HAL
archives-ouvertes.fr

An affine-intuitionistic system of types and effects: confluence and termination

Roberto Amadio, Patrick Baillot, Antoine Madet

▶ To cite this version:

HAL Id: hal-00438101

https://hal.archives-ouvertes.fr/hal-00438101v2

Submitted on 3 May 2010

An Affine-Intuitionistic System of Types and Effects: Confluence and Termination

Roberto M. Amadio[*]   Patrick Baillot[†]   Antoine Madet[*]

Abstract

We present an affine-intuitionistic system of *types and effects* which can be regarded as an extension of Barber-Plotkin *Dual Intuitionistic Linear Logic* to multi-threaded programs with effects. In the system, dynamically generated values such as references or channels are abstracted into a finite set of *regions*. We introduce a discipline of *region usage* that entails the *confluence* (and hence determinacy) of the typable programs. Further, we show that a discipline of region *stratification* guarantees *termination*.

1 Introduction

There is a well-known connection between *intuitionistic proofs* and *typed functional programs* that goes under the name of *Curry-Howard* correspondence. Following the introduction of *linear logic* [9], this correspondence has been refined to include an explicit treatment of the process of data duplication. Various formalisations of these ideas have been proposed in the literature (see, *e.g.*, [3, 4, 17, 16, 2]) and we will focus here in particular on Affine-Intuitionistic Logic and, more precisely, on an *affine* version of Barber-Plotkin *Dual Intuitionistic Linear Logic* (DILL) as described in [2].

In DILL, the operation of λ-abstraction is always *affine*, *i.e.*, the formal parameter is used at most once. The more general situation where the formal parameter has multiple usages is handled through a constructor '$!$' (read bang) marking values that can be duplicated and a destructor let filtering them and effectively allowing their duplication. Following this idea, *e.g.*, an intuitionistic judgement (1) is translated into an affine-intuitionistic one (2) as follows:

$$y : A \vdash \lambda x.x(xy) : (A \to A) \to A \tag{1}$$

$$y : (\infty, A) \vdash \lambda x.\mathsf{let}\ !z = x\ \mathsf{in}\ z!(z!y) : !(!A \multimap A\ ) \multimap A \tag{2}$$

We recall that in DILL the hypotheses are split in two zones according to their *usage*. Namely, one distinguishes between the *affine* hypotheses that can be used at most once and the *intuitionistic* ones that can be used arbitrarily many times. In our formalisation, we will use $'1'$ for the former and $'\infty'$ for the latter.

---

[1]Laboratoire PPS, Université Paris Diderot
[2]LIP - ENS-Lyon, Université de Lyon

1.1 Motivations

Our purpose is to explore an *extension* of this connection to *multi-threaded programs* with *effects*. By extending the connection, we mean in particular to design an affine-intuitionistic type system that accounts for multi-threading and side effects and further to refine the system in order to guarantee confluence (and hence determinism) and termination while preserving a reasonable expressive power. By multi-threaded program, we mean a program where distinct threads of execution may be active at the same time (as it is typically the case in concurrent programs) and by effect, we mean the possibility of executing operations that modify the *state* of a system such as reading/writing a reference or sending/receiving a message. We stress that our aim is *not* to give a purely logical interpretation of multi-threading and side-effects but rather to apply *logical methods* to a multi-threaded programming language with side-effects.

1.2 Contributions

We will start by introducing a simple-minded extension of the purely functional language with operators to run threads in parallel while reading/writing the store which is loosely inspired by *concurrent* extensions of the ML programming language such as [8] and [18] with an interaction mechanism based on (asynchronous) *channel* communication. In particular, we rely on an operator $\mathsf{get}(x)$ to read a value from an address (channel) x and on two operators $\mathsf{set}(x, V)$ and $\mathsf{pset}(x, V)$ to write a value V into an address x, in a volatile (value read is consumed) or persistent (value read is still available) way, respectively.

Following a rather standard practice (see, *e.g.*, [15, 20]), we suppose that dynamically generated values such as channels or references are *abstracted* into a finite number of *regions*. This abstraction is reflected in the type system where the type of an address *depends* on the region with which the address is associated. Thus we write $\mathsf{Reg}_r A$ for the type of addresses containing values of type A and relating to the region r of the store. Our first and probably most difficult contribution, due to the interaction of the bang modality $'!'$ with regions, is to design a system where types and usages are preserved by reduction.

The resulting functional-concurrent typed language is neither confluent nor terminating. However, it turns out that there are reasonable strategies to recover these properties. The general idea is that *confluence* can be recovered by introducing a proper discipline of *region usage* while *termination* can be recovered through a discipline of *region stratification*.

The notion of *region usage* is reminiscent of the one of *hypotheses usage* arising in affine-intuitionistic logic. Specifically, we distinguish the regions that can be used at most once to write and at most once to read from those that can be used at most once to write and arbitrarily many times to read.

The notion of *region stratification* is based on the idea that values stored in a region should only produce effects on *smaller* regions. The implementation of this idea requires a substantial refinement of the type system that has to predict the *effects* potentially generated by the evaluation of an expression.

2

This is where *type and effect systems*, as introduced in [15], come into play.

It turns out that the notions of region usage and region stratification combine smoothly, leading to the definition of an affine-intuitionistic system of types and effects. The system has affine-intuitionistic logic as its functional core and it can be used to guarantee the determinacy and termination of multi-threaded programs with effects. We stress that the nature of our contribution is mainly methodological and that more theoretical and experimental work is needed to arrive at a usable programming language. One promising direction is to add inductive data types and to extend the language to a synchronous/timed framework (cf. [1, 6]). In this framework, both confluence (determinism) and termination are valuable properties.

1.3 Related Work

Girard, through the introduction of *linear logic* [9], has widely promoted a finer analysis of the *structural rules* of logic. There have been various attempts at producing a functional programming language based on these ideas and with a reasonably handy syntax (see, *e.g.*, [3, 4, 17, 16, 2]). The logical origin of the notion of *usage* can be traced back to Girard's LU system [10] and in particular it is adopted in the Barber-Plotkin system [2] on which we build on.

A number of works on type systems for concurrent languages such as the π-calculus have been inspired by linear logic even though in many cases the exact relationships with logic are at best unclear even for the fragment without side-effects. The conditions to guarantee *confluence* are inspired by the work of Kobayashi *et al.* [14] and one should expect a comparable expressive power (see also [13, 12] for much more elaborate notions of usage).

It is well known that intuitionistic logic is at the basis of typed functional programming. The *type and effect* system introduced in [15] is an enrichment of the intuitionistic system tracing the effects of *imperative* higher-order programs acting on a *store*. The system has provided a successful static analysis tool for the problem of *heap-memory deallocation* [20]. More recently, this issue has been revisited following the ideas of linear logic [23, 7] .

The so called *reducibility candidates method* is probably the most important technique to prove *termination* of typable higher-order programs. Extensions of the method to 'functional fragments' of the π-calculus have been proposed, *e.g.*, in [24, 19]. Boudol [6] has shown that a stratification of the regions guarantees termination for a multi-threaded higher-order functional language with references and cooperative scheduling. Our formulation of the stratification discipline is actually based on [1] which revisits and extends [6].

1.4 Structure of the Paper

Section 2 introduces an affine-intuitionistic system with regions for a call-by-value functional-concurrent language. Section 3 introduces a discipline of region usage that guarantees confluence of the typable programs. Section 4 enriches the affine-intuitionistic system introduced in Section 2 with a notion of effect

$$x, y, \ldots \qquad \qquad \text{(Variables)}$$

$x, y, \ldots$			(Variables)
V	$::=$	$* \mid x \mid \lambda x.M \mid !V$	(Values)
M	$::=$	$V \mid MM \mid !M \mid \mathsf{let}\ !x = M\ \mathsf{in}\ M \mid \nu x\ M$	
		$\mathsf{set}(x, V) \mid \mathsf{pset}(x, V) \mid \mathsf{get}(x) \mid (M \mid M)$	(Terms)
S	$::=$	$(x \leftarrow V) \mid (x \Leftarrow V) \mid (S \mid S)$	(Stores)
P	$::=$	$M \mid S \mid (P \mid P) \mid \nu x\ P$	(Programs)
E	$::=$	$[\ ] \mid EM \mid VE \mid !E \mid \mathsf{let}\ !x = E\ \mathsf{in}\ M$	(Evaluation Contexts)
C	$::=$	$[\ ] \mid (C \mid P) \mid (P \mid C) \mid \nu x\ C$	(Static Contexts)

Table 1: Syntax: programs

which provides an upper bound on the set of regions on which the evaluation of a term may produce effects. Finally, Section 5 describes a discipline of region stratification that guarantees the termination of the typable programs. Proofs of the main results are available in Appendix A.

2 An Affine-Intuitionistic Type System with Regions

We introduce a typed functional-concurrent programming language equipped with a call-by-value evaluation strategy. The functional core of the language relies on Barber-Plotkin's DILL. In order to type the dynamically generated addresses of the store, we introduce *regions* and suitable notions of *usages*. The related type system enjoys *weakening* and *substitution* and this leads to the expected properties of *type preservation* and *progress*.

2.1 Syntax: Programs

Table 1 introduces the syntax of our programs. We denote variables with $x, y, \ldots$, and with V the values which are included in the category M of terms. Stores are denoted by S, and programs P are combinations of terms and stores. We comment on the main operators of the language. $*$ is a constant inhabiting the terminal type $\mathbf{1}$ (see below). $\lambda x.M$ is the *affine* abstraction and MM the application. ! marks values that can be duplicated while $\mathsf{let}\ !x = M\ \mathsf{in}\ N$ filters them and allows their multiple usage in N. In $\nu x\ M$ the operator ν generates a fresh address name x whose scope is M. $\mathsf{set}(x, V)$ and $\mathsf{pset}(x, V)$ write the value V in a *volatile* address and a *persistent* one, respectively, while $\mathsf{get}(x)$ fetches a value from the address x (either volatile or persistent). Finally $(M \mid N)$ evaluates in parallel M and N. Note that when writing either $\lambda x.M$, or $\nu x\ M$, or $\mathsf{let}\ !x = N\ \mathsf{in}\ M$ the variable x is bound in M. As usual, we abbreviate $(\lambda z.N)M$ with $M; N$, where z is not free in N. *Evaluation contexts* E follow a *call-by-value* discipline. *Static contexts* C are composed of parallel

composition and ν's. Note that stores can only appear in a static context. Thus $M = V(\mathsf{set}(x, V'); V'')$ is a legal term while $M' = V(V'' \mid (x \leftarrow V))$ is not.

2.2 Operational Semantics

Table 2 describes the operational semantics of our language. Programs are

$$
\begin{array}{rcll}
P \mid P' & \equiv & P' \mid P & \text{(Commut.)} \\
(P \mid P') \mid P'' & \equiv & P \mid (P' \mid P'') & \text{(Assoc.)} \\
\nu x\ P \mid P' & \equiv & \nu x\ (P \mid P') \quad x \notin FV(P') & (\nu_{\mid}) \\
E[\nu x\ M] & \equiv & \nu x\ E[M] \quad x \notin FV(E) & (\nu_E)
\end{array}
$$

$$
\begin{array}{rcl}
E[(\lambda x.M)V] & \to & E[[V/x]M] \\
E[\mathsf{let}\ !x = !V\ \mathsf{in}\ M] & \to & E[[V/x]M] \\
E[\mathsf{set}(x, V)] & \to & E[*] \mid (x \leftarrow V) \\
E[\mathsf{pset}(x, V)] & \to & E[*] \mid (x \Leftarrow V) \\
E[\mathsf{get}(x)] \mid (x \leftarrow V) & \to & E[V] \\
E[\mathsf{get}(x)] \mid (x \Leftarrow !V) & \to & E[!V] \mid (x \Leftarrow !V)
\end{array}
$$

Table 2: Operational semantics

considered up to a *structural equivalence* $\equiv$ which is the least equivalence relation preserved by static contexts, and which contains the equations for α-renaming, for the commutativity and associativity of parallel composition, for enlarging the scope of the ν operators to parallel programs, and for extracting the ν from an evaluation context. We use the notation $[V/x]$ for the substitution of the value V for the variable x. The reduction rules apply modulo structural equivalence and in a static context C.

Example 1.
The programs (3) and (4) are structurally equivalent (up to some renaming):

$$((\nu x\ \lambda y.M)(\nu x'\ \lambda x'.M'))V \mid P \tag{3}$$

$$\nu x\ \nu x'\ ((\lambda y.M)(\lambda y'.M'))V \mid P \tag{4}$$

This transformation exposes the term $E[(\lambda y.M)(\lambda y'.M')]$ in the static context $C = \nu x\ \nu x'\ [\ ]\mid P$, where the evaluation context E is $[\ ]V$.

In the sequel we consider the transitive closure of the relation defined by Table 2, also denoted $\to$.

Remark 1. Notice that the let rule and the get rule on a persistent store act similarly in the sense that they require the value being duplicated to be marked with a bang, while the affine β rule and the get rule on a volatile store allow to manipulate affine values.

$$
\begin{array}{llll}
r, r', \ldots & & & \text{(Regions)} \\
\alpha & ::= & \mathbf{B} \mid A & \text{(Types)} \\
A & ::= & \mathbf{1} \mid A \multimap \alpha \mid \,!A \mid \mathsf{Reg}_r A & \text{(Value-types)} \\
\Gamma & ::= & x_1 : (u_1, A_1), \ldots, x_n : (u_n, A_n) & \text{(Contexts)} \\
R & ::= & r_1 : (U_1, A_1), \ldots, r_n : (U_n, A_n) & \text{(Region contexts)}
\end{array}
$$

Table 3: Syntax: types and contexts

2.3   Syntax: Types and Contexts

Table 3 introduces the syntax of types and contexts. We denote regions with $r, r', \ldots$ and we suppose a region r is either *volatile* ($\mathcal{V}(r)$) or *persistent* ($\mathcal{P}(r)$). Types are denoted with $\alpha, \alpha', \ldots$. Note that we distinguish a special behaviour type $\mathbf{B}$ which is given to the entities of the language which are not supposed to return a value (such as a store or several values in parallel) while types of entities that may return a value are denoted with A. Among the types A, we distinguish a terminal type $\mathbf{1}$, an affine functional type $A \multimap B$, the type $!A$ of terms of type A that can be duplicated, and the type $\mathsf{Reg}_r A$ of addresses containing values of type A and related to the region r. Hereby types may depend on regions.

Before commenting on variable and region contexts, we need to define the notion of *usage*. To this end, it is convenient to introduce a set with three values $\{0, 1, \infty\}$ and a *partial* binary operation $\uplus$ such that

$$
\begin{array}{rcl}
x \uplus 0 & = & x \\
0 \uplus x & = & x \\
\infty \uplus \infty & = & \infty
\end{array}
$$

and which is undefined otherwise.

We denote with u a *variable usage* and assume that u is either 1 (a variable to be used at most once) or ∞ (a variable that can be used arbitrarily many times). Then a variable context (or simply a context) Γ has the shape:

$$
x_1 : (u_1, A_1), \ldots, x_n : (u_n, A_n)
$$

where x_i are distinct variables, $u_i \in \{1, \infty\}$ and A_i are types of terms that may return a result. Writing $x : (u, A)$ means that the variable x ranges on values of type A and can be used according to u. We write $dom(\Gamma)$ for the set $\{x_1, \ldots, x_n\}$ of variables where the context is defined. The sum on usages is extended to contexts component-wise. In particular, if $x : (u_1, A) \in \Gamma_1$ and $x : (u_2, A) \in \Gamma_2$ then $x : (u_1 \uplus u_2, A) \in (\Gamma_1 \uplus \Gamma_2)$ only if $u_1 \uplus u_2$ is defined.

Example 2. ⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯
One may check that the sum:

$$
(x : (1, A), y : (\infty, B)) \uplus (y : (\infty, B), z : (1, C))
$$

is equal to

$$x : (1, A), y : (\infty, B), z : (1, C)$$

whereas these two are not defined:

$$(x : (1, A), y : (\infty, B)) \uplus y : (1, B)$$
$$(x : (1, A), y : (1, B) \uplus y : (1, B)$$

We are going to associate a usage with regions too, but in this case a usage will be a two dimensional vector because we want to be able to distinguish write and read usages. We denote with U an element of one of the following three sets of usages:

$$\{[\infty, \infty]\} \quad \{[1, \infty], [0, \infty]\} \quad \{[0, 0], [1, 0], [0, 1], [1, 1]\}$$

where by convention we reserve the first component to describe the write usage and the second for the read usage. Thus a region with usage $[1, \infty]$ should be written at most once while it can be read arbitrarily many times.

The addition $U_1 \uplus U_2$ is defined provided that:

(a) U_1 and U_2 are in the same set of usages

(b) the component-wise addition is defined

Example 3.
If $U_1 = [\infty, \infty]$ and $U_2 = [0, \infty]$ then the sum is undefined because U_1 and U_2 are not in the same set while if $U_1 = [1, \infty]$ and $U_2 = [1, \infty]$ then the sum is undefined because $1 \uplus 1$ is undefined.

Note that in each set of usages there is a *neutral* usage U_0 such that $U_0 \uplus U = U$ for all U in the same set.

A region context R has the shape:

$$r_1 : (U_1, A_1), \ldots, r_n : (U_n, A_n)$$

where r_i are distinct regions, U_i are usages in the sense just defined, and A_i are value-types. The typing system will additionally guarantee that whenever we use a type $\mathsf{Reg}_r A$ the region context contains an hypothesis $r : (U, A)$ for some U. Intuitively, writing $r : (U, A)$ means that addresses related to region r contain values of type A and that they can be used according to the usage U. We write $dom(R)$ for the set $\{r_1, \ldots, r_n\}$ of the regions where the region context is defined. As for contexts, the sum on usages is extended to region contexts component-wise. In particular, if $r : (U_1, A) \in R_1$ and $r : (U_2, A) \in R_2$ then $r : (U_1 \uplus U_2, A) \in (R_1 \uplus R_2)$ only if $U_1 \uplus U_2$ is defined. Moreover, for $(R_1 \uplus R_2)$ to be defined we require that $dom(R_1) = dom(R_2)$. There is no loss of generality in this hypothesis because if, say, $r : (U, A) \in R_1$ and $r \notin dom(R_2)$ then we can always add $r : (U_0, A)$ to R_2 where U_0 is the neutral usage of the set to which U belongs (this is left implicit in the typing rules).

One may check that the sum:

$$(r_1 : ([1, \infty], A), r_2 : ([0, 1], B))$$
$$\uplus (r_1 : ([0, \infty], A), r_2 : ([1, 0], B))$$

is equal to

$$r_1 : ([1, \infty], A), r_2 : ([1, 1], B)$$

whereas these two are not defined:

$$(R, r : ([1, \infty], B)) \uplus (R, r : ([1, \infty], B))$$
$$(R, r : ([0, \infty], B)) \uplus (R, r : ([1, 0], B))$$

2.4 Affine-Intuitionistic Type System with Regions

Because types depend on regions, we have to be careful in stating in Table 4 when a region-context and a type are compatible $(R \downarrow \alpha)$, when a region context is well-formed $(R \vdash)$, when a type is well-formed in a region context $(R \vdash \alpha)$ and when a context is well-formed in a region context $(R \vdash \Gamma)$.

A more informal way to express the condition is to say that a judgement $r_1 : (U_1, A_1), \ldots, r_n : (U_n, A_n) \vdash \alpha$ is well formed provided that:

(a) all the region names occurring in the types $A_1, \ldots, A_n, \alpha$ belong to the set $\{r_1, \ldots, r_n\}$

(b) all types of the shape $\mathsf{Reg}_{r_i} B$ with $i \in \{1, \ldots, n\}$ and occurring in the types $A_1, \ldots, A_n, \alpha$ are such that $B = A_i$.

Example 5.

One may verify that

$$r : (U, \mathbf{1} \multimap \mathbf{1}) \vdash \mathsf{Reg}_r(\mathbf{1} \multimap \mathbf{1})$$

can be derived while these judgements cannot:

$$r : (U, \mathbf{1}) \vdash \mathsf{Reg}_r(\mathbf{1} \multimap \mathbf{1})$$
$$r : (U, \mathsf{Reg}_r \mathbf{1}) \vdash \mathbf{1}$$

Next, Table 5 introduces an affine-intuitionistic type system *with regions* whose basic judgement

$$R; \Gamma \vdash P : \alpha$$

attributes a type α to the program P in the region context R and the context Γ. Here and in the following we omit the rule for typing a program $(S \mid P)$ which is symmetric to the one for the program $(P \mid S)$.

The formulation of the so called *promotion rule*, *i.e.*, the rule that introduces the '!' operator, requires some care. In particular, we notice that its formulation

$$\frac{}{R \downarrow \mathbf{1}} \qquad\qquad \frac{}{R \downarrow \mathbf{B}}$$

$$\frac{R \downarrow A \quad R \downarrow \alpha}{R \downarrow (A \multimap \alpha)} \qquad\qquad \frac{r : (U, A) \in R}{R \downarrow \mathsf{Reg}_r A}$$

$$\frac{\forall r : (U, A) \in R \quad R \downarrow A}{R \vdash} \qquad\qquad \frac{R \vdash \quad R \downarrow \alpha}{R \vdash \alpha}$$

$$\frac{\forall x : (u, A) \in \Gamma \quad R \vdash A}{R \vdash \Gamma}$$

Table 4: Type and context formation rules (unstratified)

relies on the predicates *aff* (affine) and *saff* (strongly affine) on contexts and region contexts which we define below. The intuition is that terms whose typing depends on affine (region) contexts should *not* be duplicated, *i.e.*, should not be 'marked' with a !. Formally, we write $aff(x : (u, A))$ if $u = 1$. We also write $aff(r : ([v, v'], A))$ if either $1 \in \{v, v'\}$ or $(\mathcal{V}(r)$ and $v' \neq 0)$. Moreover, we write $aff(R; \Gamma)$ (respectively $saff(R; \Gamma)$) if the predicate *aff* holds for at least one of (respectively for all) the hypotheses in $R; \Gamma$.

Remark 2. Notice that we regard the hypothesis $r : ([v, v'], A)$ as affine if either it contains the information that we can read or write in r at most once or if r is a volatile region from which we can read. The reason for the second condition is that a volatile region may contain data that should be used at most once. For instance, assuming $\mathcal{V}(r)$, $R = r : ([\infty, \infty], A)$, and $\Gamma = x : (\infty, \mathsf{Reg}_r A)$, we can derive $R; \Gamma \vdash \mathsf{get}(x) : A$. However, we should *not* derive $R; \Gamma \vdash !\mathsf{get}(x) : !A$ for otherwise the crucial subject reduction property (Theorem 1) may be compromised.

Finally, we remark that in the conclusion of the *promotion rule* we may weaken the (region) context with a strongly affine (region) context. This is essential to obtain the following weakening property.

Lemma 1 (weakening). *If $R; \Gamma \vdash P : \alpha$ and $R \uplus R' \vdash \Gamma \uplus \Gamma'$ then $R \uplus R'; \Gamma \uplus \Gamma' \vdash P : \alpha$.*

Then we see how our type system applies to some program examples.

Example 6. ⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯⎯
Let $R = r : ([1, 1], \mathbf{1})$ and

$$M = \lambda x.\mathsf{let} \ !x = x \ \mathsf{in} \ \mathsf{get}(x) \mid \mathsf{set}(x, *)$$

We check that:

$$R; \_ \vdash M : !\mathsf{Reg}_r \mathbf{1} \multimap \mathbf{B}$$

By the rule for affine implication, this reduces to:

$$R; x : (1, !\mathsf{Reg}_r \mathbf{1}) \vdash \mathsf{let} \ !x = x \ \mathsf{in} \ \mathsf{get}(x) \mid \mathsf{set}(x, *) : \mathbf{B}$$

$$\frac{R \vdash \Gamma \qquad x : (u, A) \in \Gamma}{R; \Gamma \vdash x : A} \qquad \frac{R \vdash \Gamma}{R; \Gamma \vdash * : \mathbf{1}}$$

$$\frac{R; \Gamma, x : (1, A) \vdash M : \alpha}{R; \Gamma \vdash \lambda x.M : (A \multimap \alpha)} \qquad \frac{\begin{array}{c} R_1; \Gamma_1 \vdash M : (A \multimap \alpha) \\ R_2; \Gamma_2 \vdash N : A \end{array}}{R_1 \uplus R_2; \Gamma_1 \uplus \Gamma_2 \vdash MN : \alpha}$$

$$\frac{\begin{array}{c} R \uplus R' \vdash (\Gamma \uplus \Gamma') \qquad \mathit{saff}(R'; \Gamma') \\ R; \Gamma \vdash M : A \qquad \neg \mathit{aff}(R; \Gamma) \end{array}}{R \uplus R'; \Gamma \uplus \Gamma' \vdash !M : !A} \qquad \frac{\begin{array}{c} R_1; \Gamma_1 \vdash M : !A \\ R_2; \Gamma_2, x : (\infty, A) \vdash N : \alpha \end{array}}{R_1 \uplus R_2; \Gamma_1 \uplus \Gamma_2 \vdash \mathsf{let}\ !x = M\ \mathsf{in}\ N : \alpha}$$

$$\frac{R; \Gamma, x : (u, \mathsf{Reg}_r A) \vdash P : \alpha}{R; \Gamma \vdash \nu x\ P : \alpha} \qquad \frac{\begin{array}{c} R \vdash \Gamma \qquad x : (u, \mathsf{Reg}_r A) \in \Gamma \\ r : ([v, v'], A) \in R \qquad v' \neq 0 \end{array}}{R; \Gamma \vdash \mathsf{get}(x) : A}$$

$$\frac{\begin{array}{c} \Gamma = x : (u, \mathsf{Reg}_r A) \uplus \Gamma' \qquad \mathcal{V}(r) \\ R = r : ([v, v'], A) \uplus R' \qquad v \neq 0 \\ R \vdash \Gamma \qquad R'; \Gamma' \vdash V : A \end{array}}{R; \Gamma \vdash \mathsf{set}(x, V) : \mathbf{1}} \qquad \frac{\begin{array}{c} \Gamma = x : (u, \mathsf{Reg}_r !A) \uplus \Gamma' \qquad \mathcal{P}(r) \\ R = r : ([v, v'], !A) \uplus R' \qquad v \neq 0 \\ R \vdash \Gamma \qquad R'; \Gamma' \vdash V : !A \end{array}}{R; \Gamma \vdash \mathsf{pset}(x, V) : \mathbf{1}}$$

$$\frac{\begin{array}{c} \Gamma = x : (u, \mathsf{Reg}_r A) \uplus \Gamma' \qquad \mathcal{V}(r) \\ R = r : ([v, v'], A) \uplus R' \qquad v \neq 0 \\ R \vdash \Gamma \qquad R'; \Gamma' \vdash V : A \end{array}}{R; \Gamma \vdash (x \leftarrow V) : \mathbf{B}} \qquad \frac{\begin{array}{c} \Gamma = x : (u, \mathsf{Reg}_r !A) \uplus \Gamma' \qquad \mathcal{P}(r) \\ R = r : ([v, v'], !A) \uplus R' \qquad v \neq 0 \\ R \vdash \Gamma \qquad R'; \Gamma' \vdash V : !A \end{array}}{R; \Gamma \vdash (x \Leftarrow V) : \mathbf{B}}$$

$$\frac{R_1; \Gamma_1 \vdash P : \alpha \qquad R_2; \Gamma_2 \vdash S : \mathbf{B}}{R_1 \uplus R_2; \Gamma_1 \uplus \Gamma_2 \vdash (P \mid S) : \alpha} \qquad \frac{R_i; \Gamma_i \vdash P_i : \alpha_i \qquad P_i\ \text{not a store}\ i = 1, 2}{R_1 \uplus R_2; \Gamma_1 \uplus \Gamma_2 \vdash (P_1 \mid P_2) : \mathbf{B}}$$

Table 5: An affine-intuitionistic type system with regions

If we define $R_0 = r : ([0,0], \mathbf{1})$, then by the rule for the let we reduce to:

$$R_0; x : (1, !\mathsf{Reg}_r\mathbf{1}) \vdash x : !\mathsf{Reg}_r\mathbf{1}$$

and

$$R; x : (\infty, \mathsf{Reg}_r\mathbf{1}) \vdash \mathsf{get}(x) \mid \mathsf{set}(x, *) : \mathbf{B}$$

The former is an axiom while the latter is derived from:

$$r : ([0,1], \mathbf{1}); x : (\infty, \mathsf{Reg}_r\mathbf{1}) \vdash \mathsf{get}(x) : \mathbf{1}$$

and

$$r : ([1,0], \mathbf{1}); x : (\infty, \mathsf{Reg}_r\mathbf{1}) \vdash \mathsf{set}(x, *) : \mathbf{1}$$

Note that we can actually apply the function M to a value $!y$ which is typed using the promotion rule as follows:

$$\frac{R_0; y : (\infty, \mathsf{Reg}_r\mathbf{1}) \vdash y : \mathsf{Reg}_r\mathbf{1}}{R_0; y : (\infty, \mathsf{Reg}_r\mathbf{1}) \vdash !y : !\mathsf{Reg}_r\mathbf{1}}$$

We remark that the region context and the context play two different roles: the context counts the number of occurrences of a variable while the region context counts the number of read-write effects. In our example, the variable x occurs several times but we can be sure that there will be at most one read and at most one write in the related region.

Example 7.

We consider a *functional*

$$M = \lambda f.\lambda f'.\nu y \ (fy \mid f'y)$$

which can be given the type

$$(\mathsf{Reg}_r\mathbf{1} \multimap \mathbf{1}) \multimap (\mathsf{Reg}_r\mathbf{1} \multimap \mathbf{1}) \multimap \mathbf{B}$$

in a region context $R = r : ([0,0], \mathbf{1})$. We can apply M to the functions

$$V_1 = \lambda x.\mathsf{get}(x) \text{ and } V_2 = \lambda x.\mathsf{set}(x, *)$$

which have the appropriate types in the compatible region contexts $R' = r : ([0,1], \mathbf{1})$ and $R'' = r : ([1,0], \mathbf{1})$, respectively. Such *affine* usages would not be compatible with an intuitionistic implication as in this case one has to *promote* (put a ! in front of) V_1 and V_2 before passing them as arguments.

As in Barber-Plotkin system [2], the substitution lemma comes in two flavours:

Lemma 2 (substitution)**.** *Affine substitution (a) and intuitionistic substitution (b) preserve typing:*

(a) If $R; \Gamma, x : (1, A) \vdash M : \alpha$, $R'; \Gamma' \vdash V : A$, and $R \uplus R' \vdash \Gamma \uplus \Gamma'$ then $R \uplus R'; \Gamma \uplus \Gamma' \vdash [V/x]M : \alpha$.

(b) If $R; \Gamma, x : (\infty, A) \vdash M : \alpha$, $R'; \Gamma' \vdash !V : !A$, and $R \uplus R' \vdash \Gamma \uplus \Gamma'$ then $R \uplus R'; \Gamma \uplus \Gamma' \vdash [V/x]M : \alpha$.

We rely on Lemma 2 to show that the basic reduction rules in Table 2 preserve typing. Then, observing that typing is invariant under structural equivalence, we can lift the property to the reduction relation which is generated by the basic reduction rules.

Theorem 1 (subject reduction). *If $R; \Gamma \vdash P : \alpha$ and $P \to P'$ then $R; \Gamma \vdash P' : \alpha$.*

In our formalism, a *closed* program is a program whose only free variables have region types (as in, say, the π-calculus). For *closed* programs one can state a *progress property* saying that if a program cannot progress then, up to structural equivalence, every thread is either a value or a term of the shape $E[\mathsf{get}(x)]$ and there is no store in parallel of the shape $(x \leftarrow V)$ or $(x \Leftarrow V)$. In particular, we notice that a *closed* value of type $!A$ must have the shape $!V$ so that in well-typed closed programs such as $\mathsf{let}\ !x = V\ \mathsf{in}\ M$ or $E[\mathsf{get}(x)] \mid (x \Leftarrow V)$, V is guaranteed to have the shape $!V$ required by the operational semantics in Table 2.

Proposition 1 (progress). *Suppose P is a closed typable program which cannot reduce. Then P is structurally equivalent to a program*

$$\nu x_1, \ldots, x_m\ (M_1 \mid \cdots \mid M_n \mid S_1 \mid \cdots \mid S_p) \quad m, n, p \geq 0$$

where M_i is either a value or can be uniquely decomposed as a term $E[\mathsf{get}(y)]$ such that no value is associated with the address y in the stores $S_1, \ldots, S_p$.

3 Confluence

In our language, each thread evaluates deterministically according to a call-by-value evaluation strategy. The only source of non-determinism comes from a concurrent access to the memory. More specifically, we may have a non-deterministic program if several values are stored at the same address as in the following examples (note that we cannot type a program where values are stored at an address both in a persistent and a volatile way):

$$\mathsf{get}(x) \mid (x \Leftarrow V_1) \mid (x \Leftarrow V_2) \tag{5}$$
$$\mathsf{get}(x) \mid (x \leftarrow V_1) \mid (x \leftarrow V_2) \tag{6}$$

or if there is a race condition on a volatile address as in the following example:

$$E_1[\mathsf{get}(x)] \mid E_2[\mathsf{get}(x)] \mid (x \leftarrow V) \tag{7}$$

On the other hand, a race condition on a persistent address such as:

$$E_1[\mathsf{get}(x)] \mid E_2[\mathsf{get}(x)] \mid (x \Leftarrow V) \tag{8}$$

does not compromise determinism because the two possible reductions commute. We can rule out the problematic situations (5), (6) and (7), if:

(a) we remove from our system the region usage $[\infty, \infty]$

(b) we restrict the usages of volatile stores to those in which there is at most one read effect (hence the set $\{[1,1],[1,0],[0,1],[0,0]\}$)

To this end, we add a condition $v' \neq \infty$ to the typing rules for volatile stores $\mathsf{set}(x, V)$ and $(x \leftarrow V)$ as specified in Table 6. We denote with $\vdash_C$ provability in

$$U \in \{[1,\infty],[0,\infty]\} \cup \{[1,1],[1,0],[0,1],[0,0]\}$$

$$\cfrac{\begin{array}{cc} \Gamma = x : (u, \mathsf{Reg}_r A) \uplus \Gamma' & \mathcal{V}(r) \\ R = r : ([v,v'], A) \uplus R' & v \neq 0, v' \neq \infty \\ R \vdash \Gamma & R'; \Gamma' \vdash V : A \end{array}}{R; \Gamma \vdash \mathsf{set}(x, V) : \mathbf{1}}$$

$$\cfrac{\begin{array}{cc} \Gamma = x : (u, \mathsf{Reg}_r A) \uplus \Gamma' & \mathcal{V}(r) \\ R = r : ([v,v'], A) \uplus R' & v \neq 0, v' \neq \infty \\ R \vdash \Gamma & R'; \Gamma' \vdash V : A \end{array}}{R; \Gamma \vdash (x \leftarrow V) : \mathbf{B}}$$

Table 6: Restricted usages and rules for confluence

this restricted system. This system still enjoys the subject reduction property and moreover its typable programs are strongly confluent.

Proposition 2 (subj. red. and confluence). *Suppose* $R; \Gamma \vdash_C P : \alpha$. *Then:*

(a) If $P \to P'$ *then* $R; \Gamma \vdash_C P' : \alpha$

(b) If $P \to P'$ *and* $P \to P''$ *then either* $P' \equiv P''$ *or there is a* Q *such that* $P' \to Q'$ *and* $P'' \to Q$

Proof.

(a) We just have to reconsider the case where $E[\mathsf{set}(x, V)] \to E[*] \mid (x \leftarrow V)$ and verify that if $R; \Gamma \vdash \mathsf{set}(x, V) : \mathbf{1}$ then $R; \Gamma \vdash (x \leftarrow V) : \mathbf{B}$ which entails that $E[*] \mid (x \leftarrow V)$ is typable in the same context as $E[\mathsf{set}(x, V)]$.

(b) The restrictions on the usages forbid the typing of a store such as the one in (5) and (6) where two values are stored in the same region. Moreover, it also forbids the typing of two parallel reads on a volatile store (7).

$\square$

Remark 3. We note that the rules for ensuring confluence require that at most one value is associated with a region (single-assignment). This is quite a restrictive discipline (comparable to the one in [14]) but one has to keep in mind that it targets regions that can be accessed concurrently by several threads. Of course, the discipline could be relaxed for the regions that are accessed by

one single sequential thread. Also, *e.g.*, for optimisation purposes, one may be interested in the confluence/determinism of certain reductions even when the overall program is non-deterministic.

4 An Affine-Intuitionistic Type and Effect System

We refine the type system to include *effects* which are denoted with $e, e', \ldots$ and are finite sets of regions. The syntax of programs (Table 1) and their operational semantics (Table 2) are unchanged. The only modification to the syntax of types (Table 3) is that the affine implication is now annotated with an effect so that we write:

$$A \overset{e}{\multimap} \alpha$$

which is the type of a function that when given a value of type A may produce something of type α and an effect on the regions in e. This introduces a new dependency of types on regions and consequently the compatibility condition between region contexts and functional types in Table 4 becomes:

$$\frac{R \downarrow A \quad R \downarrow \alpha \quad e \subseteq dom(R)}{R \downarrow (A \overset{e}{\multimap} \alpha)}$$

Example 8. ⎯⎯⎯
One may verify that the judgement

$$r : (U, \mathbf{1} \overset{\{r\}}{\multimap} \mathbf{1}) \vdash$$

is derivable.
⎯⎯⎯

The typing judgements now take the shape

$$R; \Gamma \vdash P : (\alpha, e)$$

where the effect e provides an upper bound on the set of regions on which the program P may read or write when it is evaluated. In particular, we can be sure that values and stores produce an empty effect. As for the operations to read and write the store, one exploits the dependency of address types on regions to determine the region where the effect occurs (cf. [15]).

The affine-intuitionistic type and effect system is spelled out in Table 7. We stress that these rules are the same as the ones in Table 5 modulo the enriched syntax of the functional types and the management of the effect e on the right hand side of the sequents. The management the effects is *additive* as in [15], indeed effects are just *sets* of regions.

Also to allow for some flexibility, it is convenient to introduce a subtyping relation on types and effects, that is to say on pairs (α, e), as specified in Table 8. We notice that the *transitivity rule* for subtyping

14

$$\frac{R \vdash \alpha \leq \alpha' \qquad R \vdash \alpha' \leq \alpha''}{R \vdash \alpha \leq \alpha''}$$

can be derived via a simple induction on the height of the proofs.

Remark 4. The introduction of the subtyping rules has a limited impact on the structure of the typing proofs. Indeed, if $R \vdash A \leq B$ then we know that A and B may just differ in the effects annotating the functional types. In particular, when looking at the proof of the typing judgement of a value such as $R; \Gamma \vdash \lambda x.M : (A, e)$, we can always argue that A has the shape $A_1 \xrightarrow{e_1} A_2$ and, in case the effect e is not empty, that there is a shorter proof of the judgement $R; \Gamma \vdash \lambda x.M : (B_1 \xrightarrow{e_2} B_2, \emptyset)$ where $R \vdash A_1 \leq B_1$, $R \vdash B_2 \leq A_2$, and $e_2 \subseteq e_1$.

Then to prove subject reduction, we just repeat the proof of Theorem 1 while using standard arguments to keep track of the effects.

Proposition 3 (subject reduction with effects). *Types and effects are preserved by reduction.*

Remark 5. It is easy to check that a typable program such as $E[\mathsf{set}(x, V)]$ which is ready to produce an effect on the region r associated with x will indeed contain r in its effect. Thus the subject reduction property stated above as Proposition 3 entails that the type and effect system does provide an upper bound on the effects a program may produce during its evaluation.

5 Termination

Terms typable in the unstratified type and effect system (cf. Table 7) may diverge, as exemplified here:

Example 9. ───────────────────────────────────
 The following term stores at the address x a function that, given an argument, keeps fetching itself from the store forever:

$$\nu x \; \mathsf{pset}(x, !(\lambda y.\mathsf{let} \; !x = \mathsf{get}(x) \; \mathsf{in} \; xy)) \; ; \; \mathsf{let} \; !x = \mathsf{get}(x) \; \mathsf{in} \; x*$$

One may verify that it is typable in a region context

$$R = r : ([1, \infty], !(\mathbf{1} \xrightarrow{\{r\}} \mathbf{1}))$$

───

This example suggests that in order to recover termination, we may order regions and make sure that a value stored in a certain region when put in an evaluation context can only produce effects on smaller regions. This is where our type and effect system comes into play, and to formalise this idea, we introduce in Table 9 rules for the formation of types and contexts which are alternative to those in Table 4.

Example 10. ─────────────────────────────────────

Assuming Table 9 and taking $R = r : (U, \mathbf{1})$, one may check that the judgement

$$r : (U, \mathbf{1}), r' : (U', \mathbf{1} \xrightarrow{\{r\}} \mathbf{1}) \vdash$$

is derivable while

$$r' : (U', \mathbf{1} \xrightarrow{\{r'\}} \mathbf{1}) \vdash$$

is *not*. In particular, the region context of Example 9 is neither derivable.

───

It is easy to verify that the stratified system is a restriction of the unstratified one and that the subject reduction (Proposition 3) still holds in the restricted stratified system. If confluence is required, then one may add the restrictions spelled out in Table 6.

Concerning termination, we recall that there is a standard forgetful translation (_) from affine-intuitionistic logic to intuitionistic logic which amounts to forget about the modality ! and the usages and to regard the affine implication as an ordinary intuitionistic implication. Thus, for instance, the translation of types goes as follows: $\underline{!A} = \underline{A}$ and $\underline{A \multimap B} = \underline{A} \to \underline{B}$; while the translation of terms is: $\underline{!M} = \underline{M}$ and $\underline{\text{let } !x = M \text{ in } N} = (\lambda x.\underline{N})\underline{M}$. In Table 10, we lift this translation from the stratified *affine-intuitionistic* type and effect system into a stratified *intuitionistic* type and effect system defined in [1].

The translation assumes a *decoration phase* where the (free or bound) variables with a region type of the shape $\mathsf{Reg}_r A$ are labelled with the region r. Intuitively, the intuitionistic system abstracts an address x related to the region r to the region r itself so that a decorated variable x^r translates into a constant r. In the intuitionistic language, a region r is a term of region type $\mathsf{Reg}_r A$, for some A and all stores are persistent. The full definition of the language is recalled in Appendix A.2.

It turns out that a reduction in the affine-intuitionistic system is mapped to exactly a reduction in the intuitionistic system. Then the termination of the intuitionistic system proved in [1] entails the termination of the affine-intuitionistic one.

Theorem 2 (termination). *Programs typable in the stratified affine-intuitionistic type and effect system terminate.*

6 Conclusion

We have presented an affine-intuitionistic system of types and effects for a functional-concurrent programming language. The main contribution over [1] is that the functional core of the system is based on Barber-Plotkin affine-intuitionistic logic which distinguishes between affine and intuitionistic hypotheses. The 'non-logical' part of the language, with operators to read and write dynamically generated addresses of a 'store', has been refined to take into account the process of data duplication. In the type system, addresses are abstracted

into a finite number of *regions*. We have introduced a suitable discipline of region *usage* and shown that it combines with region *stratification* in the affine-intuitionistic setting to regain *confluence* and *termination*, respectively.

Future Work   Beyond these crucial properties, we hope to show that other interesting properties of the functional core can be extended to the considered framework. We think in particular of the construction of denotational models (see, *e.g*, [5]) and of bounds on the computational complexity of typable programs (see, *e.g.*, [11]).

We also recall that more work would be required to get an operational programming language, as with the introduction of inductive types and the extension to a synchronous/timed framework (cf. [1, 6]) where determinism and termination are useful properties.

7   Acknowledgements

References

[1] R.M. Amadio. On stratified regions. In Proc. *APLAS*, Springer LNCS, 2009. Extended version available as arXiv:0904.2076v2.

[2] A. Barber. Dual intuitionistic linear logic. *University of Edinburgh*, Technical report ECS-LFCS-96-347, 1996.

[3] N. Benton, G. Bierman, V. de Paiva and M. Hyland. A term calculus for intuitionistic linear logic. In Proc. *Typed Lambda Calculi and Applications*, Springer LNCS 664:75-90, 2003.

[4] N. Benton. A mixed linear and non-linear logic; proofs, terms and models. In Proc. *Computer Science Logic*, Springer LNCS 933:121-135, 2004.

[5] G. Bierman. What is a categorical model of intuitionistic linear logic? In Proc. *Typed Lambda-Calculi and Applications*, Springer LNCS 902:78-93, 1995.

[6] G. Boudol. Typing termination in a higher-order concurrent imperative language. In Proc. *CONCUR*, Springer LNCS 4703:272-286, 2007.

[7] M. Fluet, G. Morrisett, and A. Ahmed. Linear regions are all you need. In Proc. *ESOP*, Springer LNCS 3924: 7-21, 2006.

[8] A. Giacalone, P. Mishra, and S. Prasad. FACILE: A symmetric integration of concurrent and functional programming. In Proc. *TAPSOFT*, Springer LNCS 352:184-209, 1989.

[9] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1-102, 1987.

[10] J.-Y. Girard. On the unity of logic. *Ann. Pure Appl. Logic*, 59(3):201-217, 1993.

[11] J.-Y. Girard. Light linear logic. *Information and Computation*, 143(2): 175-204, 1998.

[12] A. Igarashi and N. Kobayashi. Resource usage analysis. *ACM Trans. Program. Lang. Syst.* 27(2): 264-313, 2005.

[13] N. Kobayashi. Type systems for concurrent programs. In Proc. *10th Anniversary Colloquium of UNU/IIST*, Springer LNCS 2757:439-453, 2003.

[14] N. Kobayashi, B. Pierce, and D. Turner. Linearity and the pi-calculus. *ACM Trans. on Program. Lang. and Systems*, 21(5):914-947, 1999.

[15] J. Lucassen and D. Gifford. Polymorphic effect systems. In Proc. *ACM-POPL*, pp 47-57, 1988.

[16] J. Maraist, M. Odersky, D. Turner, and Ph. Wadler. Call-by-name, call-by-value, call-by-need, and the linear lambda calculus. In Proc. *Mathematical Foundations of Programming Semantics*, Elect. Notes in Comp. Sci. 1(1), Elsevier, 1995.

[17] G. Plotkin. Type theory and recursion. In Proc. IEEE-LICS, Abstract, 1993.

[18] J. Reppy. CML: A higher-order concurrent language. In Proc. *ACM-SIGPLAN Conf. on Prog. Language Design and Implementation*, pp 293-305, 1991.

[19] D. Sangiorgi. Termination of processes. *Math. Struct. in Comp. Sci.*, 16:1-39, 2006.

[20] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2): 109-176, 1997.

[21] Ph. Wadler. A Taste of Linear Logic. In Proc. *Mathematical Foundations of Computer Science*, Springer LNCS 711:185-210, 1993.

[22] D. Walker. Substructural type systems. Chapter 1 of *Advanced topics in types and programming languages*, B. Pierce (ed.), MIT Press, 2002.

[23] D. Walker and K.Watkins. On Regions and Linear Types. In Proc. *Int. Conf. on Fun. Prog.*, pp 181-192,2001.

[24] N. Yoshida, M. Berger, and K. Honda. Strong normalisation in the π-calculus. *Information and Computation*, 191(2):145-202, 2004.

A Proofs

A.1 Proof of Theorem 1

Lemma 3 (weakening). *If $R; \Gamma \vdash P : \alpha$ and $R \uplus R' \vdash \Gamma \uplus \Gamma'$ then $R \uplus R'; \Gamma \uplus \Gamma' \vdash P : \alpha$.*

Proof. By induction on the typing of P. Following Table 5, there are 14 rules to be considered of which we highlight 3.

$P \equiv MN$ We have:

$$\frac{R_1; \Gamma_1 \vdash M : A \multimap \alpha \qquad R_2; \Gamma_2 \vdash N : A}{R_1 \uplus R_2; \Gamma_1 \uplus \Gamma_2 \vdash MN : \alpha} \ .$$

We notice that the composition operation $\uplus$ on contexts is associative and commutative (when it is defined) and that $(R_1 \uplus R_2 \uplus R') \vdash (\Gamma_1 \uplus \Gamma_2 \uplus \Gamma')$ entails that $(R_1 \uplus R') \vdash (\Gamma_1 \uplus \Gamma')$. Hence, by induction hypothesis, we get $R_1 \uplus R'; \Gamma_1 \uplus \Gamma' \vdash M : A \multimap \alpha$, from which we derive:

$$\frac{R_1 \uplus R'; \Gamma_1 \uplus \Gamma' \vdash M : A \multimap \alpha \qquad R_2; \Gamma_2 \vdash N : A}{R_1 \uplus R_2 \uplus R'; \Gamma_1 \uplus \Gamma_2 \uplus \Gamma' \vdash MN : \alpha} \ .$$

$P \equiv\ !M$ We have:

$$\frac{\begin{array}{c} R \uplus R'' \vdash \Gamma \uplus \Gamma'' \\ \mathit{saff}(R''; \Gamma'') \\ \neg \mathit{aff}(R; \Gamma) \qquad R; \Gamma \vdash M : A \end{array}}{R \uplus R''; \Gamma \uplus \Gamma'' \vdash\ !M :\ !A} \ .$$

We can always decompose R' as $R_1' \uplus R_\infty'$ and Γ' as $\Gamma_1' \uplus \Gamma_\infty'$ so that $\neg \mathit{aff}(R_\infty'; \Gamma_\infty')$ and $\mathit{saff}(R_1'; \Gamma_1')$. By induction hypothesis, we have $R \uplus R_\infty'; \Gamma \uplus \Gamma_\infty' \vdash M : A$. We notice that $\neg \mathit{aff}(R \uplus R_\infty'; \Gamma \uplus \Gamma_\infty')$ and $\mathit{saff}(R_1' \uplus R''; \Gamma_1' \uplus \Gamma'')$ (remember that $1 \uplus \infty$ is undefined). Hence we derive:

$$\frac{\begin{array}{c} (R \uplus R_\infty' \uplus R_1' \uplus R'') \vdash (\Gamma \uplus \Gamma_\infty' \uplus \Gamma_1' \uplus \Gamma'') \\ \mathit{saff}(R_1' \uplus R''; \Gamma_1' \uplus \Gamma'') \\ \neg \mathit{aff}(R \uplus R_\infty'; \Gamma \uplus \Gamma_\infty') \qquad R \uplus R_\infty'; \Gamma \uplus \Gamma_\infty' \vdash M : A \end{array}}{R \uplus R' \uplus R''; \Gamma \uplus \Gamma' \uplus \Gamma'' \vdash\ !M :\ !A} \ .$$

$P \equiv \mathsf{set}(x, V)$ We have:

$$\frac{\begin{array}{c} \Gamma = x : (u, \mathsf{Reg}_r A) \uplus \Gamma'' \\ R = r : ([v, v'], A) \uplus R'' \qquad v \neq 0 \\ R \vdash \Gamma \qquad R''; \Gamma'' \vdash V : A \end{array}}{R; \Gamma \vdash \mathsf{set}(x, V) : \mathbf{1}} \ .$$

By induction hypothesis, we have $R'' \uplus R'; \Gamma'' \uplus \Gamma' \vdash V : A$, from which we derive:

$$\frac{\begin{array}{c} \Gamma \uplus \Gamma' = x : (u, \mathsf{Reg}_r A) \uplus (\Gamma'' \uplus \Gamma') \\ R \uplus R' = r : ([v, v'], A) \uplus (R'' \uplus R') \qquad v \neq 0 \\ R \uplus R' \vdash \Gamma \uplus \Gamma' \qquad R'' \uplus R'; \Gamma'' \uplus \Gamma' \vdash V : A \end{array}}{R \uplus R'; \Gamma \uplus \Gamma' \vdash \mathsf{set}(x, V) : \mathbf{1}} \ .$$

We notice that this argument still holds when introducing the restriction $v' \neq \infty$ in order to guarantee confluence (cf. Table 6). Indeed, the restriction $v' \neq \infty$ is equivalent to require that the usage of the region r ranges in the family of usages $\{[1, 1], [1, 0], [0, 1], [0, 0]\}$.

$\square$

Lemma 4 (affine substitution lemma). *If* $R_1; \Gamma_1, x : (1, A) \vdash P : \alpha$, $R_2; \Gamma_2 \vdash V : A$, *and* $R_1 \uplus R_2 \vdash \Gamma_1 \uplus \Gamma_2$ *then* $R_1 \uplus R_2; \Gamma_1 \uplus \Gamma_2 \vdash [V/x]P : \alpha$.

Proof. By induction on the typing of P. We highlight 4 cases out of 14.

$P \equiv MN$ We have:

$$\frac{R_3; \Gamma_3' \vdash M : C \multimap \alpha \qquad R_4; \Gamma_4' \vdash N : C}{R_3 \uplus R_4; \Gamma_3' \uplus \Gamma_4' \vdash MN : \alpha} \ .$$

Because $x : (1, A)$ is an affine hypothesis, it can occur exclusively either in Γ_3' or in Γ_4'. We consider both cases.

1. $\Gamma_3' = \Gamma_3, x : (1, A)$ and $\Gamma_4' = \Gamma_4$ with $x \notin dom(\Gamma_4)$. By induction hypothesis we have $R_2 \uplus R_3; \Gamma_2 \uplus \Gamma_3 \vdash [V/x]M : C \multimap \alpha$. Plus $x \notin FV(N)$ so $[V/x]N \equiv N$, hence $R_4; \Gamma_4 \vdash [V/x]N : C$. Then we derive:

$$\frac{R_2 \uplus R_3; \Gamma_2 \uplus \Gamma_3 \vdash [V/x]M : C \multimap \alpha \qquad R_4; \Gamma_4 \vdash [V/x]N : C}{R_2 \uplus R_3 \uplus R_4; \Gamma_2 \uplus \Gamma_3 \uplus \Gamma_4 \vdash [V/x](MN) : \alpha} \ .$$

2. $\Gamma_3' = \Gamma_3$ with $x \notin dom(\Gamma_3)$ and $\Gamma_4' = \Gamma_4, x : (1, A)$.
   By induction hypothesis we have $R_2 \uplus R_4; \Gamma_2 \uplus \Gamma_4 \vdash [V/x]N : C$. Plus $x \notin FV(M)$ so $[V/x]M \equiv M$, hence $R_3; \Gamma_3 \vdash [V/x]M : C \multimap \alpha$. Then we derive:

$$\frac{R_3; \Gamma_3 \vdash [V/x]M : C \multimap \alpha \qquad R_2 \uplus R_4; \Gamma_2 \uplus \Gamma_4 \vdash [V/x]N : C}{R_2 \uplus R_3 \uplus R_4; \Gamma_2 \uplus \Gamma_3 \uplus \Gamma_4 \vdash [V/x](MN) : \alpha} \ .$$

$P \equiv !M$ We have:

$$\frac{R_1 \uplus R' \vdash (\Gamma_1 \uplus (\Gamma', x : (1, A))) \quad saff(R'; \Gamma', x : (1, A)) \quad R_1; \Gamma_1 \vdash M : A \quad \neg aff(R_1; \Gamma_1)}{R_1 \uplus R'; \Gamma_1 \uplus (\Gamma', x : (1, A)) \vdash !M : !A}$$

We deduce that $x \notin FV(!M)$, hence $[V/x](!M) \equiv !M$ and $R_1 \uplus R'; \Gamma_1 \uplus \Gamma' \vdash [V/x](!M) : !A$. By lemma 3, we get $R_1 \uplus R' \uplus R_2; \Gamma_1 \uplus \Gamma' \uplus \Gamma_2 \vdash [V/x](!M) : !A$.

21

$P \equiv \mathsf{let}\ !y = M\ \mathsf{in}\ N$ Renaming y so that $y \neq x$, we have:

$$\frac{R_3; \Gamma_3' \vdash M : !C \qquad R_4; \Gamma_4', y : (\infty, C) \vdash N : \alpha}{R_3 \uplus R_4; \Gamma_3' \uplus \Gamma_4' \vdash \mathsf{let}\ !y = M\ \mathsf{in}\ N : \alpha}$$

As in the case of application, we distinguish two cases.

1. $\Gamma_3' = \Gamma_3, x : (1, A)$ and $\Gamma_4' = \Gamma_4$ with $x \notin dom(\Gamma_4)$.
   By induction hypothesis, we have $R_2 \uplus R_3; \Gamma_2 \uplus \Gamma_3 \vdash [V/x]M : !C$.
   Plus $x \notin FV(N)$ so $[V/x]N \equiv N$, hence $R_4; \Gamma_4, y : (\infty, C) \vdash [V/x]N : \alpha$. Then we derive:

   $$\frac{\begin{array}{c} R_2 \uplus R_3; \Gamma_2 \uplus \Gamma_3 \vdash [V/x]M : !C \\ R_4; \Gamma_4, y : (\infty, C) \vdash [V/x]N : \alpha \end{array}}{R; \Gamma_2 \uplus \Gamma_3 \uplus \Gamma_4 \vdash [V/x](\mathsf{let}\ !y = M\ \mathsf{in}\ N) : \alpha} \ .$$

   where $R = R_2 \uplus R_3 \uplus R_4$.

2. $\Gamma_3' = \Gamma_3$ with $x \notin dom(\Gamma_3)$ and $\Gamma_4' = \Gamma_4, x : (1, A)$.
   By induction hypothesis we have $R_2 \uplus R_4; \Gamma_2, y : (\infty, C) \uplus \Gamma_4 \vdash [V/x]N : \alpha$. Plus $x \notin FV(M)$ so $[V/x]M \equiv M$, hence $R_3; \Gamma_3 \vdash [V/x]M : !C$. Then we derive:

   $$\frac{\begin{array}{c} R_3; \Gamma_3 \vdash [V/x]M : !C \\ R_2 \uplus R_4; \Gamma_2, y : (\infty, C) \uplus \Gamma_4 \vdash [V/x]N : \alpha \end{array}}{R; \Gamma_2 \uplus \Gamma_3 \uplus \Gamma_4 \vdash [V/x](\mathsf{let}\ !y = M\ \mathsf{in}\ N) : \alpha} \ .$$

   where $R = R_2 \uplus R_3 \uplus R_4$.

$P \equiv \mathsf{set}(y, V')$ We distinguish two cases.

1. If $y \neq x$ we have:

   $$\frac{\begin{array}{c} \Gamma_1, x : (1, A) = y : (u, \mathsf{Reg}_r C) \uplus \Gamma_1' \\ R_1 = r : ([v, v'], C) \uplus R_1' \qquad v \neq 0 \\ R_1 \vdash \Gamma_1, x : (1, A) \qquad R_1'; \Gamma_1' \vdash V' : C \end{array}}{R_1; \Gamma_1, x : (1, A) \vdash \mathsf{set}(y, V') : \mathbf{1}} \ .$$

   We deduce that $\Gamma_1' = \Gamma_1'' \uplus x : (1, A)$, and by induction hypothesis we get $R_1' \uplus R_2; \Gamma_1'' \uplus \Gamma_2 \vdash [V/x]V' : C$, from which we derive:

   $$\frac{\begin{array}{c} \Gamma_1 = y : (u, \mathsf{Reg}_r C) \uplus \Gamma_1'' \\ R_1 = r : ([v, v'], C) \uplus R_1' \qquad v \neq 0 \\ R_1 \vdash \Gamma_1 \qquad R_1' \uplus R_2; \Gamma_1'' \uplus \Gamma_2 \vdash [V/x]V' : C \end{array}}{R_1; \Gamma_1 \vdash [V/x]\mathsf{set}(y, V') : \mathbf{1}} \ .$$

   By lemma 3, we obtain

   $$R_1 \uplus R_2; \Gamma_1 \uplus \Gamma_2 \vdash [V/x]\mathsf{set}(y, V') : \mathbf{1}$$

   .

2. If $y = x$ then $[V/x]\mathsf{set}(y, V') \equiv \mathsf{set}(V, V')$, $A = \mathsf{Reg}_r C$, and $u = 1$. Moreover V must be a variable, thus we can derive:

$$\cfrac{\Gamma_1 = V : (1, \mathsf{Reg}_r C) \uplus \Gamma_1' \\ R_1 = r : ([v, v'], C) \uplus R_1' \quad v \neq 0 \\ \cfrac{R_1 \vdash \Gamma_1 \qquad R_1'; \Gamma_1' \vdash V' : C}{R_1; \Gamma_1 \vdash [V/x]\mathsf{set}(y, V') : \mathbf{1}}}{} \ ,$$

and by lemma 3 we get

$$R_1 \uplus R_2; \Gamma_1 \uplus \Gamma_2 \vdash [V/x]\mathsf{set}(y, V') : \mathbf{1}$$

.

$\square$

Lemma 5 (intuitionistic substitution lemma). *If*
$R_1; \Gamma_1, x : (\infty, A) \vdash P : \alpha$, $R_2; \Gamma_2 \vdash\ !V :\ !A$, *and* $R_1 \uplus R_2 \vdash \Gamma_1 \uplus \Gamma_2$ *then*
$R_1 \uplus R_2; \Gamma_1 \uplus \Gamma_2 \vdash [V/x]P : \alpha$.

Proof. By induction on the typing of P. We highlight 4 cases out of 14.

$P \equiv MN$ We have:

$$\cfrac{R_3; \Gamma_3' \vdash M : C \multimap \alpha \qquad R_4; \Gamma_4' \vdash N : C}{R_3 \uplus R_4; \Gamma_3' \uplus \Gamma_4' \vdash MN : \alpha} \ .$$

We distinguish 3 cases.

1. $\Gamma_3' = \Gamma_3, x : (\infty, A)$ and $\Gamma_4' = \Gamma_4$ with $x \notin dom(\Gamma_4)$.
   By induction hypothesis we have $R_2 \uplus R_3; \Gamma_2 \uplus \Gamma_3 \vdash [V/x]M : C \multimap \alpha$. Plus $x \notin FV(N)$ so $[V/x]N \equiv N$, hence $R_4; \Gamma_4 \vdash [V/x]N : C$. Then we derive:

$$\cfrac{R_2 \uplus R_3; \Gamma_2 \uplus \Gamma_3 \vdash [V/x]M : C \multimap \alpha \\ R_4; \Gamma_4 \vdash [V/x]N : C}{R_2 \uplus R_3 \uplus R_4; \Gamma_2 \uplus \Gamma_3 \uplus \Gamma_4 \vdash [V/x](MN) : \alpha} \ .$$

2. $\Gamma_3' = \Gamma_3$ with $x \notin dom(\Gamma_3)$ and $\Gamma_4' = \Gamma_4, x : (\infty, A)$.
   By induction hypothesis we have $R_2 \uplus R_4; \Gamma_2 \uplus \Gamma_4 \vdash [V/x]N : C$. Plus $x \notin FV(M)$ so $[V/x]M \equiv M$, hence $R_3; \Gamma_3 \vdash [V/x]M : C \multimap \alpha$. Then we derive:

$$\cfrac{R_3; \Gamma_3 \vdash [V/x]M : C \multimap \alpha \\ R_2 \uplus R_4; \Gamma_2 \uplus \Gamma_4 \vdash [V/x]N : C}{R_2 \uplus R_3 \uplus R_4; \Gamma_2 \uplus \Gamma_3 \uplus \Gamma_4 \vdash [V/x](MN) : \alpha} \ .$$

3. $\Gamma'_3 = \Gamma_3, x : (\infty, A)$ and $\Gamma'_4 = \Gamma_4, x : (\infty, A)$.

By induction hypothesis we have $R_2 \uplus R_3; \Gamma_2 \uplus \Gamma_3 \vdash [V/x]M : C \multimap \alpha$ and $R_2 \uplus R_4; \Gamma_2 \uplus \Gamma_4 \vdash [V/x]N : C$. Moreover we have:

$$\frac{R_5 \uplus R' \vdash \Gamma_5 \uplus \Gamma' \quad \mathit{saff}(R'; \Gamma') \quad R_5; \Gamma_5 \vdash V : A \quad \neg\mathit{aff}(R_5; \Gamma_5)}{R_2; \Gamma_2 \vdash\, !V :\, !A} \quad,$$

where $R_2 = R_5 \uplus R'$ and $\Gamma_2 = \Gamma_5 \uplus \Gamma'$. Hence we know that all the hypotheses of R' and Γ' are of weakened regions and variables. Thus we also have $R_3 \uplus R_5; \Gamma_3 \uplus \Gamma_5 \vdash [V/x]M : C \multimap \alpha$ and $R_4 \uplus R_5; \Gamma_4 \uplus \Gamma_5 \vdash [V/x]N : C$. Plus from $\neg\mathit{aff}(R_5; \Gamma_5)$ we get $R_5 \uplus R_5 = R_5$ and $\Gamma_5 \uplus \Gamma_5 = \Gamma_5$, and we can derive:

$$\frac{R_3 \uplus R_5; \Gamma_3 \uplus \Gamma_5 \vdash [V/x]M : C \multimap \alpha \quad R_4 \uplus R_5; \Gamma_4 \uplus \Gamma_5 \vdash [V/x]N : C}{R_3 \uplus R_4 \uplus R_5; \Gamma_3 \uplus \Gamma_4 \uplus \Gamma_5 \vdash [V/x](MN) : \alpha} \quad.$$

By lemma 3 we obtain $R_2 \uplus R_3 \uplus R_4; \Gamma_2 \uplus \Gamma_3 \uplus \Gamma_4 \vdash [V/x](MN) : \alpha$.

$P \equiv\, !M$ Suppose:

$$\frac{R_5 \uplus R' \vdash (\Gamma_5, x : (\infty, A)) \uplus \Gamma' \quad \mathit{saff}(R'; \Gamma') \quad R_5; \Gamma_5, x : (\infty, A) \vdash M : B \quad \neg\mathit{aff}(R_5; \Gamma_5, x : (\infty, A))}{R_5 \uplus R'; (\Gamma_5, x : (\infty, A)) \uplus \Gamma' \vdash\, !M :\, !B} \quad.$$

And also:

$$\frac{R_6 \uplus R_7 \vdash \Gamma_6 \uplus \Gamma_7 \quad \mathit{saff}(R_7; \Gamma_7) \quad \mathit{aff}(R_6; \Gamma_6) \quad R_6; \Gamma_6 \vdash V : A}{R_2; \Gamma_2 \vdash\, !V :\, !A} \quad,$$

with $R_2 = R_6 \uplus R_7$ and $\Gamma_2 = \Gamma_6 \uplus \Gamma_7$. Hence we know that all the hypotheses of R_7 and Γ_7 are of weakened regions and variables, such that $R_6; \Gamma_6 \vdash\, !V :\, !A$. By induction hypothesis we get $R_5 \uplus R_6; \Gamma_5 \uplus \Gamma_6 \vdash [V/x]M : B$ and we can derive:

$$\frac{(R_5 \uplus R_6) \uplus (R_7 \uplus R') \vdash (\Gamma_5 \uplus \Gamma_6) \uplus (\Gamma_7 \uplus \Gamma') \quad \mathit{saff}(R_7 \uplus R'; \Gamma_7 \uplus \Gamma') \quad \neg\mathit{aff}(R_5 \uplus R_6; \Gamma_5 \uplus \Gamma_6) \quad R_5 \uplus R_6; \Gamma_5 \uplus \Gamma_6 \vdash [V/x]M : B}{R_5 \uplus R_2 \uplus R'; \Gamma_5 \uplus \Gamma_2 \uplus \Gamma' \vdash [V/x]!M :\, !B} \quad.$$

$P \equiv\, \mathsf{let}\ !y = M\ \mathsf{in}\ N$ We have:

$$\frac{R_3; \Gamma'_3 \vdash M :\, !C \quad R_4; \Gamma'_4, y : (\infty, C) \vdash N : \alpha}{R_3 \uplus R_4; \Gamma'_3 \uplus \Gamma'_4 \vdash \mathsf{let}\ !y = M\ \mathsf{in}\ N : \alpha} \quad.$$

with $y \neq x$. We just spell out the case where $\Gamma'_3 = \Gamma_3, x : (\infty, A)$ and $\Gamma'_4 = \Gamma_4, x : (\infty, A)$. By induction hypothesis, we have $R_2 \uplus R_3; \Gamma_2 \uplus \Gamma_3 \vdash$

24

$[V/x]M : {!}C$ and $R_2 \uplus R_4; (\Gamma_2, y : (\infty, C)) \uplus \Gamma_4 \vdash [V/x]N : \alpha$. Moreover we have:

$$\frac{\begin{array}{cc} R_5 \uplus R' \vdash \Gamma_5 \uplus \Gamma' & \mathit{saff}(R'; \Gamma') \\ R_5; \Gamma_5 \vdash V : A & \neg\mathit{aff}(R_5; \Gamma_5) \end{array}}{R_2; \Gamma_2 \vdash {!}V : {!}A} \ ,$$

where $\Gamma_2 = \Gamma_5 \uplus \Gamma'$ and $R_2 = R_5 \uplus R'$. Hence we know that all the hypotheses of R' and Γ' are of weakened regions and variables. Thus we also have $R_3 \uplus R_5; \Gamma_3 \uplus \Gamma_5 \vdash [V/x]M : {!}C$ and $R_4 \uplus R_5; (\Gamma_4, y : (\infty, C)) \uplus \Gamma_5 \vdash [V/x]N : \alpha$. Plus from $\neg\mathit{aff}(R_5; \Gamma_5)$ we get $\Gamma_5 \uplus \Gamma_5 = \Gamma_5$ and $R_5 \uplus R_5 = R_5$, and we can derive:

$$\frac{\begin{array}{c} R_3 \uplus R_5; \Gamma_3 \uplus \Gamma_5 \vdash [V/x]M : {!}C \\ R_4 \uplus R_5; (\Gamma_4, y : (\infty, C)) \uplus \Gamma_5 \vdash [V/x]N : \alpha \end{array}}{R_3 \uplus R_4 \uplus R_5; \Gamma_3 \uplus \Gamma_4 \uplus \Gamma_5 \vdash [V/x](\mathsf{let}\ {!}y = M\ \mathsf{in}\ N) : \alpha} \ .$$

By lemma 3, we obtain $R_2 \uplus R_3 \uplus R_4; \Gamma_2 \uplus \Gamma_3 \uplus \Gamma_4 \vdash [V/x](\mathsf{let}\ {!}y = M\ \mathsf{in}\ N) : \alpha$.

$P \equiv \mathsf{set}(y, V')$ We just look at the case $y \neq x$. We have:

$$\frac{\begin{array}{cc} \Gamma_1, x : (\infty, A) = y : (u, \mathsf{Reg}_r C) \uplus \Gamma'_1 \\ R_1 = r : ([v, v'], C) \uplus R'_1 & v' \neq 0 \\ R_1 \vdash \Gamma_1, x : (\infty, A) & R'_1; \Gamma'_1 \vdash V' : C \end{array}}{R_1; \Gamma_1, x : (\infty, A) \vdash \mathsf{set}(y, V') : \mathbf{1}} \ .$$

We deduce that $\Gamma'_1 = \Gamma''_1 \uplus x : (\infty, A)$, and by induction hypothesis we get $R'_1 \uplus R_2; \Gamma''_1 \uplus \Gamma_2 \vdash [V/x]V' : C$, from which we derive:

$$\frac{\begin{array}{cc} \Gamma_1 = y : (u, \mathsf{Reg}_r C) \uplus \Gamma''_1 \\ R_1 = r : ([v, v'], C) \uplus R'_1 & v' \neq 0 \\ R_1 \vdash \Gamma_1 & R'_1 \uplus R_2; \Gamma''_1 \uplus \Gamma_2 \vdash [V/x]V' : C \end{array}}{R_1 \uplus R_2; \Gamma_1 \uplus \Gamma_2 \vdash [V/x]\mathsf{set}(y, V') : \mathbf{1}} \ .$$

$\square$

Lemma 6 (structural equivalence preserves typing). *If $R; \Gamma \vdash P : \alpha$ and $P \equiv P'$ then $R; \Gamma \vdash P' : \alpha$.*

Proof. Recall that structural equivalence is the least equivalence relation induced by the equations stated in Table 2 and closed under static contexts. Then we proceed by induction on the proof of structural equivalence. This is is mainly a matter of reordering the pieces of the typing proof of P so as to obtain a typing proof of P'. $\square$

Lemma 7 (evaluation contexts and typing). *Suppose that in the proof of $R; \Gamma \vdash E[M] : \alpha$ we prove $R'; \Gamma' \vdash M : A$. Then replacing M with a M' such that $R'; \Gamma' \vdash M' : A$, we can still derive $R; \Gamma \vdash E[M'] : \alpha$.*

Proof. By induction on the structure of E. $\square$

Lemma 8 (functional redexes). *If $R; \Gamma \vdash E[\Delta] : \alpha$ where Δ has the shape $(\lambda x.M)V$ or $\mathsf{let}\ !x = V\ \mathsf{in}\ M$ then $R; \Gamma \vdash E[[V/x]M] : \alpha$.*

Proof. If $\Delta = (\lambda x.M)V$ we appeal to the affine substitution lemma 4 and if $\Delta = \mathsf{let}\ !x = V\ \mathsf{in}\ M$ we rely on the intuitionistic lemma 5. This settles the case where the evaluation context E is trivial. If it is complex then we also need lemma 7. $\square$

Lemma 9 (side-effects redexes). *If $R; \Gamma \vdash \Delta : \alpha$ where Δ is one of the programs on the left-hand side then $R; \Gamma \vdash \Delta' : \alpha$ where Δ' is the corresponding program on the right-hand side:*

(1)	$E[\mathsf{set}(x, V)]$	$E[*] \mid (x \leftarrow V)$
(2)	$E[\mathsf{pset}(x, V)]$	$E[*] \mid (x \Leftarrow V)$
(3)	$E[\mathsf{get}(x)] \mid (x \leftarrow V)$	$E[V]$
(4)	$E[\mathsf{get}(x)] \mid (x \Leftarrow !V)$	$E[!V] \mid (x \Leftarrow !V)$

Proof. We proceed by case analysis.

1. Suppose we derive $R; \Gamma \vdash E[\mathsf{set}(x, V)] : \alpha$ from $R_2; \Gamma_2 \vdash \mathsf{set}(x, V) : \mathbf{1}$. By the typing rule for $\mathsf{set}(x, V)$ we know that $R_2 = r : ([v, v'], A) \uplus R_3$, $\mathcal{V}(r)$, $\Gamma_2 = x : (u, \mathsf{Reg}_r A) \uplus \Gamma_3$, and $R_3; \Gamma_3 \vdash V : A$. It follows that $R_2; \Gamma_2 \vdash (x \leftarrow V) : \mathbf{B}$. We can decompose $R_2; \Gamma_2$ into an additive part $(R_2; \Gamma_2)^0$ and a multiplicative one $(R_2; \Gamma_2)^1$. Then from $(R_2; \Gamma_2)^0 \vdash * : \mathbf{1}$, we can derive $R_1; \Gamma_1 \vdash E[*] : \alpha$, where $(R_1; \Gamma_1) \uplus (R_2; \Gamma_2)^1 = R; \Gamma$.

2. Suppose we derive $R; \Gamma \vdash E[\mathsf{pset}(x, V)] : \alpha$ from $R_2; \Gamma_2 \vdash \mathsf{pset}(x, V) : \mathbf{1}$. By the typing rule for $\mathsf{pset}(x, V)$ we know that $R_2 = r : ([v, v'], !A) \uplus R_3$, $\mathcal{P}(r)$, $\Gamma_2 = x : (u, \mathsf{Reg}_r !A) \uplus \Gamma_3$, and $R_3; \Gamma_3 \vdash V : !A$. It follows that $R_2; \Gamma_2 \vdash (x \Leftarrow V) : \mathbf{B}$. Then we reason as in the previous case.

3. Suppose $R_1; \Gamma_1 \vdash E[\mathsf{get}(x)] : \alpha$ is derived from $R_2; \Gamma_2 \vdash \mathsf{get}(x) : A$, that $R_3; \Gamma_3 \vdash (x \leftarrow V) : \mathbf{B}$, and that $R; \Gamma = (R_1; \Gamma_1) \uplus (R_3; \Gamma_3)$. Then $(R_2; \Gamma_2) \uplus (R_3; \Gamma_3) \vdash V : A$, by weakening. Also, let r be the region associated with the address x. We know that $\mathcal{V}(r)$ and that R_2 must have a reading usage on r. It follows that $\mathit{aff}(R_2; \Gamma_2)$ and therefore the context E *cannot* contain a !. Thus from $(R_2; \Gamma_2) \uplus (R_3; \Gamma_3) \vdash V : A$ we can derive $R; \Gamma \vdash E[V] : \alpha$.

4. Suppose $R_1; \Gamma_1 \vdash E[\mathsf{get}(x)] : \alpha$ is derived from $R_2; \Gamma_2 \vdash \mathsf{get}(x) : !A$, that $R_3; \Gamma_3 \vdash (x \Leftarrow !V) : \mathbf{B}$, and that $R; \Gamma = (R_1; \Gamma_1) \uplus (R_3; \Gamma_3)$. By the promotion rule, $R_3; \Gamma_3$ is a weakening of $R_4; \Gamma_4$ such that $\neg \mathit{aff}(R_4; \Gamma_4)$ and $R_4; \Gamma_4 \vdash V : A$. Then from $R_4; \Gamma_4 \vdash !V : !A$ we can derive $R'; \Gamma' \vdash E[!V] : \alpha$ where $R; \Gamma$ is a weakening of $(R'; \Gamma') \uplus (R_3; \Gamma_3)$.

$\square$

Theorem 3 (subject reduction). *If $R; \Gamma \vdash P : \alpha$ and $P \to P'$ then $R; \Gamma \vdash P' : \alpha$.*

Proof. We recall that $P \to P'$ means that P is structurally equivalent to a program $C[\Delta]$ where C is a static context, Δ is one of the programs on the left-hand side of the rewriting rules specified in Table 2, Δ' is the respective program on the right-hand side, and P' is syntactically equal to $C[\Delta']$.

By lemma 6, we know that $R; \Gamma \vdash C[\Delta] : \alpha$. This entails that $R'; \Gamma' \vdash \Delta : \alpha'$ for suitable R', Γ', α'. By lemmas 8 and 9, we derive that $R'; \Gamma' \vdash \Delta' : \alpha'$. Then by induction on the structure of C we argue that $R; \Gamma \vdash C[\Delta'] : \alpha$. $\qquad\square$

A.2 Proof of Theorem 2

Table 11 summarizes the main syntactic categories and the reduction rules of the intuitionistic system. It is important to notice that in the intuitionistic system regions are terms and that the operations that manipulate the store operate directly on the regions so that we write: $\mathsf{get}(r)$, $\mathsf{pset}(r, V)$, and $(r \Leftarrow V)$ rather than $\mathsf{get}(x)$, $\mathsf{pset}(x, V)$, and $(x \Leftarrow V)$.

Table 12 summarizes the typing rules for the stratified type and effect system.

Proviso To avoid confusion, in the following we write $\vdash_{AI}$ for provability in the affine-intuitionistic system and $\vdash_I$ for provability in the intuitionistic system.

The translation acts on typable programs. In order to define it, it is useful to go through a phase of *decoration* which amounts to label each occurrence (either free or bound) of a variable x of region type $\mathsf{Reg}_r A$ with the region r. For instance, suppose $R = r_1 : (U_1, A_1), \ldots, r_4 : (U_4, A_4)$ and suppose we have a provable judgement:

$$R; x_1 : (u_1, \mathsf{Reg}_{r_1} A) \vdash_{AI}$$
$$x_1 \mid \mathsf{let}\ !x_2 = \ldots\ \mathsf{in}\ x_2 \mid \lambda x_3.x_3 \mid \nu x_4\ x_4 : (\mathbf{B}, \emptyset)$$

Further suppose in the proof the variable x_i relates to the region r_i for $i = 1, \ldots, 4$. Then the decorated term is:

$$x_1^{r_1} \mid \mathsf{let}\ !x_2 = \ldots\ \mathsf{in}\ x_2^{r_2} \mid \lambda x_3.x_3^{r_3} \mid \nu x_4\ x_4^{r_4}\ .$$

The idea is that the translation of a decorated variable x^r is simply the region r so that in the previous case we obtain the following term of the intuitionistic system:

$$r_1 \mid (\lambda x_2.r_2)(\ldots) \mid \lambda x_3.r_3 \mid r_4\ .$$

Note that in the translation the ν's disappear while the λ's and let's are simulated by the intuitionistic λ's.

Assuming the decoration phase, the forgetful translation $(\_)$ is defined in Table 10.

Lemma 10. *The forgetful translation preserves provability in the following sense:*

1. If $R \vdash_{AI}$ then $\underline{R} \vdash_I$.

2. *If $R \vdash_{AI} \alpha$ then $\underline{R} \vdash_I \underline{\alpha}$.*

3. *If $R \vdash_{AI} (\alpha, e)$ then $\underline{R} \vdash_I (\underline{\alpha}, e)$.*

4. *If $R \vdash_{AI} \alpha \leq \alpha'$ then $\underline{R} \vdash_I \underline{\alpha} \leq \underline{\alpha}'$.*

5. *If $R \vdash_{AI} (\alpha, e) \leq (\alpha', e')$ then $\underline{R} \vdash_I (\underline{\alpha}, e) \leq (\underline{\alpha}', e')$.*

6. *If $R \vdash_{AI} \Gamma$ then $\underline{R} \vdash_{AI} \underline{\Gamma}$.*

7. *If $R; \Gamma \vdash_{AI} P : (\alpha, e)$ (and P has been decorated) then $\underline{R}; \underline{\Gamma} \vdash_I \underline{P} : (\underline{\alpha}, e)$.*

Proof. By induction on the provability relation $\vdash_{AI}$.

Concerning the rules for types and region contexts formation and for subtyping, the forgetful translation provides a one-to-one mapping from the rules of the affine-intuitionistic system to the rules of the intuitionistic one (the only exception are the rules for $!A$ which become trivial in the intuitionistic framework). Also note that $dom(R) = dom(\underline{R})$. With these remarks in mind, the proof of (1-5) is straightforward.

The proof of (6) follows directly from (2). We just notice that the forgetful translation of a context Γ eliminates all the variable associated with region types. The point is that if these variables occur in the program they will decorated and therefore in the translation they will be replaced by regions, *i.e.*, constants.

In the proof of (7), it is useful to make a few preliminary remarks. First, *weakening* is a derived rule for the intuitionistic system, so that if we can prove $R; \Gamma \vdash_I P : (\alpha, e)$ and $R, R' \vdash \Gamma, \Gamma'$ then we can prove $R, R'; \Gamma, \Gamma' \vdash_I P : (\alpha, e)$ too. Second, if $R_1 \uplus R_2$ is defined then $\underline{R_1} = \underline{R_2} = \underline{R_1 \uplus R_2}$. The proof is then a rather direct induction on the provability relation $\vdash_{AI}$. When we discharge an assumption and when we introduce a formal parameter with λ or with let we must distinguish the situation where the variable under consideration has region type, say, $\mathsf{Reg}_r A$. In this case the variable does not occur in the translation of the related context $\underline{\Gamma}$ and it is replaced in the term by the region r. $\square$

Next we want to relate the reduction of a program and of its translation. As already mentioned, in the intuitionistic system all stores are persistent. Consequently, a reduction such as:

$$\mathsf{get}(x^r) \mid (x^r \leftarrow V) \rightarrow V$$

might be simulated by

$$\mathsf{get}(r) \mid (r \Leftarrow \underline{V}) \rightarrow \underline{V} \mid (r \Leftarrow \underline{V}) \ .$$

In other terms, the translated program may contain more values in the store than the source program. To account for this, we introduce a 'simulation' relation $\mathcal{S}$ indexed on a pair $R; \Gamma$ such that $R \vdash \Gamma$ and Γ is just composed of variables of region type:

$$\begin{aligned}
\mathcal{S}_{R;\Gamma} = \quad &\{(P, Q) \mid R; \Gamma \vdash_{AI} P : (\alpha, e), \\
&\underline{R}; \_ \vdash_I Q : (\underline{\alpha}, e), \\
&Q \equiv (\underline{P} \mid S)\}
\end{aligned}$$

Lemma 11 (simulation). *If $(P, Q) \in \mathcal{S}_{R;\Gamma}$ and $P \to P'$ then $Q \to Q'$ and $(P', Q') \in \mathcal{S}_{R;\Gamma}$.*

Proof. Suppose $(P, Q) \in \mathcal{S}_{R;\Gamma}$. Then $(P, \underline{P}) \in \mathcal{S}_{R;\Gamma}$. Also if $P \to P'$ then $R; \Gamma \vdash_{AI} P'$ by subject reduction of the affine-intuitionistic system (incidentally, subject reduction holds for the intuitionistic system too [1]).

By definition $P \to P'$ means that P is structurally equivalent to a process P_1 which can be decomposed in a static context C and a *redex* Δ of the shape described in Table 2.

We notice that the forgetful translation preserves structural equivalence, namely if $P \equiv P_1$ then $\underline{P} \equiv \underline{P_1}$. Indeed, the commutativity and associativity rules of the affine-intuitionistic system match those of the intuitionistic system while the rules for commuting the ν's are 'absorbed' by the translation. For instance, $\underline{\nu x \ P \mid P'} = \underline{P} \mid \underline{P'} = \underline{\nu x \ (P \mid P')}$ with x not free in P'.

We also remark that the forgetful translation can be extended to static and evaluation contexts simply by defining $\underline{[\ ]} = [\ ]$. Then we note that the translation of a static (evaluation) context is an intuitionistic static (evaluation) context. In particular, this holds because the translation of a value is still a value.

Following these remarks, we can derive that $Q \equiv \underline{C[\Delta]} \mid S$. Thus it is enough to focus on the redexes Δ and show that each reduction in the affine-intuitionistic system is mapped to a reduction in the intuitionistic one and that the resulting program is still related to the program P' via the relation $\mathcal{S}_{R;\Gamma}$.

To this end, we notice that the translation commutes with the substitution so that $\underline{[V/x]M} = [\underline{V}/x]\underline{M}$. This is a standard argument, modulo the fact that the variable of region type have to be given a special treatment. For instance, we have:

$$\underline{[y^r/x^r]x^r} = \underline{y^r} = r = [r/x^r]r = [\underline{y^r}/x^r]\underline{x^r} \ .$$

Then one proceeds by case analysis on the redex Δ. Let us look at two cases in some detail. If

$$\Delta = E[\text{let } !x = V \text{ in } M] \to E[[V/x]M]$$

then

$$
\begin{aligned}
\underline{\Delta} \quad &= \quad \underline{E[\text{let } !x = V \text{ in } M]} \\
&= \quad \underline{E[(\lambda x. M)V]} \\
&\to \quad \underline{E[[V/x]M]} \\
&= \quad \underline{E}[[\underline{V}/x]\underline{M}] \\
&= \quad \underline{E}[\underline{[V/x]M} \ .
\end{aligned}
$$

On the other hand if $\Delta = E[\text{get}(x^r)] \mid (x^r \leftarrow V)$ then

$$
\begin{aligned}
\underline{\Delta} \quad &= \quad \underline{E}[\text{get}(r)] \mid (r \Leftarrow \underline{V}) \\
&\to \quad \underline{E}[\underline{V}] \mid (r \Leftarrow \underline{V}) \\
&= \quad \underline{E[V]} \mid (r \Leftarrow \underline{V}) \ .
\end{aligned}
$$

Notice that in this case we have an additional store $(r \Leftarrow \underline{V})$ which is the reason why in the definition of the relation $\mathcal{S}$ we relate a program to its translation in parallel with some additional store. $\qquad\square$

Theorem 4 ([1])**.** *If $R; \_ \vdash_I P : (\alpha, e)$ then all reductions starting from P terminate.*

Corollary 1 (termination)**.** *If $R; \Gamma \vdash_{AI} P : (\alpha, e)$ then all reductions starting from P terminate.*

Proof. By contradiction. We have $(P, \underline{P}) \in \mathcal{S}_{R;\Gamma}$ and $R; \_ \vdash_I \underline{P} : (\underline{\alpha}, e)$. If there is an infinite reduction starting from P then the simulation lemma 11 entails that there is an infinite reduction starting form $\underline{P}$. And this contradicts the termination of the intuitionistic system (Theorem 4). $\quad\square$

$$\frac{R \vdash \Gamma \quad x : (u, A) \in \Gamma}{R; \Gamma \vdash x : (A, \emptyset)} \qquad \frac{R \vdash \Gamma}{R; \Gamma \vdash * : (\mathbf{1}, \emptyset)}$$

$$\frac{R; \Gamma, x : (1, A) \vdash M : (\alpha, e)}{R; \Gamma \vdash \lambda x.M : (A \xrightarrow{e} \alpha, \emptyset)} \qquad \frac{\begin{array}{c} R_1; \Gamma_1 \vdash M : (A \xrightarrow{e} \alpha, e') \\ R_2; \Gamma_2 \vdash N : (A, e'') \end{array}}{R_1 \uplus R_2; \Gamma_1 \uplus \Gamma_2 \vdash MN : (\alpha, e \cup e' \cup e'')}$$

$$\frac{\begin{array}{cc} R \uplus R' \vdash (\Gamma \uplus \Gamma') & saff(R'; \Gamma') \\ R; \Gamma \vdash M : (A, e) & \neg aff(R; \Gamma) \end{array}}{R \uplus R'; \Gamma \uplus \Gamma' \vdash !M : (!A, e)} \qquad \frac{\begin{array}{c} R_1; \Gamma_1 \vdash M : (!A, e) \\ R_2; \Gamma_2, x : (\infty, A) \vdash (N, e') : \alpha \end{array}}{R_1 \uplus R_2; \Gamma_1 \uplus \Gamma_2 \vdash \mathsf{let}\ !x = M\ \mathsf{in}\ N : (\alpha, e \cup e')}$$

$$\frac{R; \Gamma, x : (u, \mathsf{Reg}_r A) \vdash P : (\alpha, e)}{R; \Gamma \vdash \nu x\ P : (\alpha, e)} \qquad \frac{\begin{array}{c} R \vdash \Gamma \quad x : (u, \mathsf{Reg}_r A) \in \Gamma \\ r : ([v, v'], A) \in R \quad v' \neq 0 \end{array}}{R; \Gamma \vdash \mathsf{get}(x) : (A, \{r\})}$$

$$\frac{\begin{array}{cc} \Gamma = x : (u, \mathsf{Reg}_r A) \uplus \Gamma' & \mathcal{V}(r) \\ R = r : ([v, v'], A) \uplus R' & v \neq 0 \\ R \vdash \Gamma & R'; \Gamma' \vdash V : (A, \emptyset) \end{array}}{R; \Gamma \vdash \mathsf{set}(x, V) : (\mathbf{1}, \{r\})} \qquad \frac{\begin{array}{cc} \Gamma = x : (u, \mathsf{Reg}_r !A) \uplus \Gamma' & \mathcal{P}(r) \\ R = r : ([v, v'], !A) \uplus R' & v \neq 0 \\ R \vdash \Gamma & R'; \Gamma' \vdash V : (!A, \emptyset) \end{array}}{R; \Gamma \vdash \mathsf{pset}(x, V) : (\mathbf{1}, \{r\})}$$

$$\frac{\begin{array}{cc} \Gamma = x : (u, \mathsf{Reg}_r A) \uplus \Gamma' & \mathcal{V}(r) \\ R = r : ([v, v'], A) \uplus R' & v \neq 0 \\ R \vdash \Gamma & R'; \Gamma' \vdash V : (A, \emptyset) \end{array}}{R; \Gamma \vdash (x \leftarrow V) : (\mathbf{B}, \emptyset)} \qquad \frac{\begin{array}{cc} \Gamma = x : (u, \mathsf{Reg}_r !A) \uplus \Gamma' & \mathcal{P}(r) \\ R = r : ([v, v'], !A) \uplus R' & v \neq 0 \\ R \vdash \Gamma & R'; \Gamma' \vdash V : (!A, \emptyset) \end{array}}{R; \Gamma \vdash (x \Leftarrow V) : (\mathbf{B}, \emptyset)}$$

$$\frac{\begin{array}{c} R_1; \Gamma_1 \vdash P : (\alpha, e) \\ R_2; \Gamma_2 \vdash S : (\mathbf{B}, \emptyset) \end{array}}{R_1 \uplus R_2; \Gamma_1 \uplus \Gamma_2 \vdash (P \mid S) : (\alpha, e)} \qquad \frac{\begin{array}{c} R_i; \Gamma_i \vdash P_i : (\alpha_i, e_i) \\ P_i \text{ not a store } i = 1, 2 \end{array}}{R_1 \uplus R_2; \Gamma_1 \uplus \Gamma_2 \vdash (P_1 \mid P_2) : (\mathbf{B}, e_1 \cup e_2)}$$

Table 7: An affine-intuitionistic type and effect system

$$\frac{}{R \vdash \alpha \leq \alpha} \qquad \frac{R \vdash A \leq A'}{R \vdash !A \leq !A'}$$

$$\frac{\begin{array}{c} e \subseteq e' \subseteq dom(R) \\ R \vdash A' \leq A \quad R \vdash \alpha \leq \alpha' \end{array}}{R \vdash (A \xrightarrow{e} \alpha) \leq (A' \xrightarrow{e'} \alpha')}$$

$$\frac{\begin{array}{c} e \subseteq e' \subseteq dom(R) \\ R \vdash \alpha \leq \alpha' \end{array}}{R \vdash (\alpha, e) \leq (\alpha', e')} \qquad \frac{\begin{array}{c} R; \Gamma \vdash M : (\alpha, e) \\ R \vdash (\alpha, e) \leq (\alpha', e') \end{array}}{R; \Gamma \vdash M : (\alpha', e')}$$

Table 8: Subtyping induced by effect containment

$$\frac{}{\emptyset \vdash} \qquad \frac{R \vdash A \quad r \notin dom(R)}{R, r:(U,A) \vdash}$$

$$\frac{R \vdash}{R \vdash \mathbf{1}} \qquad\qquad \frac{R \vdash}{R \vdash \mathbf{B}}$$

$$\frac{R \vdash A}{R \vdash !A} \qquad \frac{e \subseteq dom(R) \quad R \vdash A \quad R \vdash \alpha}{R \vdash (A \overset{e}{\multimap} \alpha)}$$

$$\frac{R \vdash \quad r:(U,A) \in R}{R \vdash \mathsf{Reg}_r A} \qquad \frac{R \vdash \alpha \quad e \subseteq dom(R)}{R \vdash (\alpha, e)}$$

Table 9: Formation of types and contexts (stratified)

$$\underline{\mathbf{1}} = \mathbf{1}, \quad \underline{\mathbf{B}} = \mathbf{B}, \quad \underline{A \overset{e}{\multimap} \alpha} = \underline{A} \overset{e}{\to} \underline{\alpha}, \quad \underline{!A} = \underline{A}, \quad \underline{\mathsf{Reg}_r A} = \mathsf{Reg}_r \underline{A}$$

$$\underline{r_1:(U_1,A_1),\ldots,r_n:(U_n,A_n)} = r_1:\underline{A_1},\ldots,r_n:\underline{A_n}$$

$$\underline{x:(u,A),\Gamma} = \begin{cases} x:\underline{A}, \underline{\Gamma} & \text{if } A \neq \mathsf{Reg}_r B \\ \underline{\Gamma} & \text{otherwise} \end{cases}$$

$$\underline{x} = x, \quad \underline{x^r} = r, \quad \underline{*} = *, \quad \underline{\lambda x.M} = \lambda x.\underline{M}, \quad \underline{MN} = \underline{M}\,\underline{N}$$

$$\underline{!M} = \underline{M}, \quad \underline{\text{let } !x = M \text{ in } N} = (\lambda x.\underline{N})\underline{M}, \quad \underline{\nu x\ M} = \underline{M},$$

$$\underline{\mathsf{get}(x^r)} = \mathsf{get}(r), \quad \underline{\mathsf{set}(x^r,V)} = \mathsf{set}(r,\underline{V}), \quad \underline{\mathsf{pset}(x^r,V)} = \mathsf{pset}(r,\underline{V}),$$

$$\underline{(x^r \leftarrow V)} = (r \Leftarrow \underline{V}), \quad \underline{(x^r \Leftarrow V)} = (r \Leftarrow \underline{V}), \quad \underline{P \mid P'} = \underline{P} \mid \underline{P'}$$

Table 10: Forgetful translation

$$x, y, \ldots \qquad\qquad\qquad\qquad\qquad \text{(Variables)}$$
$$r, s, \ldots \qquad\qquad\qquad\qquad\qquad \text{(Regions)}$$
$$V ::= x \mid * \mid r \mid \lambda x.M \qquad\qquad\qquad \text{(Values)}$$
$$M ::= V \mid MM \mid \mathsf{get}(V) \mid \mathsf{pset}(V, V) \mid (M \mid M) \qquad \text{(Terms)}$$
$$S ::= (r \Leftarrow v) \mid (S \mid S) \qquad\qquad\qquad \text{(Stores)}$$
$$P ::= M \mid S \mid (P \mid P) \qquad\qquad\qquad \text{(Programs)}$$
$$E ::= [\,] \mid EM \mid VE \qquad\qquad\qquad \text{(Evaluation Contexts)}$$
$$C ::= [\,] \mid (C \mid P)(P \mid C) \qquad\qquad\qquad \text{(Static Contexts)}$$

Operational semantics

$$P \mid P' \quad \equiv \quad P' \mid P \qquad \text{(Commutativity)}$$
$$(P \mid P') \mid P'' \quad \equiv \quad P \mid (P' \mid P'') \qquad \text{(Associativity)}$$

$$E[(\lambda x.M)V] \quad \rightarrow \quad E[[V/x]M]$$
$$E[\mathsf{get}(r)], (r \Leftarrow V) \quad \rightarrow \quad E[V], (r \Leftarrow V)$$
$$E[\mathsf{pset}(r, V)] \quad \rightarrow \quad E[*], (r \Leftarrow V)$$

Syntax: types and contexts

$$\alpha ::= A \mid \mathbf{B} \qquad\qquad \text{(Types)}$$
$$A ::= \mathbf{1} \mid (A \xrightarrow{e} \alpha) \mid \mathsf{Reg}_r A \qquad \text{(Value-types)}$$
$$\Gamma ::= x_1 : A_1, \ldots, x_n : A_n \qquad \text{(Contexts)}$$
$$R ::= r_1 : A_1, \ldots, r_n : A_n \qquad \text{(Region contexts)}$$

Table 11: Intuitionistic system: syntactic categories and operational semantics

$$\frac{}{\emptyset \vdash} \qquad \frac{R \vdash A \quad r \notin dom(R)}{R, r : A \vdash} \qquad \frac{R \vdash}{R \vdash \mathbf{1}} \qquad \frac{R \vdash}{R \vdash \mathbf{B}}$$

$$\frac{R \vdash A \quad R \vdash \alpha \quad e \subseteq dom(R)}{R \vdash (A \xrightarrow{e} \alpha)} \qquad \frac{R \vdash \quad r : A \in R}{R \vdash \mathsf{Reg}_r A} \qquad \frac{R \vdash \alpha \quad e \subseteq dom(R)}{R \vdash (\alpha, e)}$$

$$\frac{R \vdash \alpha}{R \vdash \alpha \leq \alpha} \qquad \frac{\begin{array}{c} R \vdash A' \leq A \quad R \vdash \alpha \leq \alpha' \\ e \subseteq e' \subseteq dom(R) \end{array}}{R \vdash (A \xrightarrow{e} \alpha) \leq (A' \xrightarrow{e'} \alpha')}$$

$$\frac{\begin{array}{c} R \vdash \alpha \leq \alpha' \\ e \subseteq e' \subseteq dom(R) \end{array}}{R \vdash (\alpha, e) \leq (\alpha', e')} \qquad \frac{R; \Gamma \vdash M : (\alpha, e) \quad R \vdash (\alpha, e) \leq (\alpha', e')}{R; \Gamma \vdash M : (\alpha', e')}$$

$$\frac{R \vdash \Gamma \quad x : A \in \Gamma}{R; \Gamma \vdash x : (A, \emptyset)} \qquad \frac{R \vdash \Gamma \quad r : A \in R}{R; \Gamma \vdash r : (\mathsf{Reg}_r A, \emptyset)} \qquad \frac{R \vdash \Gamma}{R; \Gamma \vdash * : (\mathbf{1}, \emptyset)}$$

$$\frac{R; \Gamma, x : A \vdash M : (\alpha, e)}{R; \Gamma \vdash \lambda x.M : (A \xrightarrow{e} \alpha, \emptyset)} \qquad \frac{R; \Gamma \vdash M : (A \xrightarrow{e_2} \alpha, e_1) \quad R; \Gamma \vdash N : (A, e_3)}{R; \Gamma \vdash MN : (\alpha, e_1 \cup e_2 \cup e_3)}$$

$$\frac{R; \Gamma \vdash V : (\mathsf{Reg}_r A, \emptyset)}{R; \Gamma \vdash \mathsf{get}(V) : (A, \{r\})} \qquad \frac{R; \Gamma \vdash V : (\mathsf{Reg}_r A, \emptyset) \quad R; \Gamma \vdash V' : (A, \emptyset)}{R; \Gamma \vdash \mathsf{pset}(V, V') : (\mathbf{1}, \{r\})}$$

$$\frac{r : A \in R \quad R; \Gamma \vdash V : (A, \emptyset)}{R; \Gamma \vdash (r \Leftarrow V) : (\mathbf{B}, \emptyset)} \qquad \frac{\begin{array}{c} R; \Gamma \vdash P : (\alpha, e) \\ R; \Gamma \vdash S : (\mathbf{B}, \emptyset) \end{array}}{R; \Gamma \vdash (P \mid S) : (\alpha, e)}$$

$$\frac{P_i \text{ not a store} \quad R; \Gamma \vdash P_i : (\alpha_i, e_i), \quad i = 1, 2}{R; \Gamma \vdash (P_1 \mid P_2) : (\mathbf{B}, e_1 \cup e_2)}$$

Table 12: Intuitionistic system: stratified types and effects