(\rho))$$

$$S_s(\bullet) = \bullet \quad S_s(\mu \xrightarrow{\epsilon.\varphi} \mu') = S_s(\mu) \xrightarrow{\epsilon'.(\varphi' \cup S_s(\varphi))} S_s(\mu') \quad \textit{where } S_e(\epsilon) = \epsilon'.\varphi'$$

The domain of S_s is the set of variables, for which $S_s(\alpha) \neq \alpha$, $S_s(\varrho) \neq \varrho$, and $S_s(\epsilon) \neq \epsilon.\{\}$. The substitutions I_t and I_r are the identities on type and region variables, respectively. The substitution I_e maps every effect variable ϵ to $\epsilon.\{\}$. Sometimes, S_r stands for (I_t, S_r, I_e) and $e[\rho_1/\varrho_1, \ldots, \rho_n/\varrho_n]$ for $S_r(e)$ where $S_r = \{\varrho_1 \mapsto \rho_1, \ldots, \varrho_n \mapsto \rho_n\}$.

Type schemes, $\sigma ::= \forall\, \boldsymbol{\alpha}\, .\forall\, \boldsymbol{\epsilon}\, .\forall\, \boldsymbol{\varrho}\, .\tau$, extend those of Damas and Milner [8] by binding type, effect, and region variables, with $\boldsymbol{\alpha}$ a sequence of distinct type variables $\alpha_1, \ldots, \alpha_n$ and similarly for regions and effect variables.

A type τ is an *instance* of a type scheme $\sigma = \forall\, \boldsymbol{\alpha}\, .\forall\, \boldsymbol{\epsilon}\, .\forall\, \boldsymbol{\varrho}\, .\tau'$ via substitution S_s, written $\tau \prec \sigma$ *via* S_s, if the domain of $S_s \subseteq \{\boldsymbol{\alpha}, \boldsymbol{\epsilon}, \boldsymbol{\varrho}\}$ and $S_s(\tau') = \tau$. The instance relation extends to type schemes by $\sigma \prec \sigma'$ iff, for all types τ, $\tau \prec \sigma$ *via* S_s implies $\tau \prec \sigma'$ *via* S'_s.

A type environment, *TE*, is a finite map that maps lambda-bound variables to pairs of the form (τ, ρ) and `letrec`-bound variables to pairs of the form (σ, ρ). The updated type environment, $TE + \{x \mapsto \mu\}$, maps x to μ and otherwise behaves like *TE*.

Substitutions extend to type schemes, type environments, and expressions in the obvious way, avoiding capture by renaming.

Free variables of the above semantic objects are defined in the usual way. For a semantic object O, the free type, region, and effect variables of O are $\mathit{ftv}(O)$, $\mathit{frv}(O)$, and $\mathit{fev}(O)$, respectively. Furthermore, $\mathit{fv}(O) = \mathit{ftv}(O) \cup \mathit{frv}(O) \cup \mathit{fev}(O)$.

(var)
$$\frac{TE(x) = (\tau, \rho)}{TE \vdash_{tt} x : (\tau, \rho), \emptyset}$$

(alloc-const)
$$\frac{}{TE \vdash_{tt} c \texttt{ at } \rho : (\texttt{int}, \rho), \{\rho\}}$$

(alloc-abstr)
$$\frac{TE + \{x \mapsto \mu_1\} \vdash_{tt} e : \mu_2, \varphi \quad \varphi \subseteq \varphi'}{TE \vdash_{tt} \lambda x.\, e \texttt{ at } \rho : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \rho), \{\rho\}}$$

(alloc-letrec)
$$\frac{\begin{array}{c}\{\varrho_1, \ldots, \varrho_n, \boldsymbol{\epsilon}\} \cap (fv(TE) \cup \{\rho\}) = \emptyset \quad \hat{\sigma} = \forall\, \boldsymbol{\epsilon}.\forall\, \varrho_1, \ldots, \varrho_n.\tau \\ TE + \{f \mapsto (\hat{\sigma}, \rho)\} \vdash_{tt} \lambda x.\, e_1 \texttt{ at } \rho : (\tau, \rho), \{\rho\} \quad \sigma = \forall\, \boldsymbol{\alpha}.\hat{\sigma} \\ \{\boldsymbol{\alpha}\} \cap ftv(TE) = \emptyset \quad TE + \{f \mapsto (\sigma, \rho)\} \vdash_{tt} e_2 : \mu, \varphi\end{array}}{TE \vdash_{tt} \texttt{letrec } f = \Lambda[\varrho_1, \ldots, \varrho_n].\lambda x.\, e_1 \texttt{ at } \rho \texttt{ in } e_2 : \mu, \varphi \cup \{\rho\}}$$

(app)
$$\frac{TE \vdash_{tt} e_1 : (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho), \varphi_1 \quad TE \vdash_{tt} e_2 : \mu_1, \varphi_2}{TE \vdash_{tt} e_1 \mathbin{@} e_2 : \mu_2, \varphi \cup \varphi_1 \cup \varphi_2 \cup \{\epsilon, \rho\}}$$

(recvar1)
$$\frac{\begin{array}{c}TE(f) = (\sigma, \rho) \quad \sigma = \forall\, \boldsymbol{\alpha}.\forall\, \boldsymbol{\epsilon}.\forall\, \varrho_1, \ldots, \varrho_n.\tau \\ \tau' \prec \sigma \;\; via \;\; (S_t, S_r, S_e) \quad S_r = \{\varrho_1 \mapsto \rho'_1, \ldots, \varrho_n \mapsto \rho'_n\}\end{array}}{TE \vdash_{tt} f\ [\rho'_1, \ldots, \rho'_n] \texttt{ at } \rho' : (\tau', \rho'), \{\rho', \rho\}}$$

(letregion)
$$\frac{TE \vdash_{tt} e : \mu, \varphi \quad \varrho \notin frv(TE, \mu)}{TE \vdash_{tt} \texttt{letregion } \varrho \texttt{ in } e : \mu, \varphi \setminus \{\varrho\}}$$

(effect)
$$\frac{TE \vdash_{tt} e : \mu, \varphi \quad \epsilon \notin fev(TE, \mu)}{TE \vdash_{tt} e : \mu, \varphi \setminus \{\epsilon\}}$$

(copy)
$$\frac{TE \vdash_{tt} e : (\texttt{int}, \rho), \varphi}{TE \vdash_{tt} \texttt{copy}\,[\rho, \rho']\, e : (\texttt{int}, \rho'), \varphi \cup \{\rho, \rho'\}}$$

Fig. 3. Static semantics - Part 1

(use-const)
$$\frac{}{TE \vdash_{tt} \langle c \rangle_\rho : (\mathtt{int}, \rho), \emptyset}$$

(use-abstr)
$$\frac{TE + \{x \mapsto \mu_1\} \vdash_{tt} e : \mu_2, \varphi \quad \varphi \subseteq \varphi'}{TE \vdash_{tt} \langle \lambda\, x.\, e \rangle_\rho : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \rho), \emptyset}$$

(use-letrec)
$$\frac{\begin{array}{c}\{\varrho_1, \ldots, \varrho_n, \boldsymbol{\epsilon}\} \cap (fv(TE) \cup \{\rho\}) = \emptyset \quad \hat{\sigma} = \forall\, \boldsymbol{\epsilon}\, . \forall\, \varrho_1, \ldots, \varrho_n\, . \tau \\ TE + \{f \mapsto (\hat{\sigma}, \rho)\} \vdash_{tt} \langle \lambda\, x.\, e_1 \rangle_\rho : (\tau, \rho), \emptyset \quad \sigma = \forall\, \boldsymbol{\alpha}\, . \hat{\sigma} \\ \{\boldsymbol{\alpha}\} \cap ftv(TE) = \emptyset \quad TE + \{f \mapsto (\sigma, \rho)\} \vdash_{tt} e_2 : \mu, \varphi\end{array}}{TE \vdash_{tt} \mathtt{letrec}\ f = \langle \Lambda[\varrho_1, \ldots, \varrho_n].\lambda\, x.\, e_1 \rangle_\rho\ \mathtt{in}\ e_2 : \mu, \varphi}$$

(recvar2)
$$\frac{\begin{array}{c}TE \vdash_{tt} \langle \lambda\, x.\, e \rangle_{\rho'} : (\tau, \rho'), \emptyset \quad \{\varrho_1, \ldots, \varrho_n\} \cap (frv(TE) \cup \{\rho\}) = \emptyset \\ \tau' = S_r(\tau) \quad S_r = \{\varrho_1 \mapsto \rho'_1, \ldots, \varrho_n \mapsto \rho'_n\}\end{array}}{TE \vdash_{tt} \langle \Lambda[\varrho_1, \ldots, \varrho_n].\lambda\, x.\, e \rangle_\rho\ [\rho'_1, \ldots, \rho'_n]\ \mathtt{at}\ \rho' : (\tau', \rho'), \{\rho', \rho\}}$$

Fig. 4. Static semantics - Part 2

4.2 Typing rules of λ^{region}

The typing judgment of λ^{region} has the form $TE \vdash_{tt} e : \mu, \varphi$. It reads "in type environment TE, expression e has type μ and effect φ." Figure 3 shows the typing rules for the surface terms. Except for the *(copy)*-rule and the generalization of region variables ϱ to regions ρ, the rules of Fig. 3 are identical to those of Tofte and Talpin [16]. Figure 4 shows the rules for the remaining computational terms.

Rule *(var)* is obvious. Rule *(alloc-const)* types the allocation of an integer. It specifies an effect on the respective region. Contrast this with rule *(use-const)* (from Fig. 4), which types a pointer to an integer. It does not have an effect anymore. Likewise, rule *(alloc-abstr)*, for allocating a closure, has an effect whereas the corresponding pointer reference in rule *(use-abstr)* has not. Both rules transfer the effect of the body term to the latent effect of the inferred function type. Analogously, rule *(alloc-letrec)* types the allocation of a recursive region closure whereas rule *(use-letrec)* only types the pointer to the closure. Both rules specify polymorphic recursion for effect and region variables, but not for types. In the body, e_2, the function, f, is type polymorphic, too.

Rule *(app)* collects the effects of the subexpressions, the latent effect of the function, and the effect variable. Rule *(recvar1)* types a region application before substituting a region closure for f whereas *(recvar2)* applies after the

substitution. As the latter expression is a redex which involves both regions, ρ and ρ', the effect contains both regions.

Rule *(letregion)* discharges a region variable if it does not occur in the type environment and in the expression's type. Hence, effects on deallocated regions are masked. Rule *(effect)* discharges useless effect variables. Rule *(copy)* declares the effect $\{\rho, \rho'\}$ of copying an integer from region ρ to ρ'.

5 Type soundness of λ^{region}

This section provides a syntactic type soundness proof of the small-step transition relation in λ^{region} with respect to the type system of section 4. The proof is structured as follows: first we formulate some useful lemmas. Then, we prove *type preservation*, also known as *subject reduction* [17, 7], which states that a well-typed computational term remains well-typed under the small-step transition relation $\rightarrow_r$. The second result is the *progress* property, which states that a well-typed closed term is either a value or it can be further reduced. Taken together, these two results imply type soundness.

5.1 Auxiliary Lemmas

The first lemma states that syntactic values have no effects.

Lemma 5.1 *For all TE, values v, and types μ, if $TE \vdash_{tt} v : \mu, \varphi$ then $\varphi = \emptyset$.*

The set of closed expressions is closed under small-step transition.

Lemma 5.2 *If e is closed and $e \rightarrow_r e'$, then e' is also closed.*

Substitution of a value of the correct type for a variable of the same type preserves the type of the enclosing term.

Lemma 5.3 (First Substitution Lemma) *Suppose $TE + \{x \mapsto \mu\} \vdash_{tt} e : \mu', \varphi'$ and $TE \vdash_{tt} v : \mu, \varphi$, then $TE \vdash_{tt} e[v/x] : \mu', \varphi'$.*

Proof. By induction on the derivation of $TE + \{x \mapsto \mu\} \vdash_{tt} e : \mu', \varphi'$.

The only interesting case is the application of *(var)* to (free occurrences of) x. If $TE' = TE + \{x \mapsto \mu\}$, then since $TE'(x) = TE + \{x \mapsto \mu\}(x) = \mu$, rule *(var)* yields $TE' \vdash_{tt} x : \mu, \emptyset$. On the other hand, $x[v/x] = v$ and, by assumption, $TE \vdash_{tt} v : \mu, \emptyset$ since $\varphi = \emptyset$, by Lemma 5.1.

All other cases are simple appeals to the inductive hypothesis. □

Lemma 5.4 *If $TE \vdash_{tt} e : \mu, \varphi$ then, for all substitutions S_s, we have that $S_s(TE) \vdash_{tt} S_s(e) : S_s(\mu), S_s(\varphi)$.*

Proof. By induction on the derivation of $TE \vdash_{tt} e : \mu, \varphi$. Similar to Lemma 5.3 in [16] and Lemma 4.5 in [17]. □

Lemma 5.5 *If $TE + \{f \mapsto (\sigma, \rho)\} \vdash_{tt} e : \mu, \rho'$, and $\sigma \prec \sigma'$, then $TE + \{f \mapsto (\sigma', \rho)\} \vdash_{tt} e : \mu, \rho'$*

Proof. A straightforward induction on the depth of the proof of $TE + \{f \mapsto (\sigma, \rho)\} \vdash_{tt} e : \mu, \rho'$. Proven analogously to lemma 5.4 in [16] and lemma 4.6 in [17]. □

Unfolding a `letrec`-definition also preserves typing.

Lemma 5.6 (Second Substitution Lemma) *Suppose*

(i) $\{\varrho_1 \ldots, \varrho_n, \boldsymbol{\epsilon}, \boldsymbol{\alpha}\} \cap (fv(TE) \cup \{\rho\}) = \emptyset$

(ii) $\{\boldsymbol{\alpha}\} \cap ftv(TE) = \emptyset$

(iii) $\sigma = \forall\, \boldsymbol{\alpha}\, .\hat{\sigma}$ *and* $\hat{\sigma} = \forall\, \boldsymbol{\epsilon}\, .\forall\, \varrho_1 \ldots \varrho_n\, .\tau$

(iv) $TE + \{f \mapsto (\sigma, \rho)\} \vdash_{tt} e_2 : \mu, \varphi$

(v) $TE + \{f \mapsto (\hat{\sigma}, \rho)\} \vdash_{tt} \langle \lambda\, x.\, e_1 \rangle_\rho : (\tau, \rho), \emptyset$

then $TE \vdash_{tt} e_2[\langle \Lambda[\varrho_1, \ldots, \varrho_n].\lambda\, x.\, \texttt{letrec}\ f = \gamma\ \texttt{in}\ e_1 \rangle_\rho / f] : \mu, \varphi$ *where* $\gamma = \langle \Lambda[\varrho_1, \ldots, \varrho_n].\lambda\, x.\, e_1 \rangle_\rho$.

Proof. By induction on the derivation of $TE + \{f \mapsto (\sigma, \rho)\} \vdash_{tt} e_2 : \mu, \varphi$. All cases are straightforward applications of the induction hypothesis, except for free occurrences of f in e_2.

In this case, assumption (iv) is as follows.

$$TE + \{f \mapsto (\sigma, \rho)\} \vdash_{tt} f\ [\rho'_1, \ldots, \rho'_n]\ \texttt{at}\ \rho' : \mu, \varphi \tag{10}$$

This judgment must be due to rule *(recvar1)*, that is, there exists $S_s = (S_t, S_r, S_e)$ such that $\mu = (\tau', \rho')$; $\varphi = \{\rho, \rho'\}$; $(TE + \{f \mapsto (\sigma, \rho)\})(f) = (\sigma, \rho)$; $\tau' \prec \sigma$ *via* S_s; and $S_r = \{\varrho_1 \mapsto \rho'_1, \ldots, \varrho_n \mapsto \rho'_n\}$.

From this, it must be shown using *(recvar2)* that

$$TE \vdash_{tt} \langle \Lambda[\varrho_1, \ldots, \varrho_n].\lambda\, x.\texttt{letrec}\ f = \gamma\ \texttt{in}\ e_1 \rangle_\rho\ [\rho'_1, \ldots, \rho'_n]\ \texttt{at}\ \rho' : (\tau', \rho'), \{\rho, \rho'\}$$

That is, there must exist some τ'' such that

- $TE \vdash_{tt} \langle \lambda\, x.\, \texttt{letrec}\ f = \gamma\ \texttt{in}\ e_1 \rangle_{\rho'} : (\tau'', \rho'), \emptyset$;
- $\{\varrho_1, \ldots, \varrho_n\} \cap (frv(TE) \cup \{\rho\}) = \emptyset$ (this is immediate by assumption (i));
- $\tau' = S_r(\tau'')$ where $S_r = \{\varrho_1 \mapsto \rho'_1, \ldots, \varrho_n \mapsto \rho'_n\}$.

A suitable choice for τ'' is $S'_s(\tau)$ where $S'_s = (S_t, I_r, S_e)$. Without lack of generality, α-conversion of $\varrho_1, \ldots, \varrho_n$ in γ ensures that $\varrho_1, \ldots, \varrho_n$ do not occur in the range of S_s, so that $S_s = S_r S'_s$. Since $\tau' \prec \sigma$ *via* S_s, τ'' satisfies the last requirement. It remains to show that

$$TE \vdash_{tt} \langle \lambda\, x.\, \texttt{letrec}\ f = \gamma\ \texttt{in}\ e_1 \rangle_{\rho'} : (S'_s(\tau'), \rho'), \emptyset \tag{11}$$

To see this, it is first shown that

$$TE \vdash_{tt} \langle \lambda\, x.\, \texttt{letrec}\ f = \gamma\ \texttt{in}\ e_1 \rangle_\rho : (\tau, \rho), \emptyset \tag{12}$$

To this end, suppose that $\tau = \mu_1 \xrightarrow{\epsilon'.\varphi'} \mu_2$. Then equation (12) follows from assumption (v), *(use-abstr)*, and from the implication: if, for some $\varphi'' \subseteq \varphi'$,

$$TE + \{f \mapsto (\hat{\sigma}, \rho)\} + \{x \mapsto \mu_1\} \vdash_{tt} e_1 : \mu_2, \varphi'' \tag{13}$$

then

$$TE + \{x \mapsto \mu_1\} \vdash_{tt} \texttt{letrec}\ f = \gamma\ \texttt{in}\ e_1 : \mu_2, \varphi'' \tag{14}$$

From assumption (v) it follows trivially that

$$TE + \{x \mapsto \mu_1\} + \{f \mapsto (\hat{\sigma}, \rho)\} \vdash_{tt} \langle \lambda\, x.\, e_1 \rangle_\rho : (\tau, \rho), \emptyset \tag{15}$$

Applying Lemma (5.5) to equation (13) for $\hat{\sigma} \prec \sigma$ and observing that $TE + \{f \mapsto (\hat{\sigma}, \rho)\} + \{x \mapsto \mu_1\} = TE + \{x \mapsto \mu_1\} + \{f \mapsto (\hat{\sigma}, \rho)\}$ because $x \neq f$ yields

$$TE + \{x \mapsto \mu_1\} + \{f \mapsto (\sigma, \rho)\} \vdash_{tt} e_1 : \mu_2, \varphi'' \tag{16}$$

Applying rule *(use-letrec)* to assumption (i), equation (15), and equation (16) yields the claim in equation (14).

By rule *(use-abstr)*, equation (12) is derivable from equation (14).

Applying Lemma 5.4 to equation (12) for S'_s yields

$$S'_s(TE) \vdash_{tt} S'_s(\langle \lambda\, x.\, \texttt{letrec}\ f = \gamma\ \texttt{in}\ e_1 \rangle_\rho) : S'_s(\tau, \rho), S'_s(\emptyset) \tag{17}$$

The domain of $S'_s = (S_t, I_r, S_e)$ is a subset of $\{\boldsymbol{\epsilon}, \boldsymbol{\alpha}\}$ since $\tau' \prec \sigma$ *via* (S_t, S_r, S_e). By assumption (i), $\{\boldsymbol{\epsilon}, \boldsymbol{\alpha}\}$ is disjoint from $fv(TE) \cup \{\rho\}$. Therefore,

- $S'_s(TE) = TE$;
- $S'_s(\langle \lambda\, x.\, \texttt{letrec}\ f = \gamma\ \texttt{in}\ e_1 \rangle_\rho) = \langle \lambda\, x.\, \texttt{letrec}\ f = \gamma\ \texttt{in}\ e_1 \rangle_\rho$ due to I_r; and
- $S'_s(\tau, \rho) = (S'_s(\tau), \rho)$.

Using the rule *(use-abstr)* once backwards and once forwards replaces ρ by ρ' and transforms equation (17) into equation (11), which proves the claim. □

5.2 Type preservation

The following proposition states that for every well-typed term which has a transition, the reduct has exactly the same type as the redex, but possibly less effect.

Proposition 5.7 (Type Preservation) *Suppose* $TE \vdash_{tt} e : \mu, \varphi$. *If* $e \to_r e'$ *then* $TE \vdash_{tt} e' : \mu, \varphi'$ *where* $\varphi' \subseteq \varphi$.

Proof. By induction on the definition of $\to_r$ and the number of subsequent uses of *(effect)* at the end of e's type derivation.

If $TE \vdash_{tt} e : \mu, \varphi$ derives from $TE \vdash_{tt} e : \mu, \varphi \cup \{\epsilon\}$ by rule *(effect)*, for some $\epsilon \notin fev(TE, \mu, \varphi)$, then induction yields that $TE \vdash_{tt} e' : \mu, \varphi'$ where

$\varphi' \subseteq \varphi \cup \{\epsilon\}$. If $\epsilon \notin \varphi'$ then the claim holds anyway. Otherwise, use *(effect)* to get $TE \vdash_{tt} e' : \mu, \varphi' \setminus \{\epsilon\}$ where $\varphi' \setminus \{\epsilon\} \subseteq \varphi \cup \{\epsilon\} \setminus \{\epsilon\} = \varphi$, as required.

If the last rule in the proof of $TE \vdash_{tt} e : \mu, \varphi$ is not *(effect)* then perform a case analysis. Most cases are straightforward applications of the inductive hypothesis and/or one of the preceding lemmas.

Case c `at` $\varrho \to_r \langle c \rangle_\varrho$ is obvious by rules *(alloc-const)* and *(use-const)*.

Case $\lambda x.\, e$ `at` $\varrho \to_r \langle \lambda x.\, e \rangle_\varrho$ is obvious by rules *(alloc-abstr)* and *(use-abstr)*.

Case `letrec` $f = a$ `at` ϱ `in` $e \to_r$ `letrec` $f = \langle a \rangle_\varrho$ `in` e is obvious by rules *(alloc-letrec)* and *(use-letrec)*.

Case `copy` $[\varrho_1, \varrho_2]\, \langle c \rangle_{\varrho_1} \to_r \langle c \rangle_{\varrho_2}$. By assumption, $TE \vdash_{tt}$ `copy` $[\varrho_1, \varrho_2]\, \langle c \rangle_{\varrho_1} : \mu, \varphi$, for some μ and φ. Hence, by rules *(copy)* and *(use-const)*, $\mu = (\texttt{int}, \varrho_2)$ and $\varphi = \{\varrho_1, \varrho_2\}$. For the reduct, rule *(use-const)* yields $TE \vdash_{tt} \langle c \rangle_{\varrho_2} : (\texttt{int}, \varrho_2), \emptyset$, which verifies the claim.

Case $\langle \lambda x.\, e \rangle_\varrho @ v \to_r e[v/x]$. The last rule in the typing proof of the redex must be *(app)*, hence (using Lemma 5.1)

$$\frac{TE \vdash_{tt} \langle \lambda x.\, e \rangle_\varrho : (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \varrho), \emptyset \quad TE \vdash_{tt} v : \mu_1, \emptyset}{TE \vdash_{tt} (\langle \lambda x.\, e \rangle_\varrho) \mathbin{@} v : \mu_2, \varphi \cup \{\epsilon, \varrho\}}$$

By rule *(use-abstr)* it must be that $TE + \{x \mapsto \mu_1\} \vdash_{tt} e : \mu_2, \varphi'$ for some $\varphi' \subseteq \varphi$. By the First Substitution Lemma, it follows that $TE \vdash_{tt} e[v/x] : \mu_2, \varphi'$. Clearly, $\varphi' \subseteq \varphi \cup \{\epsilon, \varrho\}$, which proves the claim.

Case `letregion` ϱ `in` $v \to_r v[\bullet/\varrho]$. The last rule in the typing proof of the redex must have been *(letregion)*:

$$\frac{TE \vdash_{tt} v : \mu, \varphi' \quad \varrho \notin frv(TE, \mu)}{TE \vdash_{tt} \texttt{letregion}\ \varrho\ \texttt{in}\ v : \mu, \varphi' \setminus \{\varrho\}}$$

By Lemma 5.1, $\varphi' = \emptyset$, hence $\varphi' \setminus \{\varrho\} = \emptyset$. By Lemma 5.4, $TE[\bullet/\varrho] \vdash_{tt} v[\bullet/\varrho] : \mu[\bullet/\varrho], \varphi'[\bullet/\varrho]$. Since $\varrho \notin frv(TE, \mu)$ and $\varphi' = \emptyset$, this amounts to $TE \vdash_{tt} v[\bullet/\varrho] : \mu, \emptyset$, which verifies the claim.

Case `letrec` $f = \gamma$ `in` $e_2 \to_r e_2[\langle \Lambda[\varrho_1, \ldots, \varrho_n].\lambda x.\texttt{letrec}\ f = \gamma\ \texttt{in}\ e_1 \rangle_\rho / f]$ where $\gamma = \langle \Lambda[\varrho_1, \ldots, \varrho_n].\lambda x.\, e_1 \rangle_\rho$. The last step in the typing derivation for the redex must apply rule *(use-letrec)*:

$$\frac{\begin{array}{c}\{\varrho_1 \ldots, \varrho_n, \boldsymbol{\epsilon}\} \cap (fv(TE) \cup \{\rho\}) = \emptyset \quad \hat{\sigma} = \forall\, \boldsymbol{\epsilon}\, . \forall\, \varrho_1 \ldots \varrho_n\, . \tau \\ TE + \{f \mapsto (\hat{\sigma}, \rho)\} \vdash_{tt} \langle \lambda x.\, e_1 \rangle_\rho : (\tau, \rho), \emptyset \\ \{\boldsymbol{\alpha}\} \cap ftv(TE) = \emptyset \quad \sigma = \forall\, \boldsymbol{\alpha}\, . \hat{\sigma} \quad TE + \{f \mapsto (\sigma, \rho)\} \vdash_{tt} e_2 : \mu, \varphi\end{array}}{TE \vdash_{tt} \texttt{letrec}\ f = \langle \Lambda[\varrho_1, \ldots, \varrho_n].\lambda x.\, e_1 \rangle_\rho\ \texttt{in}\ e_2 : \mu, \varphi}$$

By the Second Substitution Lemma, this yields
$TE \vdash_{tt} e_2[\langle \Lambda[\varrho_1, \ldots, \varrho_n].\lambda\, x.\, \texttt{letrec}\ f = \gamma\ \texttt{in}\ e_1 \rangle_\rho / f] : \mu, \varphi$ as required.

Case $\langle \Lambda[\varrho_1, \ldots, \varrho_n].\lambda\, x.e \rangle_\varrho\ [\rho'_1, \ldots, \rho'_n]\ \texttt{at}\ \varrho' \to_r \langle \lambda\, x.e[\rho'_1/\varrho_1, \ldots, \rho'_n/\varrho_n] \rangle_{\varrho'}$. The last step of the typing derivation for the redex is *(recvar2)*:

$$\frac{TE \vdash_{tt} \langle \lambda\, x.\, e \rangle_{\varrho'} : (\tau, \varrho'), \emptyset \qquad S_r = \{\varrho_1 \mapsto \rho'_1, \ldots, \varrho_n \mapsto \rho'_n\} \quad \tau' = S_r(\tau) \quad \{\varrho_1, \ldots, \varrho_n\} \cap frv(TE) = \emptyset}{TE \vdash_{tt} \langle \Lambda[\varrho_1, \ldots, \varrho_n].\lambda\, x.\, e \rangle_\varrho\ [\rho'_1, \ldots, \rho'_n]\ \texttt{at}\ \varrho' : (\tau', \varrho'), \{\varrho, \varrho'\}}$$

So, by Lemma 5.4, we have $S_r(TE) \vdash_{tt} S_r(\langle \lambda\, x.\, e \rangle_{\varrho'}) : S_r(\tau, \varrho'), \emptyset$, but since $S_r(\tau) = \tau'$ and $\{\varrho_1, \ldots, \varrho_n\} \cap frv(TE) = \emptyset$, we have that
$TE \vdash_{tt} \langle \lambda\, x.\, e[\rho'_1/\varrho_1, \ldots, \rho'_n/\varrho_n] \rangle_{\varrho'} : (\tau', \varrho'), \emptyset$.

Case $e \to_r e'$ implies $e \mathbin{@} e_0 \to_r e' \mathbin{@} e_0$. The last step in the typing derivation of the left term is the rule *(app)*:

$$\frac{TE \vdash_{tt} e : (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho), \varphi_1 \qquad TE \vdash_{tt} e_0 : \mu_1, \varphi_2}{TE \vdash_{tt} e \mathbin{@} e_0 : \mu_2, \varphi \cup \varphi_1 \cup \varphi_2 \cup \{\epsilon, \rho\}}$$

By induction, $TE \vdash_{tt} e' : (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho), \varphi'_1$ where $\varphi'_1 \subseteq \varphi_1$. By rule *(app)*,

$$\frac{TE \vdash_{tt} e' : (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho), \varphi'_1 \qquad TE \vdash_{tt} e_0 : \mu_1, \varphi_2}{TE \vdash_{tt} e \mathbin{@} e_0 : \mu_2, \varphi \cup \varphi'_1 \cup \varphi_2 \cup \{\epsilon, \rho\}}$$

and $\varphi \cup \varphi'_1 \cup \varphi_2 \cup \{\epsilon, \rho\} \subseteq \varphi \cup \varphi_1 \cup \varphi_2 \cup \{\epsilon, \rho\}$.

Case $e \to_r e'$ implies $v \mathbin{@} e \to_r v \mathbin{@} e'$. Similarly simple appeal to the inductive hypothesis.

Case $e \to_r e'$ implies $\texttt{copy}\,[\varrho_1, \varrho_2]\, e \to_r \texttt{copy}\,[\varrho_1, \varrho_2]\, e'$. Similarly simple appeal to the inductive hypothesis.

Case $e \to_r e'$ implies $\texttt{letregion}\ \varrho\ \texttt{in}\ e \to_r \texttt{letregion}\ \varrho\ \texttt{in}\ e'$. Similarly simple appeal to the inductive hypothesis. □

5.3 Canonical Forms

A canonical forms lemma determines the form of a value, given its type.

Lemma 5.8 (Canonical Forms) (i) *If* $TE \vdash_{tt} v : (\mu_1 \xrightarrow{\epsilon.\varphi'} \mu_2, \rho), \varphi$ *then there are* x *and* e *so that* $v = \langle \lambda\, x.\, e \rangle_\rho$.

(ii) *If* $TE \vdash_{tt} v : (\texttt{int}, \rho), \varphi$ *then there is some constant* c *such that* $v = \langle c \rangle_\rho$.

5.4 Progress

The progress property states that a well-typed term is either a syntactic value or can be further reduced, if it is closed and does not affect a dead region.

Proposition 5.9 (Progress) *If* $TE \vdash_{tt} e : \mu, \varphi$ *and* $\bullet \notin \varphi$ *then either*

(i) *there exists* e' *such that* $e \to_r e'$ *or*

(ii) e *is a value or*

(iii) e *has form* $\mathcal{E}[x]$*, with* $x \in \mathit{free}(e)$ *or*

(iv) e *has from* $\mathcal{E}[f\ [\rho_1, \ldots, \rho_n]\ \mathtt{at}\ \varrho]$*, with* $f \in \mathit{free}(e)$*.*

Proof. Easy induction on the structure of e:

Case x: Item iii applies;

Case $\langle c \rangle_\rho$: Item (ii) applies;

Case $\langle \lambda\, x.\, e \rangle_\rho$: Item (ii) applies;

Case $c\ \mathtt{at}\ \rho$: By *(alloc-const)*, $TE \vdash_{tt} c\ \mathtt{at}\ \rho : (\mathtt{int}, \rho), \{\rho\}$ and, by assumption, $\rho \neq \bullet$. Hence, Item (i) applies with reduction (1);

Case $\lambda\, x.\, e\ \mathtt{at}\ \rho$: By *(alloc-abstr)*, $TE \vdash_{tt} \lambda\, x.\, e\ \mathtt{at}\ \rho : \mu, \{\rho\}$ and, by assumption, $\rho \neq \bullet$. Hence, Item (i) applies with reduction (2);

Case $e_1 \mathbin{@} e_2$: by induction, there are the following cases for e_1:

- one of Item (i), Item (iii), or Item (iv) applies to e_1 with evaluation context $\mathcal{E}$. Since $\mathcal{E} \mathbin{@} e_2$ is an evaluation context, too, the respective case applies to $e_1 \mathbin{@} e_2$.
- Item (ii) applies to e_1. By typability, $TE \vdash_{tt} e_1 \mathbin{@} e_2 : \mu, \varphi$. This must be due to rule *(app)*, so that $TE \vdash_{tt} e_1 : (\mu_1 \xrightarrow{\epsilon.\varphi_2} \mu, \rho), \varphi_1$, where $\{\epsilon, \rho\} \cup \varphi_2 \cup \varphi_1 \subseteq \varphi$. By Lemma 5.8, e_1 has the form $\langle \lambda\, x.\, e \rangle_\rho$ where $\rho \neq \bullet$, by assumption. Again, by induction, there are the following cases for e_2:
  - one of Item (i), Item (iii), or Item (iv) applies to e_2 with evaluation context $\mathcal{E}$. Since $e_1 \mathbin{@} \mathcal{E}$ is an evaluation context, too, the respective case applies to $e_1 \mathbin{@} e_2$.
  - Item (ii) applies to e_2, so that $e_1 \mathbin{@} e_2$ is a beta-redex. Hence, Item (i) applies with reduction (6) since $\rho \neq \bullet$.

Case $\mathtt{letrec}\ f = a\ \mathtt{at}\ \rho\ \mathtt{in}\ e$: by typability and $\rho \neq \bullet$, Item (i) applies with reduction (3);

Case $\mathtt{letrec}\ f = \langle a \rangle_\rho\ \mathtt{in}\ e$: Item (i) applies with reduction (7);

Case $\mathtt{copy}\,[\rho, \rho']\, e$: by induction, there are the following cases for e:

- one of Item (i), Item (iii), or Item (iv) applies to e with evaluation context $\mathcal{E}$. Since $\mathtt{copy}\,[\rho_1, \rho_2]\, \mathcal{E}$ is an evaluation context, too, the respective case applies to $\mathtt{copy}\,[\rho_1, \rho_2]\, e$.
- Item (ii) applies to e. By typability, $TE \vdash_{tt} \mathtt{copy}\,[\rho, \rho']\, e : (\mathtt{int}, \rho'), \varphi$. By rule *(copy)*, it must be that $TE \vdash_{tt} e : (\mathtt{int}, \rho), \varphi'$ where $\varphi = \varphi' \cup \{\rho, \rho'\}$. By Lemma 5.8, $e = \langle c \rangle_{\rho_1}$. Furthermore, by assumption and typability,

neither ρ nor ρ' can be $\bullet$. Therefore, Item (i) applies with reduction (5).

Case `letregion` ϱ `in` e: by induction, there are the following cases for e:

- one of Item (i), Item (iii), or Item (iv) applies to e with evaluation context $\mathcal{E}$. Since `letregion` ϱ `in` $\mathcal{E}$ is also an evaluation context, the respective case applies to `letregion` ϱ `in` e.
- Item (ii) applies to e. Therefore, Item (i) applies with reduction (4).

Case $f\ [\rho_1, \ldots, \rho_n]$ `at` ρ': Item (iv) applies.

Case $\langle \Lambda[\varrho_1, \ldots, \varrho_n].\lambda\, x.\, e\rangle_\rho\ [\rho_1, \ldots, \rho_n]$ `at` ρ': by typability and assumption, neither ρ nor ρ' is $\bullet$; hence, Item (i) applies with reduction (8).

□

5.5 Soundness

Finally, the type soundness theorem says that a well-typed closed term either gives rise to an infinite reduction sequence, or it eventually reduces to a value of the same type.

Theorem 5.10 (Type Soundness) *Suppose e is a closed surface term and $[\,] \vdash_{tt} e : (\tau, \varrho), \varphi$ with $\bullet \notin \varphi$. Then, either there exists some value v with $e \twoheadrightarrow_r v$ and $[\,] \vdash_{tt} v : (\tau, \varrho), \emptyset$ or, for each e' where $e \twoheadrightarrow_r e'$, there exists e'' with $e' \to_r e''$.*

Proof. Immediate consequence of propositions 5.7 and 5.9 and lemmas 5.1 and 5.2 □

5.6 Discussion

The proof as presented here is a slight variation of the original strategy of Wright and Felleisen [17]. Instead of proving that it is always possible to continue evaluation of a typable closed term unless the term is already a value (the progress property), they prove the contraposition. This requires the introduction of *stuck* expressions: an expression e is stuck if e is not a value and there is no e' for which $e \to_r e'$. Then one approximates the set of expressions that become stuck by the set of *faulty expressions*, for which the following can be shown: if a closed expression cannot be reduced to a faulty expression, evaluation either does not terminate or returns a value. If faulty expressions coincide with untypable expressions, type soundness follows by type preservation.

We believe it is more natural to show progress as in proposition 5.9, instead of using the unintuitive and superfluous notion of stuck states.

6 Example

It is instructive to look at an example of small-step evaluation and see how it preserves typing. Assume the usual semantics for `let` $x = e_1$ `in` e_2 and

suppose an existing ϱ_1 :

$$
\begin{array}{l}
(\texttt{letregion}\ \varrho_2\ \texttt{in} \\
\quad \texttt{let}\ f_1 = \lambda x.\, x\ \texttt{at}\ \varrho_2\ \texttt{in} \\
\quad\quad \texttt{let}\ f_2 = \lambda f.\, (4\ \texttt{at}\ \varrho_1)\ \texttt{at}\ \varrho_1\ \texttt{in} \\
\quad\quad\quad \lambda y.\, (f_2 \mathbin{@} f_1)\ \texttt{at}\ \varrho_1 \qquad\qquad)\ \mathbin{@}\ (2\ \texttt{at}\ \varrho_1) \twoheadrightarrow_r \\
\texttt{letregion}\ \varrho_2\ \texttt{in}\ \lambda y.\, (\langle \lambda f.\, 4\ \texttt{at}\ \varrho_1 \rangle_{\varrho_1} \mathbin{@} \langle \lambda x.\, x \rangle_{\varrho_2})\ \texttt{at}\ \varrho_1 \mathbin{@} (2\ \texttt{at}\ \varrho_1) \twoheadrightarrow_r \\
\langle \lambda y.\, (\langle \lambda f.\, 4\ \texttt{at}\ \varrho_1 \rangle_{\varrho_1} \mathbin{@} \langle \lambda x.\, x \rangle_\bullet) \rangle_{\varrho_1} \mathbin{@} \langle 2 \rangle_{\varrho_1} \twoheadrightarrow_r \\
\langle \lambda f.\, 4\ \texttt{at}\ \varrho_1 \rangle_{\varrho_1} \mathbin{@} \langle \lambda x.\, x \rangle_\bullet \twoheadrightarrow_r \\
\langle 4 \rangle_{\varrho_1}
\end{array}
$$

All intermediate terms are typable with the rules of Figures 3 and 4. The lambda abstraction $\lambda x.\, x\ \texttt{at}\ \rho_2$ is first allocated in ρ_2, which is safely deallocated after evaluation of $\lambda y.\, \ldots\ \texttt{at}\ \varrho_1$, since ϱ_2 does neither occur free in the type of this lambda, nor in its environment. The dangling pointer $\langle \lambda x.\, x \rangle_\bullet$ remains visible, but it is never dereferenced and disappears eventually.

7 Conclusion and further work

We have presented a small-step operational semantics for the region calculus and given a syntactic type soundness proof. Since it is solely based on rewriting and induction, the proof is considerably easier than the original soundness proof of Tofte and Talpin.

We were able to elide an explicit store from our presentation of the semantics because the region calculus of Tofte and Talpin never updates the contents of the store. Including references in our framework should be possible at the price of including an explicit store component in the transition relation.

The original motivation for this work is the desire to use the region calculus as a foundation for the binding-time analysis phase of *offline partial evaluation* [5, 12, 11]. Region inference seems to provide exactly the right kind of flow analysis for binding-time analysis in programming languages with ML-style polymorphism.

We are currently working on a binding-time annotated version of the region calculus. The corresponding type soundness result amounts to the correctness of the binding-time analysis. It remains to show the other properties (for instance semantics preservation), in analogy to the results of Hatcliff and Danvy [10] for a monomorphic version of the computational metalanguage.

References

[1] Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. Region analysis and the polymorphic lambda calculus. In *Proc. of the 14th Annual IEEE symposium on Logic in Computer Science*, Trento, Italy, July 1999. IEEE Computer Society Press.

[2] Lars Birkedal and Mads Tofte. A constraint-based region inference algorithm. *Theoretical Computer Science*, to appear, 1999.

[3] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Proc. 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 171–183, St. Petersburg, Fla., January 1996. ACM Press.

[4] Cristiano Calcagno. Stratified operational semantics for safety and correctness of the region calculus. Personal communication, July 2000. Draft obtainable from ftp://ftp.disi.unige.it/person/CalcagnoC/regions.ps.

[5] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Proc. 20th Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.

[6] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In Alex Aiken, editor, *Proc. 26th Annual ACM Symposium on Principles of Programming Languages*, pages 262–275, San Antonio, Texas, USA, January 1999. ACM Press.

[7] H. B. Curry, R. Feys, and W. Craig. *Combinatory Logic*, volume I. North Holland, 1958.

[8] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proc. 9th Annual ACM Symposium on Principles of Programming Languages*, pages 207–212. ACM, 1982.

[9] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 10(2):235–271, 1992.

[10] John Hatcliff and Olivier Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, 7(5):507–542, 1997.

[11] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.

[12] Torben Æ. Mogensen and Peter Sestoft. Partial evaluation. In Allen Kent and James G. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 37, pages 247–279. Marcel Dekker, 270 Madison Avenue, New York, New York 10016, 1997.

[13] Mads Tofte and Lars Birkedal. Unification and polymorphism in region inference. In *Milner Festschrift*, 1996.

[14] Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(5):724–767, 1998.

[15] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proc. 21st Annual ACM Symposium on Principles of Programming Languages*, pages 188–201, Portland, OG, January 1994. ACM Press.

[16] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[17] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 15 November 1994.

[18] S. Dal Zilio and Andrew Gordon. Region analysis and a pi-calculus with groups. In *Proceedings of MFCS '00*, Lecture Notes in Computer Science, 2000.