A Simplified Account of Region Inference

Jean-Pierre Talpin

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A simplified account of region inference

Jean-Pierre Talpin

N˚4104

January 2001

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--4104--FR+ENG

INRIA
RENNES

A simplified account of region inference

Jean-Pierre Talpin

Thème 1 — Réseaux et systèmes
Projet Ep-Atr

Rapport de recherche n° 4104 — January 2001 — 30 pages

Abstract: Region inference is a program analysis which statically determines regions of memory in which values are stored. Regions are allocated and de-allocated according to a stack discipline. Region inference is an attractive alternative to dynamic memory management via garbage collection for the implementation of mission-critical applications, of inter-operating system components, of certifiable software. In the aim of putting region-based memory management to work for the design and implementation of real-time system software, the goal of the present article is to consolidate present knowledge on region-based memory management by giving a simplified account of region inference. We achieve this goal by giving an inductive proof of correctness for region inference w.r.t. a small step operational semantics and by defining a sound, fixed-point iterative, inference algorithm.

Key-words: Type inference, program analysis, memory management

(Résumé : tsvp)

L'inférence de régions pour les nuls

Résumé : L'inférence de régions est une analyse de programmes qui consiste à déterminer des régions de mémoire dans lesquelles les valeurs d'un programmes sont stockées. Les régions sont allouées et désallouées conformément à une discipline de pile. L'inférence de régions est une alternative attractive à la gestion dynamique de mémoire au moyen d'un "glaneur de cellules", tout spécialement s'il est question de mettre en œuvre un système temps-réel au moyen d'un langage de programmation de haut-niveau.

Mots-clé : Inférence de types, analyse de programmes, gestion mémoire

Contents

1 Introduction

Region inference is a program analysis which consists of determining a compile-time approximation of the dynamic lifetime of objects by associating regions to values and by collecting the use of values in given regions. A region calculus defines the corresponding typed intermediate representation of functional programs. It implements a memory management protocol that is fully synthesized at compile-time instead of being delegated at runtime to a garbage collection algorithm. This region management protocol consists of operating a stack. Regions reference blocks of memories. Regions are pushed on the stack when created, used to indicate where to allocate objects, and poped from the stack to reclaim memory blocks.

Region-based memory management is an attractive alternative to the dynamic management of memory via a garbage collection algorithm. Region inference significantly reduces the amount of memory required to run programs. Region inference does not require any pause during execution: allocation and deallocation of memory regions are constant-time operations. Region inference consists of explicit program annotations which are automatically checkable.

Region-based memory management is thus is a good candidate for the implementation of mission-critical certifiable software. Comparisons on running times demonstrate that region technology is competitive with other existing implementation technology. Finally, local program optimizations which are made possible by region inference , such as favoring stack-allocation of data, are also useful in the presence of garbage collection. Thus one can use region inference as a development tool for decreasing memory consumption, even in the presence of garbage collection.

The basic idea of a region calculus is to partition storage into regions, and to perform the management of regions according to a stack discipline. Each value, be it heap-allocated or stack-allocated, is annotated with the particular region which denotes the abstract, approximate, location into which it is stored. The construct e/r restricts the scope of a region r to the expression e. It is used to manage the allocation and de-allocation of objects belonging to the region r during the evaluation of expression e. When the expression e/r is entered, a region r is created and pushed on the stack. During the execution of e, objects are allocated into that region. When it is exited, the region r is poped from the stack and all objects linked to that region are deallocated. An effect system for the region calculus guarantees the safety of de-allocating the defunct region r after the evaluation of expression e. A region inference algorithm compiles to the region calculus by computing suitable region annotations for values, and inserting restriction constructs where necessary.

In the aim of putting region-based memory management to work for the design and implementation of real-time embedded systems, the goal of the present article is to consolidate the basic principles of region-based memory management by

giving a simplified account of region inference. To achieve this goal, we give an inductive proof of correctness for region inference with respect to an intuitive, small-step, operational semantics of the λ-calculus with regions. The proof makes use of the technique of syntactic soundness of Wright and Feleisen [22]. Next, we give a simple, iterative, inference algorithm for region inference in the presence of polymorphic recursion. We prove the termination and the correctness of this algorithm with respect to the inference system of the region calculus by borrowing some results on fixed-point theory to the work of Cousot [5].

2 A minimal region calculus

To focus on the essentials of region inference, we consider a minimal subset of the Tofte-Talpin calculus, that consists of the call-by-value λ-calculus (grammar $\boldsymbol{e}$ of figure 6 ahead), region-polymorphic recursive functions $\mathsf{rec}\ f[\boldsymbol{r}]e$ and elementary region annotations: $[r]e$ for parameterizing an expression e by a region r, $e\,\mathsf{at}\,s$ for instantiating it with s and e/r for lexically restricting the scope of a region r to an expression e.

2.1 Syntax

We consider an infinite set of variables $x, y, z \in X$ and of regions $r, s \in R$. We write $\boldsymbol{r}$ for a sequence of regions (grammar $\boldsymbol{r}$ of figure 1). We write $(r :: \boldsymbol{s})$ for adding r to a sequence $\boldsymbol{s}$ and $\boldsymbol{s}\backslash r$ for removing it.

Values v are either λ-abstractions $(\mathsf{fun}\ x.e)_r$ allocated at a given region r or recursive values $\mathsf{rec}\ x[\boldsymbol{r}]\,v$ of parameterized by the regions $\boldsymbol{r}$.

Expressions are defined by the grammar e of figure 1. The expression $x\,\mathsf{at}\,\boldsymbol{r}$ instantiates the value of x at the regions $\boldsymbol{r}$. The expression $(\mathsf{fun}\,x.e)\,\mathsf{at}\,r$ allocates the λ-abstractions $\mathsf{fun}\ x.e$ at the region r. The expression $\mathsf{rec}\ x[\boldsymbol{r}]\,e$ defines the recursive value x by the value of e (e is restricted to the form $(\mathsf{fun}\ x.e)\,\mathsf{at}\,s$). It is parameterized by the regions $\boldsymbol{r}$. The expression e/r limits the extend or scope of the region r to the expression e. Function application is written $e\,e'$. We identify v to $v\,\mathsf{at}\,[]$ as well as $(e/r)/s$, $(e/s)/r$ to $e/(r :: s)$.

v	::=	$(\mathsf{fun}\ x.e)_r \mid \mathsf{rec}\ x[\boldsymbol{r}]\,v$	(value)
e	::=	$x\,\mathsf{at}\,\boldsymbol{r} \mid (\mathsf{fun}\ x.e)\,\mathsf{at}\,r \mid \mathsf{rec}\ x[\boldsymbol{r}]\,e \mid e/r \mid e\,e' \mid v$	(expression)
c	::=	$[] \mid \mathsf{rec}\ x[\boldsymbol{r}]\,c \mid c/r \mid c\,e \mid v\,c$	(context)

Figure 1: Syntax of the λ-calculus with regions

As in Wright and Feleisen's work, we consider an additional syntactic category for evaluation contexts c (grammar c of figure 1). We write [] for the empty

context, ce and oc for the operator and operand contexts of function application and $\mathsf{rec}\ x[\boldsymbol{r}]\,c$ for the context that corresponds to allocating the value of a recursive definition.

2.2 Small-step operational semantics of the region calculus

The semantics of an expression e is defined by the relation $e \xrightarrow{\boldsymbol{r}} e'$ and $e \xrightarrow{\boldsymbol{r}}\!\!\!\!\rightarrow v$ (figure 2). The relation $e \xrightarrow{\boldsymbol{r}} e'$ defines the elementary transitions of the evaluation relation. The axiom (rec) unfolds the recursive object $\mathsf{rec}\, f[\boldsymbol{r}]v$ and instantiates it at the regions $\boldsymbol{s}$ (we identify $\xrightarrow{[]}$ to $\rightharpoonup$). This transition is obtained by substituting the region parameters $\boldsymbol{r}$ by the sequence $\boldsymbol{s}$ in v (depicted by the term $v[\boldsymbol{s}/\boldsymbol{r}]$) and by substituting the identifier f by $\mathsf{rec}\, f[\boldsymbol{r}]v$ (depicted by the term $(\_)[\mathsf{rec}\, f[\boldsymbol{r}]v/f]$).

The axiom (abs) allocates a λ-abstraction $\mathsf{fun}\ x.e$ at the region r, keeps track of the use of the region r over the arrow $\xrightarrow{r}$ and returns the object $(\mathsf{fun}\ x.e)_r$. The axiom (app) defines the transition that corresponds to the application of a λ-abstraction $\mathsf{fun}\ x.e$ at the region r to the argument o. The transition is obtained by substituting x by v in e. The region r is again kept book of over the arrow $\xrightarrow{r}$.

Now, notice that both axioms (abs) and (app) are subject to a side-condition which stipulates that the given region r should not be the region constant free. The free region is introduced by the axiom (free) that defines the transition for a fully evaluated restriction expression v/r. This transition consists of reclaiming every object allocated in the region r by sending it back to the free space area. This operation is rendered by a substitution of the term r by the term free in v. In doing so, any object of region r captured by v becomes unusable, as defined in the axioms (abs) and (app).

The relation $e \xrightarrow{\boldsymbol{r}}\!\!\!\!\rightarrow e'$ is the transitive closure of the relation $e \xrightarrow{r} e'$. It is defined by the rule (trans) that assembles the elementary transitions promoted by the rules (cont) and (mask). The rules (trans) and (cont) are standard, albeit the side-condition $\boldsymbol{r} \notin \mathrm{br}(c)$ of the rule (cont), which is here to make sure that $\boldsymbol{r}$ (which is of the form [] or r in this rule, by definition of $\rightharpoonup$) does not cross the boundary of its lexical scope. The rule (mask) starts from a transition $\xrightarrow{r}\!\!\!\!\rightarrow$ that affects the region r and defines the scope at which maskign r from $\rightarrow\!\!\!\!\rightarrow$ is legal. Note that no side-condition $r \neq \mathsf{free}$ is required here since no axiom can produce the transition $\xrightarrow{\mathsf{free}}$.

The closure $e \xrightarrow{\boldsymbol{r}}\!\!\!\!\rightarrow v$ of the relation $\rightarrow\!\!\!\!\rightarrow$ defines the class of expressions e which yield to a result v by using the regions $\boldsymbol{r}$. All expressions do not, however, converge to a result. Reactive systems for instance, usually consist of infinitely iterating loops. We write $e \Uparrow^{\boldsymbol{r}}$ for the relation that denotes the divergent run of an expression e. We write $e \Uparrow^{\boldsymbol{r}}$ iff there exists an infinite sequence $(e_n \xrightarrow{\boldsymbol{r}_n}\!\!\!\!\rightarrow e_{n+1})_{n\geq 0}$ given $e_0 = e$ and $\boldsymbol{r}_n \subseteq \boldsymbol{r}$ for all $n \geq 0$.

In order to keep the operational semantics as simple as possible, we do not make the error axioms and error propagation rules explicit in the operational semantics. This variant of [22] is introduced by Harper in [8]. Accordingly, we write $e \not\rightarrow\!\!\!\!\!\!\triangle$ for an expression that is stuck in that does not admit any transition by the axioms of the figure 2. By extension, we write $e \not\rightarrow\!\!\!\!\!\!\triangle$ iff $e' \not\rightarrow\!\!\!\!\!\!\triangle$ holds for all contexts $c[]$ such that $e = c[e']$.

$$(\mathsf{rec}\, f[\boldsymbol{r}]v)\,\mathsf{at}\,\boldsymbol{s} \;\rightharpoonup\; (v[\boldsymbol{s}/\boldsymbol{r}])[\mathsf{rec}\, f[\boldsymbol{r}]v/f] \qquad (\mathrm{rec})_{|\boldsymbol{r}|=|\boldsymbol{s}|}$$

$$(\mathsf{fun}\; x.e)\,\mathsf{at}\, r \;\xrightarrow{r}\; (\mathsf{fun}\; x.e)_r \qquad (\mathrm{abs})_{r\neq\mathsf{free}}$$

$$(\mathsf{fun}\; x.e)_r\, v \;\xrightarrow{r}\; e[v/x] \qquad (\mathrm{app})_{r\neq\mathsf{free}}$$

$$v/r \;\rightharpoonup\; v[\mathsf{free}/r] \qquad (\mathrm{free})$$

$$\frac{e \xrightarrow{\boldsymbol{r}} e'}{c[e] \xrightarrow{\boldsymbol{r}} c[e']}\;(\mathrm{cont})_{\boldsymbol{r}\notin \mathrm{br}(c)} \qquad \frac{e \xrightarrow{r} e'}{e/r \rightharpoonup e'/r}\;(\mathrm{mask}) \qquad \frac{e \xrightarrow{\boldsymbol{r}} e' \quad e' \xrightarrow{\boldsymbol{s}} e''}{e \xrightarrow{\boldsymbol{r}::\boldsymbol{s}} e''}\;(\mathrm{trans})$$

Figure 2: Operational semantics $e \xrightarrow{\boldsymbol{r}} e'$ of the λ-calculus with regions

Example 1 *We explain the key intuition rendered by the rule (free) by considering a very simple expression that creates a dangling pointer. We write* $\mathsf{let}\, x = e\, \mathsf{in}\, e'$ *for* $(\mathsf{fun}\, x.e')\, e$.

$$e = (\mathsf{let}\, f = \overbrace{\mathsf{fun}\; x.x\;\mathsf{at}\; r}^{e_1}\, \mathsf{in}\, \mathsf{let}\, g = \overbrace{\mathsf{fun}\; y.f(y)\;\mathsf{at}\; s}^{e_2}\, \mathsf{in}\, \overbrace{g(f)}^{e_3})/s$$

The region r is free in e. The region s is bound by the right-most restriction sign. In call-by-value evaluation order, the expression first defines a new region named s, evaluates the sub-expression e_1, putting the identity λ-abstraction $\mathsf{fun}\; x.x$ *at the region r. Then, having substituted f by its value* $(\mathsf{fun}\; x.x)_r$*, the expression* $\mathsf{let}\, g = e_2\, \mathsf{in}\, e_3$ *is evaluated. This is handled by first allocating the result* $\mathsf{fun}\; y.(\mathsf{fun}\; x.x)_r(y)$ *of the sub-expression e_2 at the region s and then by substituting g by that value in the remaining sub-expression e_3, yielding* $(\mathsf{fun}\; x.x)_r$*. To pass the restriction sign, however, the region s must be deallocated. This is rendered by substituting s by the region constant* free *that denotes the free space.*

The result of the expression is $(\mathsf{fun}\ x.x)_r$.

$$
\begin{array}{rcll}
e & = & (\mathsf{let}\ f = \mathsf{fun}\ x.x\ \mathsf{at}\ r\ \mathsf{in}\ \mathsf{let}\ g = \mathsf{fun}\ y.f(y)\ \mathsf{at}\ s\ \mathsf{in}\ g(f)) & /s \\
& \xrightarrow{r} & (\mathsf{let}\ f = [(\mathsf{fun}\ x.x)_r]\ \mathsf{in}\ \mathsf{let}\ g = \mathsf{fun}\ y.f(y)\ \mathsf{at}\ s\ \mathsf{in}\ g(f)) & /s \quad \textit{by (abs)} \\
& = & \mathsf{let}\ g = \mathsf{fun}\ y.(\mathsf{fun}\ x.x)_r(y)\ \mathsf{at}\ s\ \mathsf{in}\ g((\mathsf{fun}\ x.x)_r)) & /s \quad \textit{by (let)} \\
& \rightarrow & \mathsf{let}\ g = [(\mathsf{fun}\ y.(\mathsf{fun}\ x.x)_r(y))_s]\ \mathsf{in}\ g((\mathsf{fun}\ x.x)_r)) & /s \quad \textit{by (abs)} \\
& = & (\mathsf{fun}\ y.(\mathsf{fun}\ x.x)_r(y))_s\,(\mathsf{fun}\ x.x)_r) & /s \quad \textit{by (let)} \\
& \rightarrow & [(\mathsf{fun}\ x.x)_r\,(\mathsf{fun}\ x.x)_r] & /s \quad \textit{by (app)} \\
& \xrightarrow{r} & [(\mathsf{fun}\ x.x)_r] & /s \quad \textit{by (app)} \\
& \rightarrow & [(\mathsf{fun}\ x.x)_r] & \quad \textit{by (free)}
\end{array}
$$

Had we restricted the scope of the region r to, e.g., e/r, we would have obtained the "dangling" object $(\mathsf{fun}\ x.x)_{\mathsf{free}}$. *By definition of the rule (app), any application of this object would not satisfy the side-condition $r \neq \mathsf{free}$, yielding an error, by definition of the relation $\not\rightarrow$. In the remainder, we shall therefore regard the expression e/r as ill-typed.*

$$
\begin{array}{rcll}
\forall v,\ [e/r]\,v & \rightarrow & [(\mathsf{fun}\ x.x)_{\mathsf{free}}]\,v & \textit{by (free)} \\
& \not\rightarrow & & \textit{by (app)}
\end{array}
$$

3 Type-checking regions

We express region inference in two steps. One, presented next, is an inductive proof system that defines the type-checkable programs of the region calculus. It takes the form of a judgment $\Gamma \vdash e : \epsilon$ which is defined by induction on the syntax of expressions e. The judgment $\Gamma \vdash e : \epsilon$ means that the expression e has a type and an effect denoted by ϵ under the hypothesis represented by Γ. The second step, presented in the section 4, is an algorithm which translates (or compiles) source expressions $\boldsymbol{e}$ of the λ-calculus into type-checkable programs e of the region calculus.

3.1 Type system

In the region calculus, the type of an expression is denoted by a term ϵ of the form $\tau^{\boldsymbol{\rho}}$. It consists of the type τ of the datum returned by the expression and of the store effect $\boldsymbol{\rho}$ performed to evaluate the expression.

A data type τ is either a variable α for a literal x, an arrow type $\tau' \rightarrow \tau^{\varrho,\boldsymbol{\rho}}$ for a λ-abstraction (where τ' stands for the type of the formal parameter, τ for that of the result and $\varrho, \boldsymbol{\rho}$ the effect of the function) or a pair τ_r for an object (i.e. a value of type τ allocated in the region r).

An effect $\boldsymbol{\rho}$ is represented by an extensible record in a way similar to that of Remy [16] and Leroy [12]. It consists of regions r and of extension variables ϱ which are assembled using an extension operator $\boldsymbol{\rho}, \boldsymbol{\rho}'$. The occurrence of a

region r in the effect of an expression means that the expression which has that effect may define or use a value in the region r. The occurrence of an extension variable ϱ in an effect $\boldsymbol{\rho}$ identifies a (group of) functions that may be called within that expression.

We identify $(\tau^{\boldsymbol{\rho}})^{\boldsymbol{\rho}'} = \tau^{\boldsymbol{\rho},\boldsymbol{\rho}'}$ and $\tau = \tau^{\emptyset}$. We identify records $\boldsymbol{\rho}$ under neutrality [rule (u)], idempotency [rule (i)], associativity [rule (a)] and right-commutativity [rule (c)]. We write $\boldsymbol{\rho}\backslash r$ for the record $\boldsymbol{\rho}$ without the region r.

$$\begin{array}{rcll} (\boldsymbol{\rho},\boldsymbol{\rho}'),\boldsymbol{\rho}'' & = & (\boldsymbol{\rho},\boldsymbol{\rho}'),\boldsymbol{\rho}'' & \text{(a)} \\ \varrho,(\boldsymbol{\rho},\boldsymbol{\rho}') & = & \varrho,(\boldsymbol{\rho}',\boldsymbol{\rho}) & \text{(c)} \\ \boldsymbol{\rho},\emptyset & = & \boldsymbol{\rho} & \text{(u)} \\ \boldsymbol{\rho},\boldsymbol{\rho} & = & \boldsymbol{\rho} & \text{(i)} \end{array}$$

Types are associated to names in an environment Γ and are potentially quantified over free regions ($\forall r.\tau$) and extension variables ($\forall\varrho.\tau$). We write $\boldsymbol{\varrho}$ for a sequence of extension variables. Again, we identify τ with $\forall[].\tau$ and $\forall r :: s.\tau$ with $\forall s.\forall r.\tau$ and $\forall r.\forall s.\tau$. We write $\mathrm{fr}(\_)$ and $\mathrm{fv}(\_)$ (resp. $\mathrm{br}(\_)$ and $\mathrm{bv}(\_)$) for the sets of free (resp. bound) regions and extension variables of a term. By definition, we assume the region constant free to be referenced in any context Γ (i.e. that we have $\mathsf{free} \in \mathrm{fr}(\Gamma), \forall\Gamma$).

We write σ for a substitution and $(\_)\sigma$ for its application to a term. The term $[]$ denotes the identity. We write $[\tau/\alpha]$ a substitution of α by τ, $[r/r']$ of r' by r and $[(\varrho,\boldsymbol{\rho})/\varrho']$ of ϱ' by the (larger) effect $\varrho,\boldsymbol{\rho}$. The functional composition of substitutions is noted $\sigma\circ\sigma'$. We write $\mathrm{dom}\,\sigma$ for the domain of a substitution which is defined as the set of terms which are not identical to their image by substitution.

$$\begin{array}{rcll} \boldsymbol{\rho} & ::= & \emptyset \mid r \mid \varrho \mid \boldsymbol{\rho},\boldsymbol{\rho}' & \text{(record of effects)} \\ \Gamma & ::= & [] \mid \Gamma,[x:\forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau] & \text{(type environment)} \\ \tau & ::= & \alpha \mid \tau' \to \tau^{\varrho,\boldsymbol{\rho}} \mid \tau_r \qquad \epsilon ::= \tau^{\boldsymbol{\rho}} & \text{(type and effect)} \\ \sigma & ::= & [] \mid [\tau/\alpha] \mid [r/r'] \mid [(\varrho,\boldsymbol{\rho})/\varrho'] \mid \sigma\circ\sigma' & \text{(substitution)} \end{array}$$

Figure 3: Type system

3.2 Inference system

The figure 4 defines the judgment $\Gamma \vdash e : \epsilon$ of the static semantics by induction on the syntax e of expression. The type of an expression is checked much as in the polymorphic λ-calculus [14]. The effect of an expression is a record of all the regions at which it allocates data [e.g. rule (ABS)] and all the regions an record extensions from which λ-abstraction are invoked [e.g. rule (APP)].

The rule (REC) allows to type-check the parameterization of regions in for recursive function definitions by determining the corresponding polymorphic type of the function. Conversely, The rule (VAR) to instantiate those parameters by determining the matching type instantiation.

The rule (OBS) checks the locality of the region r to the scope of the restriction sign e/r by the side-condition $r \notin \mathrm{fr}(\Gamma) \cup \mathrm{fr}(\tau)$. The term e/r of example 1 is an instance of expression that is not type-checkable by the rule (OBS), because it attempts to restrict the scope of the region r of its result, of type $(\alpha \to \alpha^{\varrho})_r$.

The rule (OBS') allows to remove superfluous extension variables from the effect of an expression. Finally, the rule (SUB) of subsumption tells that, if e has type ϵ then it admits any bigger effect $\epsilon^{\boldsymbol{\rho}}$.

$$\frac{\Gamma, [x:\tau'] \vdash e:\tau^{\boldsymbol{\rho}}}{\Gamma \vdash \mathsf{fun}\ x.e\ \mathsf{at}\ r:((\tau' \to \tau^{\varrho,\boldsymbol{\rho}})_r)^r}\ (\mathrm{ABS})_{x\notin \mathrm{dom}\,\Gamma}$$

$$\frac{\Gamma \vdash e:((\tau \to \epsilon)_r)^{\boldsymbol{\rho}} \quad \Gamma \vdash e':\tau^{\boldsymbol{\rho}'}}{\Gamma \vdash e\,e':\epsilon^{\boldsymbol{\rho},\boldsymbol{\rho}',r}}\ (\mathrm{APP})$$

$$\frac{\Gamma, [x:\forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau] \vdash e:\tau^r}{\Gamma \vdash \mathsf{rec}\ x[\boldsymbol{r}]e:\tau^r}\ (\mathrm{REC})\ \begin{array}{l}\boldsymbol{r}=\mathrm{fr}(\tau)\backslash\mathrm{fr}(\Gamma)\cup\{r\}\\ \boldsymbol{\varrho}=\mathrm{fv}(\tau)\backslash\mathrm{fv}(\Gamma)\\ x\notin\mathrm{dom}\,\Gamma\end{array}$$

$$\Gamma, [x:\forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau] \vdash x\ \mathsf{at}\ (\boldsymbol{r}\sigma):\tau\sigma\ (\mathrm{VAR})_{\mathrm{dom}\,\sigma=\mathrm{fr}(\boldsymbol{r})\cup\mathrm{fv}(\boldsymbol{\varrho})}$$

$$\frac{\Gamma \vdash e:\epsilon^r}{\Gamma \vdash e/r:\epsilon}\ (\mathrm{OBS})_{r\notin\mathrm{fr}(\Gamma)\cup\mathrm{fr}(\epsilon)} \qquad \frac{\Gamma \vdash e:\epsilon^{\varrho}}{\Gamma \vdash e:\epsilon}\ (\mathrm{OBS'})_{\varrho\notin\mathrm{fv}(\Gamma)\cup\mathrm{fv}(\epsilon)} \qquad \frac{\Gamma \vdash e:\epsilon}{\Gamma \vdash e:\epsilon^{\boldsymbol{\rho}}}\ (\mathrm{SUB})$$

Figure 4: Region inference system $\Gamma \vdash e:\epsilon$

Example 2 *The factorial function is the simplest example to convince oneself on the importance of incorporating polymorphic recursion in the effect system of the region calculus (assuming it now understands operations on integers as described in [20]). First, the λ-expression that defines the factorial is as follows.*

$$\mathsf{rec}\ \mathsf{fac.fun}\ n.\mathsf{if}\ (n<2)\ \mathsf{then}\ 1\ \mathsf{else}\ n \times \mathsf{fac}(n-1)$$

Second, a monomorphic rule for typing recursive function would consider the hypothesis $\Gamma, [x:\tau] \vdash e:\tau^r$ to conclude $\Gamma \vdash \mathsf{rec}\,x[\boldsymbol{r}]e:\tau^r$ instead of the rule (rec).

Now, assume the type $\forall \varrho :: r :: r' :: r''.\mathsf{int}_r \times \mathsf{int}_{r'} \to (\mathsf{int}_{r''})^{\varrho.r.r':r''}$ *for binary operators on integers. A region assignment of the factorial function with a monomorphic recursion rule cannot perform any better than giving the monomorphic type* $\mathsf{int}_{r_a} \to \mathsf{int}_{r_b}^{\varrho.r_a.r_b}$.

$$\begin{array}{l}\mathsf{rec\ fac.fun}\ n.(\ \mathsf{if}\ (n\ (<\ \mathsf{at}\ (r_a :: r_1 :: r_2))\ (2\ \mathsf{at}\ r_1))/r_1\ \mathsf{then}\ 1\ \mathsf{at}\ r_b \\ \qquad\qquad \mathsf{else}\ n\ (\times\ \mathsf{at}\ (r_a :: r_b :: r_b))\ \mathsf{fac}(n\ (-\ \mathsf{at}\ (r_a :: r_3 :: r_a))\ (1\ \mathsf{at}\ r_3))/r_3)/r_2\end{array}$$

This means that, during recursion, all intermediate results are put in region r_a *and all intermediate arguments in region* r_b.

Polymorphic recursion allows a much more efficient region assignment by considering the polymorphic type $\forall r_a, r_b.\forall \varrho.\mathsf{int}_{r_a} \to (\mathsf{int}_{r_b})^{\varrho.r_a.r_b}$, *which allows to lexically isolate all the intermediate arguments and results of each recursive call into singleton regions* r_4 *and* r_5.

$$\begin{array}{l}\mathsf{rec\ fac}[r_a, r_b]\mathsf{fun}\ n. \\ \qquad (\mathsf{if}\ \ (n\ (<\ \mathsf{at}\ (r_a :: r_1 :: r_2))\ (2\ \mathsf{at}\ r_1))/r_1\ \mathsf{then}\ 1\ \mathsf{at}\ r_b \\ \qquad\quad \mathsf{else}\ (n\ \ (\times\ \mathsf{at}\ (r_a :: r_5 :: r_b)) \\ \qquad\qquad\qquad ((\mathsf{fac\ at}\ (r_4, r_5))(n\ (-\ \mathsf{at}\ (r_a :: r_3 :: r_5))\ (1\ \mathsf{at}\ r_3))/r_3)/r_4)/r_5)/r_2\end{array}$$

3.3 Type soundness

We formulate type soundness by considering the syntactic proof method proposed by Wright and Feleisen in [22], extended to the region calculus. Just as Wright and Feleisen, we consider an inference system (i.e. that of figure 4) extended with the rule (ABS-VAL) and (REC-VAL) for type-checking evaluated and inlined objects. We write $|\boldsymbol{r}|$ for the length of the sequence $\boldsymbol{r}$.

$$\frac{\Gamma \vdash \mathsf{fun}\ x.e\ \mathsf{at}\ r : \tau^r}{\Gamma \vdash (\mathsf{fun}\ x.e)_r : \tau}\ (\text{ABS-VAL}) \qquad \frac{\Gamma \vdash \mathsf{rec}\ x[\boldsymbol{r}]v : \tau}{\Gamma \vdash (\mathsf{rec}\ x[\boldsymbol{r}]v)\ \mathsf{at}\ \boldsymbol{s} : \tau[\boldsymbol{s}/\boldsymbol{r}]}\ (\text{REC-VAL})\ \begin{array}{l}\boldsymbol{r} \notin \mathrm{fr}(\Gamma) \\ |\boldsymbol{r}| = |\boldsymbol{s}|\end{array}$$

Figure 5: Type of values

In contrast to other techniques that have been considered to prove the correctness of the region calculus so far, the syntactic soundness method does not require to establish an inductive or co-inductive semantical relation between types and stores (or other co-inductively defined recursive objects). This is exemplified by the statement of the subject reduction property (lemma 1).

The subject reduction lemma stipulates that if both a transition $e \xrightarrow{\boldsymbol{r}} e'$ from e to e' is possible and if e has type ϵ given the hypothesis Γ, then e' has a type ϵ' of smaller effect that ϵ and ϵ contains the regions $\boldsymbol{r}$ used during the transition (the overloaded notation $\epsilon^{\boldsymbol{r}}$ is defined by $\epsilon^{[]} = \epsilon$ and $\epsilon^{r.\boldsymbol{s}} = (\epsilon^r)^{\boldsymbol{s}}$).

Lemma 1 (reduction) *If* $e \xrightarrow{r} e'$ *and* $\Gamma \vdash e : \epsilon$ *then* $\Gamma \vdash e' : \epsilon$ *and* $\epsilon = \epsilon^{r}$.

The structure of the proof for lemma 1 is standard (see appendix). It mainly makes use of sub-lemmas on replacement and on substitution which are stated next and proved in appendix.

Lemma 2 (weakening) *If* $\Gamma \vdash e : \epsilon$ *and* $x \notin \mathrm{dom}\,\Gamma$ *then* $\Gamma, [x : \tau] \vdash e : \epsilon$

Lemma 3 (replacement) *If*

1. $\Gamma \vdash c[e] : \epsilon$
2. $\Gamma, \Gamma' \vdash e : \epsilon'$
3. $\Gamma, \Gamma' \vdash e' : \epsilon'$
4. $(\epsilon')^{\boldsymbol{r}} = \epsilon'$ *and* $\boldsymbol{r} \notin \mathrm{br}(c)$

then $\Gamma \vdash c[e'] : \epsilon$ *and* $\epsilon^{\boldsymbol{r}} = \epsilon$

Lemma 4 (substitution) *If* $\Gamma \vdash e : \epsilon$ *then* $\Gamma\sigma \vdash e\sigma : \epsilon\sigma$

Lemma 5 (syntactic substitution) *If*

1. $\Gamma, [x : \forall \boldsymbol{r}. \forall \boldsymbol{\varrho}. \tau] \vdash e : \epsilon$
2. $\boldsymbol{r} \notin \mathrm{fr}(\Gamma)$
3. $\boldsymbol{\varrho} \notin \mathrm{fv}(\Gamma)$
4. $\Gamma \vdash v : \tau$

then $\Gamma \vdash e[v/x] : \epsilon$.

The main result is build up from the subject reduction property and is stated next. We consider closed expressions i.e. programs e s.t. $\mathrm{fv}(e) = \emptyset$ and $\mathrm{fr}(e) = \emptyset$. Since a program has, by convention, no free region, it may only return the result unit, of type $\Gamma \vdash \mathsf{unit} : \mathsf{unit}$ for all Γ and no region.

Theorem 1 (soundness) *If* $\vdash e : \mathsf{unit}$ *then either* $e \Uparrow$ *or* $e \twoheadrightarrow \mathsf{unit}$

The overall structure of the proof for the syntactic soundness property is standard. It is built by using auxiliary lemmas which discriminate faulty and non-faulty expressions.

Lemma 6 (uniform evaluation) *For a closed* e, *either* $e \twoheadrightarrow v$, $e \Uparrow$ *or* $e \not\twoheadrightarrow$

Lemma 7 (faulty expression) *For a closed* e, *if* $e \not\twoheadrightarrow$ *then* $\neg(e : \epsilon)$

For the structure of the proofs of lemmas 6 and 7 see [22]. The adaptation to the region is straightforward, thanks to the definition of faults in section 2.2; the lemma 7 considers both type-errors (e.g. to take a number for a function) and expressions which attempt access dangling pointers. Note that expressions are closed i.e. s.t. $\mathrm{fv}(e) = \emptyset$. Hence, in particular, $\mathsf{free} \notin \mathrm{fv}(e)$.

4 A simple inference algorithm

The second part of our presentation consists of the definition of a simple inference algorithm for translating source expressions of the λ-calculus into type-checkable programs of the region calculus.

The presentation proceeds in two steps. The first step consists of formulating the simplest algorithmic presentation of type and effect inference. The second step consists of a seamless extension of this algorithm to deal with the issue of typing region-polymorphic recursive functions (section 4.2).

The first step of our presentation is to state a simple algorithm for inferring regions for the call-by-value λ-calculus (grammar $\boldsymbol{e}$), but without recursion.

$$\boldsymbol{v} ::= x \mid \mathsf{fun}\, x.\boldsymbol{e} \qquad \boldsymbol{e} ::= \boldsymbol{v} \mid \boldsymbol{e}\,\boldsymbol{e}' \mid \mathsf{rec}\, x.\boldsymbol{v} \qquad (\lambda\text{-expression})$$

Figure 6: Source λ-expressions

The algorithm and its soundness proof are mainly an adaptation of that introduced in [17] and [18, pp. 113-117] to type references in ML. It consists of:

- the inference function $\mathcal{I}$, which inductively translates each sub-terms of a λ-expression $\boldsymbol{e}$ in the region calculus
- the unification function $\mathcal{U}$, which returns the most general unifier that corresponds to each type equation encountered in the program.

The unification procedure expects a pair $[\![\tau, \tau']\!]$ of types and returns the most-general unifier σ that satisfies the equation $\tau = \tau'$. If that equation cannot be satisfied by a finite type term (e.g. for $\alpha = (\alpha \to \alpha^{\varrho})_r$), then unification fails.

$$\begin{array}{lll}
\mathcal{U}[\![\tau, \tau']\!] = \ \mathsf{case}\ (\tau, \tau')\ \mathsf{of} & & \\
\qquad (\alpha, \tau) \mid (\tau, \alpha) & \Rightarrow & \mathsf{if}\ (\alpha \notin \mathrm{fv}(\tau))\ \mathsf{then}\ [\tau/\alpha]\ \mathsf{else}\ \mathsf{fail} \\
\qquad (\tau \to \epsilon, \tau' \to \epsilon') & \Rightarrow & \mathsf{let}\ \sigma = \mathcal{U}[\![\tau, \tau']\!]\ \mathsf{in}\ \sigma \circ \mathcal{U}[\![\epsilon\sigma, \epsilon'\sigma]\!] \\
\qquad (\tau_r, \tau'_{r'}) & \Rightarrow & \mathsf{let}\ \sigma = [r'/r]\ \mathsf{in}\ \sigma \circ \mathcal{U}[\![\tau\sigma, \tau'\sigma]\!] \\
\qquad (\tau^{\varrho,\boldsymbol{\rho}}, \tau'^{\varrho',\boldsymbol{\rho}'}) & \Rightarrow & \mathsf{let}\ \sigma = [\varrho, \boldsymbol{\rho}, \boldsymbol{\rho}'/\varrho] \circ [\varrho/\varrho']\ \mathsf{in}\ \sigma \circ \mathcal{U}[\![\tau\sigma, \tau'\sigma]\!] \\
\qquad \langle _, _ \rangle & \Rightarrow & \mathsf{fail}
\end{array}$$

Figure 7: Unification algorithm $\mathcal{U}[\![\epsilon, \epsilon']\!] = \sigma$

The inference procedure is implemented by a pair of mutually recursive functions. The function $\mathcal{I}'$ is defined by induction on the syntax of expression. To

a pair $[\![\Gamma, \boldsymbol{e}]\!]$ consisting of a type environment and of a source expression, it associates a triple $[\![e:\epsilon, \sigma]\!]$ which consists of a target expression e of the region calculus, its type and effect ϵ and a substitution σ that affects the free variables and regions of the environment Γ. The main function $\mathcal{I}$ filters local regions $\boldsymbol{r}$ and extension variables $\boldsymbol{\varrho}$ from the effect computed by the function $\mathcal{I}'$ and introduces restriction signs $e/\boldsymbol{r}$ as required.

$$
\begin{array}{rcl}
\mathcal{I}[\![\Gamma, \boldsymbol{e}]\!] = \mathsf{let}\ \langle \tau^{\boldsymbol{\rho}}, \sigma\rangle & = & \mathcal{I}'[\![\Gamma, \boldsymbol{e}]\!] \\
\boldsymbol{\varrho} & = & (\mathrm{fv}(\boldsymbol{\rho}) \setminus \mathrm{fv}(\Gamma)) \setminus \mathrm{fv}(\tau) \\
\boldsymbol{r} & = & (\mathrm{fr}(\boldsymbol{\rho}) \setminus \mathrm{fr}(\Gamma)) \setminus \mathrm{fr}(\tau) \\
\mathsf{in}\ \langle e/\boldsymbol{r} : \tau^{(\boldsymbol{\rho}\setminus\boldsymbol{\varrho})\setminus\boldsymbol{r}}, \sigma\rangle & & \\
\mathsf{and}\ \mathcal{I}'[\![\Gamma, \boldsymbol{e}]\!] = \mathsf{case}\ \boldsymbol{e}\ \mathsf{of} & & \\
x & \Rightarrow & \mathsf{if}\ ([\mathsf{x}:\tau] \in \Gamma)\ \mathsf{then}\ \langle \mathsf{x}:\tau, []\rangle\ \mathsf{else}\ \mathsf{fail} \\
\boldsymbol{e}\ \boldsymbol{e}' & \Rightarrow & \mathsf{let}\ \alpha, \varrho, r\ \mathrm{fresh} \\
 & & \langle e:\tau^{\boldsymbol{\rho}}, \sigma\rangle = \mathcal{I}[\![\Gamma, \boldsymbol{e}]\!] \\
 & & \langle e':\tau'^{\boldsymbol{\rho}'}, \sigma'\rangle = \mathcal{I}[\![\Gamma\sigma, \boldsymbol{e}']\!] \\
 & & \sigma'' = \mathcal{U}[\![\tau\sigma', (\tau' \to (\alpha^{\varrho}))_r]\!] \\
 & & \mathsf{in}\ \langle (e\sigma'\ e')\sigma'' : (\alpha\sigma'')^{((\boldsymbol{\rho}\sigma'), \boldsymbol{\rho}', \varrho, r)\sigma''}, \sigma\circ\sigma'\circ\sigma''\rangle \\
\mathsf{fun}\ x.\boldsymbol{e} & \Rightarrow & \mathsf{let}\ \alpha, r, \varrho\ \mathrm{fresh} \\
 & & \langle e:\tau^{\boldsymbol{\rho}}, \sigma\rangle = \mathcal{I}[\![\Gamma[x:\alpha], \boldsymbol{e}]\!] \\
 & & \mathsf{in}\ \langle (\mathsf{fun}\ x.e)\ \mathsf{at}\ r : ((\alpha\sigma \to \epsilon_{\varrho,r,\boldsymbol{\rho}})_r)^r, \sigma\rangle
\end{array}
$$

Figure 8: Inference algorithm $\mathcal{I}[\![\Gamma, \boldsymbol{e}]\!] = [\![e:\epsilon, \sigma]\!]$

The main property that will be required for the remainder of the presentation is the soundness of the inference algorithm $\mathcal{I}$ for the λ-expressions $\boldsymbol{e}$ without recursion.

Theorem 2 (soundness of $\mathcal{I}$) *If $\mathcal{I}[\![\Gamma, \boldsymbol{e}]\!] = \langle e:\epsilon, \sigma\rangle$ (figure 8) then $\Gamma\sigma \vdash e:\epsilon$*

The proof (see appendix) is by induction on the structure of expressions. It makes use of the type substitution lemma 4 and of the lemma 8 on the soundness of the unification procedure. The proof of property 8 (see appendix) is by induction on the structure of the pair τ, τ'.

Lemma 8 (Soundness of $\mathcal{U}$) *if $\mathcal{U}[\![\tau, \tau']\!] = \sigma$ then $\tau\sigma = \tau'\sigma$*

4.1 Labelling regions

We are now to consider the extension of our inference algorithm $\mathcal{I}$ with region-polymorphic recursive functions. In other words, we aim at defining a simple

extension of the inference algorithm $\mathcal{I}$ that can produce a competitive assignment of regions to a λ-expression with recursion.

Until now, the problem of inferring a region-polymorphic assignment of recursive functions has been addressed in terms of the semi-unification algorithm of [10] by [3].

We present an alternative algorithm, which we believe to be simpler, that is constructed starting from the procedure $\mathcal{I}$ of the previous section and by using a simple fixed-point iteration loop that finitely unfolds the type of recursive functions.

Example 3 *It is easy to understand that a naive iteration, starting from an over-approximation of the expected fixed-point, would fail to always terminate. Consider for instance a recursive function f that builds and returns a λ-abstraction that captures f before returning its formal parameter x (e.g.:* $\mathsf{rec}\, f.\mathsf{fun}\, x.(\mathsf{fun}\, y.(f\, y))\, x$*, and/or its variants). Each step of a brute-force iteration would introduce new instances of the extension variables corresponding to the effect of f.*

In order to ensure termination of the fixed-point iteration, we just need to device a mean for restricting the number of regions and extension variables to be introduced during the each iteration, and then to "widen" the search for a fixed-point by unifying the types obtained at the successive iteration steps. The control of the number of fresh regions and extension variables that can be introduced by the inference algorithm for type-checking a recursive function $\mathsf{rec}\, f.\boldsymbol{e}$ is implemented by relating it to the number of references of the recursive function name f in $\boldsymbol{e}$.

$\boldsymbol{l} ::= [] \mid l, \boldsymbol{l}$	(label)
$\boldsymbol{\rho} ::= \emptyset \mid r_{\boldsymbol{l}} \mid \varrho_{\boldsymbol{l}} \mid \boldsymbol{\rho} * \boldsymbol{\rho}'$	(labelled record)
$\sigma ::= \ldots \mid [r_{\boldsymbol{l}}/r] \mid [\varrho_{\boldsymbol{l}}/\varrho]$	(substitution)

Figure 9: Labels of regions

Each reference is first uniquely related to a label l. Then, every time a region r or an extension variable is introduced at the label l, that label is recorded together with the region name l. This forms a structure $r_{\boldsymbol{l}}$ that consists of the region name and a (local) record of the label names $\boldsymbol{l}$ (grammars $\boldsymbol{l}$ and $\boldsymbol{\rho}$ of figure 9). Further attempts to instantiate the same region r, at the same program label l, will yield that same region $r_{\boldsymbol{l}}$ since $l \in \boldsymbol{l}$.

The figure 10 details the instantiation procedure for the region polymorphic types of recursive functions. We identify $r_{[]}$ to r, $\varrho_{[]}$ to ϱ and $(r_{\boldsymbol{l}})_{\boldsymbol{l}'}$ to $r_{\boldsymbol{l},\boldsymbol{l}'}$.

$$\mathrm{inst}_l[\![\forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau]\!] = \langle \tau, \sigma \rangle \text{ where } \sigma : \begin{array}{rcl} \forall r_l \in \boldsymbol{r} & \mapsto & \begin{cases} s_{l,l} & \text{, if } (l \notin \boldsymbol{l}) \text{ and } s \text{ fresh} \\ r_l & \text{, if } (l \in \boldsymbol{l}) \end{cases} \\ \forall \varrho_l \in \boldsymbol{\varrho} & \mapsto & \begin{cases} \varrho'_{l,l} & \text{, if } (l \notin \boldsymbol{l}) \text{ and } \varrho' \text{ fresh} \\ \varrho_l & \text{, if } (l \in \boldsymbol{l}) \end{cases} \end{array}$$

Figure 10: Instantiation of labelled regions

4.2 A fixed-point theory

To validate the simple trick depicted in the previous section, we instantiate the fixed-point theory presented in [5] to the case of the chain constructed by our fixed-point algorithm, which is presented next.

Definition 1 (fixed-point iterator) *For any environment Γ and λ-abstraction $\boldsymbol{v}$, we note $\mathcal{F}_{\Gamma,\boldsymbol{v}}$ the fixed-point iterator defined by*

$$\begin{array}{rcl} \mathcal{F}_{\Gamma,\boldsymbol{v}}[\![f : \forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau, \boldsymbol{v} : \tau, \sigma]\!] = \mathsf{let}\ \langle v' \,\mathsf{at}\, r : (\tau')^r, \sigma'' \rangle & = & \mathcal{I}'[\![\Gamma\sigma, [f : \forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau], \boldsymbol{v}]\!] \\ \sigma' & = & \sigma \circ \sigma'' \circ \mathcal{U}[\![\tau\sigma'', \tau']\!] \\ \boldsymbol{r}' & = & (\mathrm{fr}(\tau'\sigma') \backslash \mathrm{fr}(\Gamma\sigma')) \backslash r\sigma' \\ \boldsymbol{\varrho}' & = & \mathrm{fv}(\tau'\sigma') \backslash \mathrm{fv}(\Gamma\sigma') \\ \mathsf{in}\ \langle f : \forall \boldsymbol{r}'.\forall \boldsymbol{\varrho}'.\tau'\sigma', v'\sigma' : \tau'\sigma', \sigma' \rangle & & \end{array}$$

We write $\mathcal{C}_{\Gamma,\boldsymbol{v}}$ the chain defined using $\mathcal{F}_{\Gamma,\boldsymbol{v}}$ by

$$\mathcal{C}_{\Gamma,\boldsymbol{v}} = (\mathcal{C}^n)_{n\geq 0} \text{ where } \mathcal{C}^0 = \langle f : \forall \boldsymbol{r}_0.\forall \boldsymbol{\varrho}_0.\tau_0, \boldsymbol{v} : \tau_0, [] \rangle \text{ and } \forall n \geq 0,\ \mathcal{C}^{n+1} = \mathcal{F}_{\Gamma,\boldsymbol{v}}[\![\mathcal{C}^n]\!]$$

Finally, we write $\langle f : \forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau, \boldsymbol{v} : \tau, \sigma \rangle \sqsubseteq_{\boldsymbol{v}} \langle \forall \boldsymbol{r}'.\forall \boldsymbol{\varrho}'.\tau', v' : \tau', \sigma' \rangle$ the partial order defined iff there exists σ'' such that $\sigma' = \sigma \circ \sigma''$, $\tau' = (\tau\sigma')\sigma''$, $\boldsymbol{r}' \supseteq (\boldsymbol{r}\sigma')\sigma''$ and $\boldsymbol{\varrho}' \supseteq (\boldsymbol{\varrho}\sigma')\sigma''$.

Note that $\mathcal{F}$ is extensive for the partial order relation $\sqsubseteq$ and that the number of regions introduced by $\mathcal{F}$ is correlated to the number of f in $\boldsymbol{v}$. Therefore, given that each iteration introduces finitely many regions variables, the strictly increasing chain $\mathcal{C}$ is finite. By application of the, yielding a fixed-point of $\mathcal{F}$.

Lemma 9 (termination) *For all finite environments Γ and λ-abstractions v, the iteration $\mathcal{F}_{\Gamma,\boldsymbol{v}}$ starting from a finite $\mathcal{C}^0_{\Gamma,\boldsymbol{v}}$ either fails by $\mathcal{U}$ or terminates after a finite number $n < \omega$ of iterations, yielding a fixed-point $\mathcal{C}^n_{\Gamma,\boldsymbol{v}} = \mathcal{F}_{\Gamma,\boldsymbol{v}}\,\mathcal{C}^n_{\Gamma,\boldsymbol{v}}$.*

Example 4 *Reconsider the factorial function of example 2. It has only one recursive call labelled l. The polymorphic type determined in the example 2 can be computed in two steps of iteration.*

$$\mathsf{rec}\ \mathsf{fac.fun}\ n.\mathsf{if}\ (n < 2)\ \mathsf{then}\ 1\ \mathsf{else}\ n \times \mathsf{fac}_l(n-1)$$

At the first iteration, the regions and extension variables of fac *introduced at the label l are polymorphic (fresh names are taken). At the second iteration, they become monomorphic and are instantiated by themselves.*

$$\begin{array}{lcll}
\tau_0 & = & \mathsf{int}_r \to (\mathsf{int}_{r'})^{\varrho} & \text{, initial assignment with fresh } r, r', \varrho \\
\tau_1 & = & \mathsf{int}_{r_l} \to (\mathsf{int}_{r'_l})^{\varrho_l . r_l . r'_l} & \text{, first iteration and instantiation at } l \\
\tau_2 & = & \mathsf{int}_{r_l} \to (\mathsf{int}_{r'_l})^{\varrho_l . r_l . r'_l} & \text{, fixed-point } \tau_1 = \tau_2
\end{array}$$

4.3 Typing recursive functions

Assume that, for a recursive definition $\mathsf{rec}\ f.\boldsymbol{v}$, every occurrence of f in $\boldsymbol{v}$ is now temporarily labelled as f_l by a distinct l. Using the fixed-point iteration mechanism of section 4.2, determining the type of a recursive function $\mathsf{rec}\ f.\boldsymbol{v}$ just amounts to choosing the appropriate initial type assignment τ before starting the iteration. This is the operation that is specified in the figure 11.

It consists of first determining the "shape" or data-type $\tau_0\sigma'_0$ of the recursive function to be type-checked. This is done by type-checking $\boldsymbol{v}$ with the assignment $[f:\alpha]$ and then by unifying the result τ_0 to the initial α with σ'_0.

$$\begin{array}{lcll}
\mathcal{I}'[\![\Gamma, \boldsymbol{e}]\!] = \mathsf{case}\ \boldsymbol{e}\ \mathsf{of} & & & \\
\qquad\qquad \vdots & & & \\
\qquad \mathsf{rec}\ f.\boldsymbol{v} & \Rightarrow & \mathsf{let}\ \langle f:\forall \boldsymbol{r}'.\forall \boldsymbol{\varrho}'.\tau', v':\tau', \sigma\rangle = \mathrm{lfp}\, \mathcal{F}_{\Gamma,\boldsymbol{v}}[\![f:\forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau, \boldsymbol{v}:\tau, []]\!] & \\
& & \qquad\qquad\qquad\qquad\qquad\qquad \alpha\ \text{fresh} & \\
& & \qquad\qquad\qquad\qquad\qquad \langle v_0:\tau_0, \sigma_0\rangle = \mathcal{I}'[\![\Gamma[f:\alpha], \boldsymbol{v}]\!] & \\
& & \qquad\qquad\qquad\qquad\qquad\qquad\ \sigma'_0 = \mathcal{U}[\![\alpha\sigma_0, \tau_0]\!] & \\
& & \qquad\qquad\qquad\qquad\qquad\qquad\ \tau = |\tau_0\sigma'_0| & \\
& & \qquad\qquad\qquad\qquad\qquad\qquad\ \boldsymbol{r} = \mathrm{fr}(\tau) & \\
& & \qquad\qquad\qquad\qquad\qquad\qquad\ \boldsymbol{\varrho} = \mathrm{fv}(\tau) & \\
& & \mathsf{in}\ \langle \mathsf{rec}\ f[\boldsymbol{r}]v':\tau', \sigma\rangle & \\
\qquad f^l & \Rightarrow & \mathsf{let}\ \forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau = \Gamma(f) & \\
& & \qquad\qquad \langle \tau, \sigma\rangle = \mathrm{inst}_l[\![\forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau]\!] & \\
& & \mathsf{in}\ \langle f\ \mathsf{at}\ \boldsymbol{r}\sigma : \tau\sigma, []\rangle & \\
\mathcal{U}[\![\tau, \tau']\!] = \mathsf{case}\ (\tau, \tau')\ \mathsf{of} & & & \\
\qquad\qquad \vdots & & & \\
\qquad (\tau_{r_l}, \tau'_{r'_{l'}}) & \Rightarrow & \mathsf{let}\ \sigma = [r_{l,l'}/r] \circ [r/r']\ \mathsf{in}\ \sigma \circ \mathcal{U}[\![\tau\sigma, \tau'\sigma]\!] &
\end{array}$$

Figure 11: Unification and inference algorithm for recursive functions

The second step consists of replacing all regions and extension variables of $\tau_0\sigma'$ by fresh ones. This is denoted by the operation $\tau = |\tau_0\sigma'|$ which is defined

by induction on the grammar of types by $|\alpha| = \alpha$, $|\tau_r| = |\tau|_{r'}$ for a fresh r' and $|\tau' \to \tau^{\boldsymbol{\rho}}| = |\tau'| \to |\tau|^{\varrho}$ for a fresh ϱ.

The fixed-point iterator is then applied to the type assignment $f : \forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau$ where $\boldsymbol{r}$ and $\boldsymbol{\varrho}$ are the fresh region and extension variables of τ.

Next, the instantiation of a recursive function name f at a program label l is modeled by the instantiation of its assignment $\forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau$ ate the label l. As we already discussed, this may yield the introduction of a limited number of fresh region and extension variables. The last operation that needs to be specified is the "widening" of regions and extension variables that is performed during unification. Given a type equation $\tau_{r_{\boldsymbol{l}}} =^? \tau'_{r'_{\boldsymbol{l'}}}$, this amounts to let r' match r and collapse the labels as $\boldsymbol{l}, \boldsymbol{l'}$ (substitution σ).

The soundness of the above inference algorithm with respect to the region calculus can simply be stated as follows [we write $\overline{\tau}$ the removal off all labels l from a term (e.g. τ)].

Theorem 3 (soundness of $\mathcal{I}$) *If* $\mathcal{I}[\![\Gamma, \boldsymbol{e}]\!] = \langle e : \epsilon, \sigma \rangle$ *then* $\overline{\Gamma\sigma} \vdash \overline{e} : \overline{\epsilon}$

The proof (see appendix) only departs from the standard induction principle of that of theorem 2 by considering the properties satisfied by the obtained fixed-point, which is precisely to satisfy the hypothesis of rule (REC)!

5 Related Work

The basic ideas of region inference are introduced by Tofte and Talpin in [20] and borrow notions of effect systems to Lucassen and Gifford [13], of algebraic reconstruction to Jouvelot and Gifford [11] and of effect inference to Talpin and Jouvelot [17].

Region inference has been developed for the typed call-by-value lambda calculus and it is used in the ML Kit with Regions [19]. A soundness proof for region inference is presented [20]. Other analyses which accompany region inference are described by Aiken in [1] and Birkedal in [3].

In [6], Crary et al. show that the principles of typed assembly languages of Morissett et al. [15] carry over to an extension of the region calculus with explicit, parametric polymorphism with subtyping.

In [2], Banerjee et al. propose a denotational theory for another extension $\mathcal{F}_{\#}$ of the region calculus in the polymorphic λ-calculus and establish the containment of the region calculus in $\mathcal{F}_{\#}$

In [7], Dal Zilio and Gordon establish an encoding of the region calculus in the π-calculus with groups and show how the soundness results of the former carry over in the latter generalization.

A couple of additional operational semantics for the region calculus have appeared more recently. One, due to Helsen and Thiemman [9], shows the soundness

of type and region assignment in the region calculus by considering a small-step operational semantics of the form $e \to e'$. The proposed semantics is not instrumented with a recording of store-effect (e.g. $\boldsymbol{r}$ in $e \xrightarrow{\boldsymbol{r}} e'$). Therefore, the soundness of observable store effect inference is not covered by this study (the type soundness theorem does not exhibit any relation between effects in the dynamic and the static semantics). Another proof, of Calcagno [4], establishes a correspondence or stratification between an abstract, high-level, big-step operational semantics for the region calculus, and the semantics of Tofte and Talpin [20], via a sophisticated relation of bisimulation.

In [21], Wadler proposes a small-step call-by-name semantics for a monadic reformulation of the imperative effect system of [17] (a monad $\mathsf{T}^{\rho}\,\tau$ stands for a type and effect τ^{ρ}). Unfortunately, the proposed operational semantics is non-deterministic[1] and does not meet a type soundness property [21, proposition 3.8]. A monadic reformulation [21, figure 9] of the effect inference algorithm of [17] is also given, but it is ill-typed[2] and cannot meet a completeness property either [21, proposition 4.3].

6 Conclusion

The related work demonstrates that the region calculus, both by its practical importance and its theoretical sophistication, has attracted the interest of many researchers and has been the subject of numerous recent studies. We hope to have successfully demonstrated that the region calculus could be given a much more simplified account of, by abstracting its semantics from the details of its actual implementation, yielding in a simple and purely syntactic small-step semantics. We have established new relations between fixed-point theory and type inference, showing that the determination of region polymorphic recursive function, could be specified in terms of a fixed-point iteration, given minimal hypothesis and instrumentation to ensure termination.

References

[1] Aiken, A., Faehndrich, M., and Levien, R. "Better static memory management: improvements to region-based analysis of higher-order languages". Conference on Programming Language Design and Implementation. ACM Press, 1995.

[1] For instance, $[l_1 : 0], \mathsf{new}\,(\mathsf{get}\,\mathsf{l}_1)$ either yields $[l_1 : 0, l_2 : \langle 0 \rangle], l_2$, of type $\mathsf{T}^{\mathsf{read}(\mathsf{r}_1),\mathsf{init}(\mathsf{r}_2)}\mathsf{ref}_{\mathsf{r}_2}\mathsf{int}$, from (get), (new0), (step), (new), (step), (trans) in [21, figure 5] or $[l_1 : 0, l_2 : \langle (\mathsf{get}\,\mathsf{l}_1) \rangle], l_2$, of type $\mathsf{T}^{\mathsf{init}(\mathsf{r}_2)}\mathsf{ref}_{\mathsf{r}_2}\mathsf{T}^{\mathsf{read}(\mathsf{r}_1)}\mathsf{int}$, from (new)

[2] it uses the unification algorithm $\mathcal{U}$ of the effect inference algorithm $\mathcal{I}_{\text{eff}}$ [21, figure 6]

[2] Banerjee, A., Heintze, N., and Riecke, J. " Region analysis and the polymorphic lambda calculus" Symposium on Logic in Computer Science, pp. 88-97. IEEE Press, 1999.

[3] Birkedal, L., and Tofte, M. "A constraint-based region inference algorithm". Theoretical Computer Science. Elsevier, 1998.

[4] Calcagno, C. "Stratified operational semantics for safety and correctness of region calculus". Symposium on Principles of Programming Languagues. ACM Press, 2000.

[5] Cousot, P. "Semantic foundations of program analysis". Program Flow Analysis: Theory and Applications, ch. 10, pp. 303-342, Prentice-Hall, 1981.

[6] Crary, K., Walker, D., and Morrisett, G. "Typed Memory Management in a Calculus of Capabilities". Symposium on Principles of Programming Languagues. ACM Press, 1999.

[7] Dal Zilio, S., and Gordon, A. D. "Region analysis and a p-calculus with groups". International Symposium on Mathematical Foundations of Computer Science. Lecture Notes in Computer Science, v. 1893. Springer Verlag, 2000.

[8] Harper, R. "A simplified account of polymorphic references". Information Processing Letters, v. 51, pp. 201-206. Elsevier, 1994.

[9] Helsen, S., and Thiemann, P. "Syntactic type soundness for the region calculus". Workshop on Higher Order Operational Techniques in Semantics. Elsevier electronic notes in theoretical computer science, 2000.

[10] Henglein, F. "Type inference with polymorphic recursion". Transactions on Programming Languages and Systems, v. 15(2), pp. 253–289. ACM Press, 1993.

[11] Jouvelot, P., and Gifford, D. "Algebraic reconstruction of types and effects". Symposium on Principles of Programming Languages. ACM Press, 1991.

[12] Leroy, X. "Typage polymorphe d'un langage algorithmique". Thèse de doctorat". Université Paris 7", 1992.

[13] Lucassen, J., and Gifford, D. "Polymorphic effect systems". Symposium on Principles of Programming Languages. ACM Press, 1988.

[14] Milner, R. "A theory of type polymorphismin programming languages". Journal of Computer and System Sciences, v. 17, pp. 348-375. Elsevier, 1978.

[15] Morissett, G., Walker, D., Crary, K., and Glew., N. "From system F to typed assembly language". Transactions on Programming Languages and Systems, v. 21(3), pp. 528-569. ACM Press, 1999.

[16] Remy, D. "Typing Record Concatenation for Free". Symposium on Principles Of Programming Languages". ACM Press, 1992.

[17] Talpin, J.-P., and Jouvelot, P. "The type and effect discipline". Information and Computation, Vol. 111(2), pages 245-296. Academic Press, 1994.

[18] Talpin, J.-P. "Aspects théoriques et pratiques de l'inférence de types et d'effets". Thèse de Doctorat. Université Paris VI and Ecole des Mines de Paris, May 1993.

[19] Tofte, M., Birkedal, L., Elsman, M., Hallenberg, N., Olesen, T., Sestoft ., and Bertelsen, P. "Programming with regions in the ML kit version 3". Technical report 98/25. Department of Computer Science, University of Copenhagen, 1998.

[20] Tofte, M., and Talpin, J.-P. "Region-based memory management". Information and Computation, Vol. 132(2), pages 109-176. Academic Press, 1997.

[21] Wadler, P. "The marriage of effects and monads". International Conference on Functional Programming. ACM Press, 1998.

[22] Wright, A., and Feleisen, M. "A syntactic approach to type soundness". Information and Computation, v. 115, pp. 38-94. Academic Press, 1994.

A Formal Proofs

A.1 Proof of lemma 1.

Lemma 1 *If* $e \xrightarrow{r}\!\!\!\!\searrow\ e'$ *and* $\Gamma \vdash e : \epsilon$ *then* $\Gamma \vdash e' : \epsilon$ *and* $\epsilon = \epsilon^{\boldsymbol{r}}$

The proof is by case analysis according to the reduction $e \xrightarrow{r} e'$ and by induction on the derivation of $e \xrightarrow{r} e'$

AXIOM (rec) By hypothesis,

$$(\mathsf{rec}\ f[\boldsymbol{r}]v)\ \mathsf{at}\ \boldsymbol{s} \rightharpoonup (v[\boldsymbol{s}/\boldsymbol{r}])[\mathsf{rec}\ f[\boldsymbol{r}]v/f] \quad \text{and} \quad \Gamma \vdash (\mathsf{rec}\ f[\boldsymbol{r}]v)\ \mathsf{at}\ \boldsymbol{s} : \epsilon \tag{1}$$

From (1) and the rule (REC-VAL)

$$\exists \tau, \epsilon = \tau[\boldsymbol{s}/\boldsymbol{r}] \quad \text{and} \quad \Gamma \vdash \mathsf{rec}\ f[\boldsymbol{r}]v : \tau \tag{2}$$

From (2) and the rule (REC)

$$\Gamma, [f : \forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau] \vdash v : \tau \quad \text{s.t.} \quad \boldsymbol{r} = \mathrm{fr}(\tau) \backslash \mathrm{fr}(\Gamma) \text{ and } \boldsymbol{\varrho} = \mathrm{fv}(\tau) \backslash \mathrm{fv}(\Gamma) \tag{3}$$

From (3) and the lemma 4

$$\Gamma, [f : \forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau] \vdash v[\boldsymbol{s}/\boldsymbol{r}] : \tau[\boldsymbol{s}/\boldsymbol{r}] \tag{4}$$

From (2), (4) and the lemma 5,

$$\Gamma, [f : \forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau] \vdash (v[\boldsymbol{s}/\boldsymbol{r}])[\mathsf{rec}\ f[\boldsymbol{r}]v/f] : \epsilon$$

AXIOM (abs) By hypothesis,

$$(\mathsf{fun}\ x.e)\ \mathsf{at}\ r \xrightarrow{r} (\mathsf{fun}\ x.e)_r \quad \text{and} \quad \Gamma \vdash \mathsf{fun}\ x.e\ \mathsf{at}\ r : ((\tau' \to \epsilon)_r)^r \tag{5}$$

From (5) and by the rule (ABS-VAL),

$$\Gamma \vdash (\mathsf{fun}\ x.e)_r : (\tau' \to \epsilon)_r$$

AXIOM (app) By hypothesis,

$$(\mathsf{fun}\ x.e)_r\ v \xrightarrow{r} e[v/x] \quad \text{and} \quad \Gamma \vdash (\mathsf{fun}\ x.e)_r\ v : \epsilon^r \tag{6}$$

From (6) and the rule (APP)

$$\Gamma \vdash (\mathsf{fun}\ x.e)_r : (\tau \to \epsilon)_r \quad \text{and} \quad \Gamma \vdash v : \tau \tag{7}$$

From (7) and the rule (ABS)

$$\Gamma, [x : \tau] \vdash e : \epsilon \tag{8}$$

From (7), (8) and the lemma 5,

$$\Gamma \vdash e[v/x] : \epsilon \tag{9}$$

From (9) and the rule (SUB)

$$\Gamma \vdash e[v/x] : \epsilon^r$$

AXIOM (free) By hypothesis,

$$v/r \rightharpoonup v[\mathsf{free}/r] \quad \text{and} \quad \Gamma \vdash v/r : \epsilon \tag{10}$$

From (10) and the rule (OBS)

$$\Gamma \vdash v : \epsilon^r \quad \text{and} \quad r \notin \mathrm{fr}(\Gamma) \cup \mathrm{fr}(\epsilon) \tag{11}$$

From (11) and the rule (ABS-VAL) their exists a derivation of

$$\Gamma \vdash v : \epsilon \tag{12}$$

From (12) and the lemma 4

$$\Gamma \vdash v[\mathsf{free}/r] : \epsilon$$

RULE (cont) By hypothesis

$$c[e] \xrightarrow{\boldsymbol{r}} c[e'] \quad \text{and} \quad \Gamma \vdash c[e] : \epsilon \tag{13}$$

From (13) and the rule (cont)

$$e \xrightarrow{\boldsymbol{r}} e' \quad \text{and} \quad \boldsymbol{r} \notin \mathrm{br}(c) \tag{14}$$

From (13) there exists Γ' and a derivation concluding

$$\Gamma, \Gamma' \vdash e : \epsilon' \tag{15}$$

From (14) and by induction hypothesis on the derivation (15)

$$\Gamma, \Gamma' \vdash e' : \epsilon' \quad \text{and} \quad \epsilon' = (\epsilon')^{\boldsymbol{r}} \tag{16}$$

From (13), (15), (16) and the lemma 3

$$\Gamma \vdash c[e'] : \epsilon \quad \text{and} \quad \epsilon = \epsilon^{\boldsymbol{r}}$$

RULE (mask) By hypothesis,

$$e/r \twoheadrightarrow e'/r \quad \text{and} \quad \Gamma \vdash e/r : \epsilon \tag{17}$$

From (17) and the rule (mask)

$$e \rightharpoonup e' \tag{18}$$

From (17) and the rule (OBS)

$$\Gamma \vdash e : \epsilon^r \quad \text{and} \quad r \notin \mathrm{fr}(\Gamma) \cup \mathrm{fr}(\epsilon) \tag{19}$$

From (18) and by induction hypothesis on the derivation (19)

$$\Gamma \vdash e' : \epsilon^r \tag{20}$$

From (20) and the rule (OBS)

$$\Gamma \vdash e'/r : \epsilon$$

RULE (trans) By hypothesis,

$$e \xrightarrow{r::s} e'' \quad \text{and} \quad \Gamma \vdash e : \epsilon \tag{21}$$

From (21) and by rule (trans)

$$e \xrightarrow{r} e' \quad \text{and} \quad e' \xrightarrow{s} e'' \tag{22}$$

From (21) and by induction hypothesis on (22)

$$\Gamma \vdash e' : \epsilon \quad \text{and} \quad \epsilon = \epsilon^r \tag{23}$$

From (23) and by induction hypothesis on (22)

$$\Gamma \vdash e'' : \epsilon \quad \text{and} \quad \epsilon = \epsilon^s$$

□

A.2 Proof of lemma 2.

Lemma 2 (weakening) *If* $\Gamma \vdash e : \epsilon$ *and* $x \notin \operatorname{dom} \Gamma$ *then* $\Gamma, [x : \tau] \vdash e : \epsilon$

The complete proof is by induction on the structure of e. In order to distinguish the regions bound in e from those free in τ, during the inductive development of the proof, one can simply use a bijective substitution to rename the bound ones. □

A.3 Proof of lemma 3.

Lemma 3 *If*

1. $\Gamma \vdash c[e] : \epsilon$
2. $\Gamma, \Gamma' \vdash e : \epsilon'$
3. $\Gamma, \Gamma' \vdash e' : \epsilon'$
4. $(\epsilon')^r = \epsilon'$ *and* $r \notin \operatorname{br}(c)$

then $\Gamma \vdash c[e'] : \epsilon$ *and* $\epsilon^r = \epsilon$

The proof is similar to that of [22, p. 15]. Take the axioms and rules of the figure 4 and note $\mathcal{T}$ the tree of deductions that corresponds to the derivation whose conclusion (and root of $\mathcal{T}$) is a judgment $\Gamma \vdash e : \epsilon$.

Let $\mathcal{T}$ the deduction of root $\Gamma \vdash c[e] : \epsilon$. which contains the subtree $\mathcal{T}'$ of root $\Gamma, \Gamma' \vdash e : \epsilon'$. Let $\mathcal{T}''$ a tree of root $\Gamma, \Gamma' \vdash e' : \epsilon'$ and replace $\mathcal{T}'$ by $\mathcal{T}''$ in $\mathcal{T}$ and e by e' in the remainder of $\mathcal{T}$. We obtain a tree whose conclusion is $\Gamma \vdash c[e'] : \epsilon$.

Now, suppose that ϵ' contains r and that r is not bound in $c[]$. Since the only rule that may remove r is (OBS) and requires s bound in c, then $\epsilon = \epsilon^r$ □

A.4 Proof of lemma 4.

Lemma 4 *If* $\Gamma \vdash e : \epsilon$ *then* $\Gamma\sigma \vdash e\sigma : \epsilon\sigma$

The proof is identical to that of [17, pp. 257-258]. Its structure is an induction on the structure of e. The inductive development of the proof requires a distinction between regions bound in e and regions free in $\mathrm{im}\,\sigma$. This is obtained by making use of a bijective substitution. □

A.5 Proof of lemma 5.

Lemma 5 *If*

1. $\Gamma, [x : \forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau] \vdash e : \epsilon$
2. $\boldsymbol{r} \notin \mathrm{fr}(\Gamma)$
3. $\boldsymbol{\varrho} \notin \mathrm{fv}(\Gamma)$
4. $\Gamma \vdash v : \tau$

then $\Gamma \vdash e[v/x] : \epsilon$.

The proof is by induction on the lenght of the derivation of (1). We proceed by case analysis on e.

Case $e = y \,\mathsf{at}\, \boldsymbol{s}$ By hypothesis (1)

$$\Gamma, [x : \forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau] \vdash y \,\mathsf{at}\, \boldsymbol{s} : \epsilon \tag{24}$$

If $x \neq y$ then from (24) the conclusion

$$\Gamma, [x : \forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau] \vdash y \,\mathsf{at}\, \boldsymbol{s}[v/x] : \epsilon$$

If $x = y$ then by hypothesis (4) and the lemma 4

$$\Gamma[\boldsymbol{s}/\boldsymbol{r}] \vdash v[\boldsymbol{s}/\boldsymbol{r}] : \tau[\boldsymbol{s}/\boldsymbol{r}] \tag{25}$$

From (25) and the hypothesis (2)

$$\Gamma \vdash v[\boldsymbol{s}/\boldsymbol{r}] : \tau[\boldsymbol{s}/\boldsymbol{r}] \tag{26}$$

If $v = (\mathsf{fun}\, z.e')_r$ and $\boldsymbol{r} = \boldsymbol{s} = []$ then from (26) the conclusion

$$\Gamma, [x : \forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau] \vdash x \,\mathsf{at}\, [][(\mathsf{fun}\, z.e')_r/x] : \epsilon$$

If $v = \mathsf{rec}\, z[\boldsymbol{r}]v'$ then from (26) and rule (REC-VAL) the conclusion

$$\Gamma, [x : \forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau] \vdash \overbrace{x \,\mathsf{at}\, [\boldsymbol{s}]}^{e}[\overbrace{\mathsf{rec}\, z[\boldsymbol{r}]v'}^{v}/x] : \epsilon$$

Case $e = \mathsf{fun}\ y.e' \ \mathsf{at}\ r$ By hypothesis (1)

$$\Gamma, [x : \forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau] \vdash \mathsf{fun}\ y.e' \ \mathsf{at}\ r : ((\tau_1 \to \tau_2^{\varrho,\boldsymbol{\rho}})_r)^r \tag{27}$$

From (27) and the rule ABS

$$\Gamma, [x : \forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau], [y : \tau_1] \vdash e' : \tau_2^{\varrho,\boldsymbol{\rho}} \tag{28}$$

Let σ be a bijection from $\boldsymbol{r}$ (resp. $\boldsymbol{\varrho}$) to fresh $\boldsymbol{r'}$ (resp. fresh $\boldsymbol{\varrho'}$). From (28), by the hypothesis (2-3) and the lemma 4

$$\Gamma, [y : \tau_1\sigma], [x : \forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau] \vdash e' : \tau_2^{\varrho,\boldsymbol{\rho}}\sigma \tag{29}$$

By hypothesis (4) and the lemma 2,

$$\Gamma, [y : \tau_1\sigma] \vdash v : \tau \tag{30}$$

From (29-30) and by induction hypothesis

$$\Gamma, [y : \tau_1\sigma], [x : \forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau] \vdash e'[v/x] : \tau_2^{\varrho,\boldsymbol{\rho}}\sigma \tag{31}$$

From (31), since σ is a bijection and by the lemma 4

$$\Gamma, [y : \tau_1], [x : \forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau] \vdash e'[v/x] : \tau_2^{\varrho,\boldsymbol{\rho}} \tag{32}$$

From (32) and the rule (ABS) with $y \neq x$

$$\Gamma, [x : \forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau] \vdash (\mathsf{fun}\ y.e' \ \mathsf{at}\ r)[v/x] : ((\tau_1 \to \tau_2^{\varrho,\boldsymbol{\rho}})_r)^r$$

Case $e = (\mathsf{fun}\ y.e')_r$ is identical to the above case of $e = \mathsf{fun}\ x.e' \ \mathsf{at}\ r$ and from the rule (ABS-VAL).

Case $e = \mathsf{rec}\ f[\boldsymbol{r}]\, e'$ By hypothesis (1)

$$\Gamma, [x : \forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau] \vdash \mathsf{rec}\ f[\boldsymbol{r}]\, e' : \tau_1^r \tag{33}$$

From (33) and the rule (REC)

$$\overbrace{\Gamma, [x : \forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau]}^{\Gamma'}[f : \forall \boldsymbol{r'}.\forall \boldsymbol{\varrho'}.\tau_1] \vdash e' : \tau_1^r \tag{34}$$

$$\boldsymbol{r'} = \mathrm{fr}(\tau_1)\backslash\mathrm{fr}(\Gamma') \cup \{r\} \text{ and } \boldsymbol{\varrho'} = \mathrm{fv}(\tau_1)\backslash\mathrm{fv}(\Gamma') \tag{35}$$

By hypothesis (4) and the lemma 2,

$$\Gamma, [f : \forall \boldsymbol{r'}.\forall \boldsymbol{\varrho'}.\tau_1] \vdash v : \tau \tag{36}$$

From (34) and (36), by the hypothesis (2) and (3) and by induction on e',

$$\overbrace{\Gamma, [x : \forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau]}^{\Gamma'}[f : \forall \boldsymbol{r'}.\forall \boldsymbol{\varrho'}.\tau_1] \vdash e'[v/x] : \tau_1^r \tag{37}$$

From (35), (37) and the rule (REC),

$$\Gamma \vdash (\mathsf{rec}\ f[\boldsymbol{r}]\, e')[v/x] : \tau_1^r$$

Case $e = (\mathsf{rec}\ x[\boldsymbol{r}]\, v)\,\mathsf{at}\,\boldsymbol{s}$ is identical to the above case of $e = \mathsf{rec}\ x[\boldsymbol{r}]\, v$ and from the rule (REC-VAL).

Case $e = e'/r$ A routine induction. By hypothesis,

$$\Gamma, [x : \forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau] \vdash e'/r : \epsilon$$

By definition of rule (OBS),

$$\Gamma, [x : \forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau] \vdash e' : \epsilon^r$$

By induction hypothesis on e'

$$\Gamma, [x : \forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau] \vdash e'[v/x] : \epsilon^r$$

By definition of rule (OBS),

$$\Gamma, [x : \forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau] \vdash (e'[v/x])/r : \epsilon$$

Case $e = e'\, e''$ By hypothesis,

$$\Gamma, [x : \forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau] \vdash e'\, e'' : \epsilon^{\boldsymbol{\rho},\boldsymbol{\rho}',r}$$

By definition of rule (APP),

$$\Gamma \vdash e' : ((\tau \to \epsilon)_r)^{\boldsymbol{\rho}} \quad \text{and} \quad \Gamma \vdash e'' : \tau^{\boldsymbol{\rho}'}$$

By induction hypothesis on e'

$$\Gamma \vdash e'[v/x] : ((\tau \to \epsilon)_r)^{\boldsymbol{\rho}}$$

By induction hypothesis on e''

$$\Gamma \vdash e''[v/x] : \tau^{\boldsymbol{\rho}'}$$

By definition of rule (APP),

$$\Gamma, [x : \forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau] \vdash (e'\, e'')[v/x] : \epsilon^{\boldsymbol{\rho},\boldsymbol{\rho}',r}$$

□

A.6 Proof of theorem 1.

Theorem 1 *If* $e : \mathsf{unit}$ *then either* $e \Uparrow$ *or* $e \rightharpoonup \mathsf{unit}$

The proof is similar to that of [22, p. 20]. Let e a closed expression of type unit. By the lemma 6, either $e \rightharpoonup v$, $e \Uparrow$ or $e \not\rightharpoonup$.

Suppose $e \rightharpoonup v$, from $e : \mathsf{unit}$ and by the lemma 1, $v : \mathsf{unit}$. Hence, $v = \mathsf{unit}$.

Suppose $e \not\rightharpoonup$, by definition, this requires that $e \rightharpoonup e'$ and that $e'' \not\rightharpoonup$ for all e'' and c such that $e' = c[e'']$. From $e : \mathsf{unit}$, $e \rightharpoonup e'$ and by the lemma 1, $e' : \mathsf{unit}$. From $e \not\rightharpoonup$ and the lemma 7, $\neg(e : \epsilon)$. Since $e : \mathsf{unit}$ contradicts $\neg(e : \epsilon)$, the hypothesis $e \not\rightharpoonup$ is false.

Hence, either $e \Uparrow$ or $e \rightharpoonup \mathsf{unit}$. □

A.7 Proof of lemma 6.

Lemma 6 *For a closed e, either $e \rightharpoonup v$, $e \Uparrow$ or $e \not\rightharpoonup$*

The proof is identical to the that of [22, p. 19] which is done by induction on the length of the reduction sequence. For a closed e (i.e. s.t. $\mathrm{fv}(e) = \mathrm{fr}(e) = \emptyset$), we need to show that either e is faulty, that $e \rightharpoonup e'$ and that e' is closed or that $e \rightharpoonup v$. By definition of $\rightharpoonup$, $e \rightharpoonup e'$ requires $e_1 \rightharpoonup e_1'$ with $e = c[e_1]$ and $e' = c[e_1']$ from rule (CONT). Recalling that $c ::= [] \mid \mathsf{rec}\ x[\boldsymbol{r}]\, c \mid c/r \mid c\, e'' \mid v\, c$, the proof proceeds by induction on the structure of e as in [22, p. 19]. □

A.8 Proof of lemma 7.

Lemma 7 *For a closed e, if $e \not\rightharpoonup$ then $\neg(e : \epsilon)$*

By definition, $e \not\rightharpoonup$ requires that $e \rightharpoonup e'$ and that $e'' \not\rightharpoonup$ for all e'' and c such that $e' = c[e'']$. To prove the lemma, it is sufficient to proceed by case analysis on $e'' \not\rightharpoonup$. According to the side-conditions of the axioms and rules of figure 4, there are four cases

$$|\boldsymbol{r}| \neq |\boldsymbol{s}| \text{ and } (\mathsf{rec}\ f[\boldsymbol{r}]v)\ \mathsf{at}\ \boldsymbol{s} \not\rightharpoonup \quad (38)$$

$$(\mathsf{fun}\ x.e)\ \mathsf{at}\ \mathsf{free} \not\rightharpoonup \quad (39)$$

$$(\mathsf{fun}\ x.e)_{\mathsf{free}}\ v \not\rightharpoonup \quad (40)$$

$$\mathsf{unit}\ v \not\rightharpoonup \quad (41)$$

Condition ($|\boldsymbol{r}| \neq |\boldsymbol{s}|$) of case (38) contradicts the side-condition of rule (REC-VAL). Since e is a closed expression, $\mathrm{fr}(e) = \emptyset$ hence $\mathsf{free} \notin \mathrm{fr}(e)$ and cases (39-40) are thus impossible for a closed expression e. Case (41) requires $\Gamma \vdash \mathsf{unit} : \mathsf{unit}$ and $\Gamma \vdash v : \epsilon$ which cannot match the hypothesis of rule (APP). In conclusion, if $e'' \not\rightharpoonup$ then $\neg(\Gamma \vdash e'' : \epsilon)$ for all Γ, hence the lemma. □

A.9 Proof of theorem 2.

Theorem 2 *If $\mathcal{I}[\![\Gamma, \boldsymbol{e}]\!] = \langle e : \epsilon, \sigma \rangle$ (figure 8) then $\Gamma\sigma \vdash e : \epsilon$*

The proof is identical to that of [18, pp. 115-116] modulo notations. It proceeds by a mutual induction on the recursive calls to $\mathcal{I}$ and $\mathcal{I}'$, which is handled by case analysis on the structure of $\boldsymbol{e}$. The treatment of the call to the function $\mathcal{I}'$ within $\mathcal{I}$ can be factorized as an intermediate lemma:

$$\text{If } \mathcal{I}'[\![\Gamma, \boldsymbol{e}]\!] = \langle e : \epsilon, \sigma \rangle \text{ (figure 8) then } \Gamma\sigma \vdash e : \epsilon$$

to which the rule (OBS) and (OBS′) apply to derive the result. Each recursive call to $\mathcal{I}$ within $\mathcal{I}'$ corresponds to an induction hypothesis □

A.10 Proof of lemma 8.

Lemma 8 *If* $\mathcal{U}[\![\tau, \tau']\!] = \sigma$ *then* $\tau\sigma = \tau'\sigma$

The proof is identical to that of [18, pp. 115] modulo notations. It proceeds by induction on the structure of the pair $[\![\tau, \tau']\!]$ and the inspection of the corresponding scenari of $\mathcal{U}$.

Note that, in the case of the extension of the unification algorithm with labelled regions (figure 11), it similarly holds that, if $\mathcal{U}[\![\tau, \tau']\!] = \sigma$, then $\tau\sigma = \tau'\sigma$ (for the algorithm) and thus $\overline{\tau\sigma} = \overline{\tau'\sigma}$ (for the inference system) □

A.11 Proof of lemma 9.

Lemma 9 *For all finite environments* Γ *and* λ*-abstractions* v*, the iteration* $\mathcal{F}_{\Gamma,\boldsymbol{v}}$ *starting from a finite* $\mathcal{C}^0_{\Gamma,\boldsymbol{v}}$ *either fails by* $\mathcal{U}$ *or terminates after a finite number* $n < \omega$ *of iterations, yielding a fixed-point* $\mathcal{C}^n_{\Gamma,\boldsymbol{v}} = \mathcal{F}_{\Gamma,\boldsymbol{v}}\,\mathcal{C}^n_{\Gamma,\boldsymbol{v}}$.

First note that the operator $\mathcal{F}$ is extensive for the partial order relation $\sqsubseteq$. Indeed, notice that the successive results of the iteration are unified the ones with the others. This has two effects. One is to preserve the partial order $\sqsupseteq$ between each iteration. Therefore, $\mathcal{F}$ admits a least fixed-point.

The other effect of the unification is to force the number of labels l to grow and accumulate in the regions of $\boldsymbol{v}$, thus gradually limitating the amount of polymorphism.

Hence, for a finite environment Γ and a λ-abstraction $\boldsymbol{v}$, the number of regions introduced by $\mathcal{F}$ is correlated to the number of labels present in $\boldsymbol{v}$, which equals the number of occurrences of the recursive function name f in $\boldsymbol{v}$.

As a result, given that each iteration $\mathcal{F}$ introduces decreasingly finitely many regions, the strictly increasing chain $\mathcal{C}$ is finite and there exists $n < \omega$ s.t. $\mathcal{C}^n_{\Gamma,\boldsymbol{v}} = \mathcal{F}_{\Gamma,\boldsymbol{v}}\,\mathcal{C}^n_{\Gamma,\boldsymbol{v}}$ □

A.12 Proof of theorem 3.

Theorem 3 *If* $\mathcal{I}[\![\Gamma, \boldsymbol{e}]\!] = \langle e : \epsilon, \sigma\rangle$ *then* $\overline{\Gamma\sigma} \vdash \overline{e} : \overline{\epsilon}$

The proof is by induction on the structure of expression or, equivalently, by induction on the recursive calls to $\mathcal{I}$. The case analysis for expressions other than of the forms $\mathsf{rec}\ f.\boldsymbol{e}$ and f_l is identical to that of theorem 2 modulo the temporary insertion/removal of labels.

Case of $\boldsymbol{e} = \mathsf{rec}\ f.\boldsymbol{v}$ By hypothesis

$$\mathcal{I}'[\![\Gamma, \mathsf{rec}\ f.\boldsymbol{v}]\!] = \langle \mathsf{rec}\ f\,[\boldsymbol{r}]e' : \tau', \sigma\rangle \tag{42}$$

By definition of figure 11, this requires that

$$\begin{aligned}
\mathrm{lfp}\,\mathcal{F}_{\Gamma,\boldsymbol{v}}[\![f:\forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau, \boldsymbol{v}:\tau, []\!] &= \langle f:\forall \boldsymbol{r}'.\forall \boldsymbol{\varrho}'.\tau', v':\tau', \sigma\rangle && (43)\\
\mathcal{I}'[\![\Gamma[f:\alpha], \boldsymbol{v}]\!] &= \langle v_0:\tau_0, \sigma_0\rangle \text{ and } \alpha \text{ fresh} && (44)\\
\mathcal{U}[\![\alpha\sigma_0, \tau_0]\!] &= \sigma_0' && (45)\\
\tau &= |\tau_0\sigma_0'| && (46)\\
\boldsymbol{r} = \mathrm{fr}(\tau) \text{ and } \boldsymbol{\varrho} &= \mathrm{fv}(\tau) && (47)
\end{aligned}$$

Conditions (44-47) make sure that τ is the largest element of the chain $\mathcal{C}_{\Gamma,\boldsymbol{v}}$ for $\sqsupseteq$. Then, from (43) and the lemma 9, there exists $0 < n < \omega$ s.t.

$$\mathcal{F}_{\Gamma,\boldsymbol{v}}\langle f:\forall \boldsymbol{r}_n.\forall \boldsymbol{\varrho}_n.\tau_n, \boldsymbol{v}:\tau_n, \sigma_n\rangle = \langle f:\forall \boldsymbol{r}_{n+1}.\forall \boldsymbol{\varrho}_{n+1}.\tau_{n+1}, \boldsymbol{v}:\tau_{n+1}, \sigma_{n+1}\rangle \quad (48)$$

From (43) , (48) and by definition 1,

$$\begin{aligned}
\mathcal{I}'[\![\Gamma\sigma_n, [f:\forall \boldsymbol{r}_n.\forall \boldsymbol{\varrho}_n.\tau_n], \boldsymbol{v}]\!] &= \langle v_{n+1} \,\mathsf{at}\, r:(\tau_{n+1})^r, \sigma\rangle && (49)\\
\sigma_{n+1} &= \sigma_n \circ \sigma \circ \mathcal{U}[\![\tau_n\sigma, \tau_{n+1}]\!] && (50)\\
\boldsymbol{r}_{n+1} &= (\mathrm{fr}(\tau_{n+1}\sigma_{n+1})\backslash \mathrm{fr}(\Gamma\sigma_{n+1}))\backslash r\sigma_{n+1} && (51)\\
\boldsymbol{\varrho}_{n+1} &= \mathrm{fv}(\tau_{n+1}\sigma_{n+1})\backslash \mathrm{fv}(\Gamma\sigma_{n+1}) && (52)
\end{aligned}$$

From (49) and by induction hypothesis on $\boldsymbol{v}$,

$$\overline{(\Gamma\sigma_n, [f:\forall \boldsymbol{r}_n.\forall \boldsymbol{\varrho}_n.\tau_n])\sigma \vdash v_{n+1} \,\mathsf{at}\, r:(\tau_{n+1})^r} \quad (53)$$

From (50-53) and the lemma 4

$$\overline{\Gamma\sigma_{n+1}, [f:\forall \boldsymbol{r}_n.\forall \boldsymbol{\varrho}_n.\tau_n\sigma_{n+1}] \vdash (v_{n+1} \,\mathsf{at}\, r)\sigma_{n+1}:((\tau_{n+1})^r)\sigma_{n+1}} \quad (54)$$

From (48) and (54)

$$\overline{\Gamma\sigma_{n+1}, [f:\forall \boldsymbol{r}_{n+1}.\forall \boldsymbol{\varrho}_{n+1}.\tau_{n+1}] \vdash (v_{n+1} \,\mathsf{at}\, r)\sigma_{n+1}:(\tau_{n+1})^{r\sigma_{n+1}}} \quad (55)$$

From (50-52), (55) and the rule (REC),

$$\overline{\Gamma\sigma_{n+1} \vdash \mathsf{rec}\, f[\boldsymbol{r}_{n+1}](v_{n+1} \,\mathsf{at}\, r)\sigma_{n+1}:(\tau_{n+1})^{r\sigma_{n+1}}}$$

Case of $\boldsymbol{e} = f_l$ By hypothesis

$$\mathcal{I}'[\![\Gamma, [f:\forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau], f^l]\!] = \langle f \,\mathsf{at}\, \boldsymbol{r}\sigma:\tau\sigma, []\rangle \quad (56)$$

By definition of figure 11, this requires that

$$\mathrm{inst}_l[\![\forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau]\!] = \langle \tau, \sigma\rangle \quad (57)$$

By definition of inst_l and the rule (var),

$$\overline{\Gamma, [f:\forall \boldsymbol{r}.\forall \boldsymbol{\varrho}.\tau] \vdash f \,\mathsf{at}\, \boldsymbol{r}\sigma:\tau\sigma}$$

□

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399