Fair Cooperative Multithreading (*)

or

Typing Termination in a Higher-Order Concurrent Imperative Language

Gérard Boudol

INRIA Sophia Antipolis

Abstract. We propose a new operational model for shared variable concurrency, in the context of a concurrent, higher-order imperative language à la ML. In our model the scheduling of threads is cooperative, and a non-terminating process suspends itself on each recursive call. A property to ensure in such a model is fairness, that is, any thread should yield the scheduler after some finite computation. To this end, we follow and adapt the classical method for proving termination in typed formalisms, namely the realizability technique. There is a specific difficulty with higher-order state, which is that one cannot define a realizability interpretation simply by induction on types, because applying a function may have side-effects at types not smaller than the type of the function. Moreover, such higher-order side-effects may give rise to computations that diverge without resorting to explicit recursion. We overcome these difficulties by introducing a type and effect system for our language that enforces a stratification of the memory. The stratification prevents the circularities in the memory that may cause divergence, and allows us to define a realizability interpretation of the types and effects, which we then use to prove the intended termination property. Our realizability interpretation also copes with dynamic thread creation.

1. Introduction

This work is concerned with the design of languages for concurrent programming with shared memory. In the recent past, some new applications have emerged, like web servers, network games or large scale databases, that are open to many simultaneous connections or requests, and are therefore inherently massively concurrent. It has been argued that kernel threads usually supported by operating systems are too limited and too inefficient to provide a convenient means for programming such applications, and that a user-level thread facility would provide better support [5, 7, 10, 40]. More generally, it appears that the preemptive discipline for scheduling threads is not very convenient for programming the above-mentioned applications, and that an event-driven model, or more generally a cooperative discipline is better suited for this purpose [2, 9, 10, 29].

In the *cooperative* programming model, a thread decides, by means of specific instructions (like yield for instance), when to leave its turn to another concurrent thread, and the scheduling is therefore distributed among the components. In this model, there is no

(*) Work partially supported by the CRE FT-R&D no 46136511, and by the ANR-SETI-06-010 grant.

data race, and modular programming can be supported, since using for instance a library function that operates on a shared, mutable data structure (as this is the case of methods attached to objects) does not require rewriting the library code. This is the model that we adopt as the basis for our concurrency semantics.

However, this model also has its drawbacks. A first one is that it does not directly support true concurrency, that is, it is not much better suited than the traditional sequential programming model to exploit multi-processor architectures. We do not address this issue here (for some work in this direction, see [16, 18]). The other flaw of cooperative scheduling is that if the active thread does not cooperate, failing to yield the scheduler, then the model is broken, in the sense that no other component will have a chance to execute. In other words, in cooperative programming, programs *must be cooperative*, or *fair*, that is, they should be guaranteed to either terminate or suspend themselves infinitely often. In particular, this property should be enforced in programming languages of the *reactive* family [14, 16, 19, 27, 32, 35]. Failing to cooperate may happen for instance if the active thread performs a blocking i/o, or runs into an error, or raises an uncaught exception, or diverges. Some efforts have been done to provide a better support to cooperative scheduling from the operating system [5, 7], and to develop asynchronous versions of system services. From the programming language point of view, cooperative programming should better be confined to be used in the framework of a *safe* language, like ML, where a program does not silently fall into an error. However, this is not enough: we have to avoid divergence in some way, while still being able to program non-terminating applications – any server for instance should conceptually have an infinite life duration, and should not be programmed to stop after a while.

In order to ensure fairness in cooperative programming, our proposal is to introduce a specific construct for programming non-terminating processes, the semantics of which is that a looping process suspends itself on each recursive call. We are assuming here that calling ordinary recursive functions – like for instance sorting a list – always terminate. Indeed, non-termination in the evaluation of an expression is usually the symptom of a programming error,[1] and it is therefore worth having a distinguished construct for programming intentionally looping processes. The idea of suspensive looping is not, however, enough to ensure fairness in a higher-order imperative programming model à la ML, that we use as a basis here (some other choices are obviously possible). We have to face a technical problem, which is that recursion may be encoded in two ways in a ML-like, or rather, for that matter, a SCHEME-like language. Indeed, it is well-known that one can define a fixpoint combinator in the untyped (call-by-value) λ-calculus. Moreover, as shown long ago by Landin [24], one can implement recursion by means of circular higher-order references (this is indeed the way it is implemented), like in

$$\begin{array}{rcl} \Upsilon & = & (\mathsf{let}\ f = (\mathsf{ref}\,\lambda xx)\ \mathsf{in}\ f := \lambda x((\,!\,f)x)\,;\,!\,f) \\ & \simeq & \mathsf{rec}\,f(x)(fx) \end{array} \tag{1}$$

where we use ML's notations $(\mathsf{ref}\,V)$ for creating a reference with initial value V, and $!\,u$ for reading the value of the reference u. The well-known method to recover from the first difficulty, disallowing the ability to derive fixpoint combinators, is to use a *type system*, but

[1] There is a lot of work on techniques for ensuring termination of recursive programs (we refrain from mentioning any paper from the huge literature on this topic), which is not the issue we are addressing here.

this is not enough to ensure termination of non-recursive programs in an imperative and functional language: in a simple type system, the expression Υ above has type $(\tau \rightarrow \tau)$ (it can indeed be written in OCaml for instance), but it diverges when applied to any value. As far as we can see, nothing has ever been proposed to ensure termination in a higher-order *imperative* (and concurrent) language, thus disallowing implicit recursion via the store. In this work we show that we can use a type and *effect* system [26] for this purpose. This is our main technical contribution.

Among the arguments used to show termination in typed higher-order formalisms, the realizability method is perhaps the best known, and certainly the most widely applicable. The realizability technique consists in defining, by induction on the structure of types, [2] an interpretation of types as sets of expressions, so that

1. the interpretation of a type only contains expressions enjoying the intended computational property (e.g. weak or strong normalizability);

2. typing is *sound*: a typed expression belongs to the interpretation of its type, or *realizes* its type.

The main ingredient in the definition of such an interpretation of types is that an expression M realizes a functional type $(\tau \rightarrow \sigma)$ if and only if its application (MN) to any argument N realizing τ is an expression that realizes σ. A realizability interpretation is therefore a special case of a "logical relation" [28, 31]. Such a realizability interpretation was first introduced by Kleene for intuitionistic arithmetic [22], though not for the purpose of proving termination. The technique was then used by Tait in [37], under the name of "convertibility" (with no reference to Kleene's notion of realizability), to show (weak) normalizability in the simply typed λ-calculus, and subsequently by Girard (see [21]) with his "candidats de réductibilité", and by Tait again [38] (who related it to Kleene's work) to show strong normalizability in higher-order typed λ-calculi. As a matter of fact, this technique seems to apply to most type theories – see the textbooks [8, 21, 23]. It has also been used for (functional fragments of) higher-order process calculi, and most notably the π-calculus [34, 42].

However, as far as I can see, the realizability technique has not been previously used for higher-order imperative (and concurrent) languages: the work that is technically the closest to ours, and which was our main source of inspiration, is the one by Pitts and Stark [30], who introduced logical relations to provide means to prove observational equivalence of programs (not to prove termination), but their language is restricted to offer only storable values of basic types. The program of Example (1) shows the main difficulty in attempting to define a realizability interpretation for higher-order imperative languages: to define the interpretation of a type τ, one should have previously defined the interpretation of the types of values stored in the memory that an expression of type τ may manipulate, but these types have no reason to be strictly smaller than τ. As another example, unrelated to divergence, one may imagine a function, say from lists of integers to integers, that reads from the memory (or import from a module) a second-order function like `map`, and uses it for its own computations.

To preclude the circularities in the memory that may cause recursion-free divergence, our type and effect system stratifies the memory into *regions*, in such a way that functional

[2] A more elaborate definition has to be used in the case of recursive types, see [12].

values stored in a given region may only have a latent effect, such as updating a reference, in strictly "lower" regions, thus rejecting (1) for instance. This stratification turns out to be also the key to defining a realizability interpretation, by a sophisticated induction over types and effects. We introduce such a realizability interpretation, for which our type and effect system is sound. From this we conclude that any typable program is fair, provided that using ordinary recursive functions does not introduce divergence.

The paper is organized as follows: we first define the syntax and operational semantics of our core language, introducing a "yield-and-loop" construct for programming non-terminating applications, and a new way of managing threads over an ML-like language. Our language also features block-structured local region declarations, for effect masking purposes. Then we define our type and effect system, where the main novelty is the region typing context, introducing a stratification into the memory: a region is given a type that can only involve effects in regions previously supplied with a type. The correctness of the typing of the region scoping construct is shown as part of a subject reduction property. Next we show our type safety result. To this end we introduce a realizability interpretation of the types and effects, and show that the type system is sound with respect to this interpretation. We then briefly conclude.

Some related works. We already mentioned that [30] was our main source of inspiration. This work has been slightly extended in [11], which deals with references that can hold values of a basic type, or of a reference type. In [3] a logical relation is defined for proving program equivalence in a language with recursive and quantified types, and the extension to mutable references is indicated as future work. This paper relies on an indexed model of types introduced in [6], which has been used in [4] to build a semantical model of reference types. This model uses a stratification on types, based on the number of computing steps for which an expression appears to belong to some type. Quoting [3], "*this stratification is essential for handling various circularities*," yet it differs from the one we introduce here for dealing with higher-order store. In [20], a syntactic notion of "imperative realizability" is defined for an ML-like language, with some restrictions to preclude aliasing, but without taking into account the fact that an expression of some type may have effects at bigger types, and therefore this notion does not seem to be well-founded. Some other related works are [13, 25], that follow a denotational approach to higher-order store; in the former, in particular, a parameterized semantical logical relation is introduced for proving program equivalence in a higher-order imperative language with recursive types. For simplicity, we are using here simple types, since we are mainly interested in the problem of defining a realizability interpretation for a language with higher-order references, which looks independent of the problem of extending the realizability technique to other enrichments to the simple types, like with recursive or higher-order types, for which one may try to adapt the syntactical techniques of [17] for instance.

2. Syntax

2.1 The language

Our core concurrent programming language is an ML-like language, that is a call-by-value λ-calculus extended with imperative constructs for creating, reading and updating references in the memory, and enriched with concurrent programming constructs. For reasons that are explained below, we also use a region-scoping construct, for "effect masking" purposes, as in [26]. The concurrent programming constructs include a thread-spawning construct

$(\mathsf{thread}\ M)$, and a "yield-and-loop" value $\nabla y M$, to be run by applying it to a void value $()$. This is our main linguistic novelty. This is similar to a recursive function $\mathsf{rec}\, y()M$, but we wish to stress the fact that the semantics are quite different. An expression $(\nabla y M())$ represents a recursive process which *yields the scheduler*, while unfolding a copy of M (where y is recursively bound to $\nabla y M$) to be performed when the scheduler resumes it. This construct is useful to build non-terminating processes, which should not hold the scheduler forever. In a more realistic language, we would also consider synchronization constructs, like the ones of reactive programming for instance [14, 16, 27, 32, 35].

We assume given an infinite set $\mathcal{R}eg$ of *region names* (or simply regions). These names will be used either as variables, which may be bound, or constants. We let ρ range over $\mathcal{R}eg$. The set $\mathcal{R}eg$ is the disjoint union of two infinite sets $\mathcal{R}eg_{sh}$, the set of possibly *shared* regions, and $\mathcal{R}eg_{un}$, the set of *unshared* regions. This distinction is introduced for two reasons: first, we would like to be able to distinguish "processes", that is programs that have their own memory but do not share references with other components, because these could be executed in preemptive mode, on different processors for instance (see [16, 18]). Second, we wish to distinguish potentially shared references, relative to which some compiler or hardware optimizations should be disallowed, like with "volatile" variables. This distinction of shared or unshared regions in the memory will be used in the type system. We also assume given an infinite set $\mathcal{L}oc$ of abstract *memory locations.*[3] We let u, $v\ldots$ range over $\mathcal{L}oc$. What we call a *reference* in this paper is a pair (u, ρ), that we will always denote by u_ρ, of a memory location and a region. The set $\mathcal{R}ef$ of references is therefore $\mathcal{L}oc \times \mathcal{R}eg$. We shall denote $\mathcal{L}oc \times \{\rho\}$ as $\mathcal{L}oc_\rho$. Finally we assume given an infinite set $\mathcal{V}ar$ of *variables*, ranged over by f, g, x, y, $z\ldots$. The syntax of our core language is as follows:

$$\begin{array}{lcll}
M,\ N\ldots & ::= & V \mid (MN) & \textit{expressions}\\
& \mid & (\mathsf{ref}_\rho M) \mid (!\,M) \mid (M := N) & \\
& \mid & (\mathsf{local}\ \rho\ \mathsf{in}\ M) \mid (M\backslash\varrho) \mid (\mathsf{thread}\, M) & \\
V,\ W\ldots & ::= & x \mid \lambda x M \mid \mathsf{rec}\, f(x)M \mid u_\rho \mid () \mid \nabla y M & \textit{values}
\end{array}$$

We require reference creation $(\mathsf{ref}_\rho M)$ to occur in an explicitly given region ρ, although in a type and effect inference approach (see [36]) this could perhaps be inferred. We denote by $\mathcal{V}al$ the set of values. As usual, the variable x is bound in $\lambda x M$, and similarly for f and x in $\mathsf{rec}\, f(x)M$, and for y in $\nabla y M$. An expression is said to be *closed* if it does not contain free term variables (but it may contain free regions). We shall use the standard notations for $(\lambda x M N)$, namely $(\mathsf{let}\ x = N\ \mathsf{in}\ M)$, and $(N\ ;\ M)$ when x is not free in M. The local region declaration $(\mathsf{local}\ \rho\ \mathsf{in}\ M)$ is a binder for ρ in M. We denote by $\{x \mapsto V\}M$ the capture-avoiding substitution of the value V for the free occurrences of the variable x in M, and similarly we write $\{\rho \mapsto \varrho\}M$ for region substitution. We shall consider expressions up to α-conversion, that is up to the renaming of bound variables and regions.

The $(\mathsf{local}\ \rho\ \mathsf{in}\ M)$ construct is denoted (private ϱ M) in [26]. A similar construct is considered in [39], namely $\mathsf{letregion}\ \rho\ \mathsf{in}\ M\ \mathsf{end}$, but the latter one is introduced for memory management purposes, rather than for effect masking purposes. The construction $(M\backslash\varrho)$ is denoted $(*\text{private}*\ \ \varrho\ M)$ in [26]. It is introduced in order to define the operational semantics of the region declaration construct, and is therefore a run-time construct, that

[3] This should rather be a run-time concept, but it will simplify the technical developments to include memory addresses in the source language.

should not be used in source programs. This is not a binder for ϱ. We denote by $\mathsf{reg}(M)$ the set of regions that occur free in M, so that in particular

$$\begin{array}{rcl} \mathsf{reg}(u_\rho) & = & \{\rho\} \\ \mathsf{reg}(\mathsf{ref}_\rho M) & = & \{\rho\} \cup \mathsf{reg}(M) \\ \mathsf{reg}(\mathsf{local}\ \rho\ \mathsf{in}\ M) & = & \mathsf{reg}(M) - \{\rho\} \\ \mathsf{reg}(M \backslash \varrho) & = & \mathsf{reg}(M) \cup \{\varrho\} \end{array}$$

We shall only consider for evaluation *legal* expressions, defined as follows:

Definition (Legal Expressions) 2.1. *An expression M is* legal *if and only if the sub-expressions* ($\mathsf{local}\ \rho\ \mathsf{in}\ N$) *of M are such that N does not contain any reference in region ρ, that is* $\mathsf{ref}(N) \cap \mathcal{L}oc_\rho = \emptyset$.

Since a program written in the source language is normally supposed not to contain any memory address, such a program is obviously a legal expression. We denote by $\mathsf{ref}(M)$ the set of references that occur in M, assuming that this is a legal expression.

2.2 Redexes, evaluation contexts, and configurations

The operational semantics mainly consists in reducing expressions into values, which, as usual, means repeatedly reducing a *redex* (reducible expression)[4] inside an *evaluation context* (see [41]). The redexes and evaluation contexts are given as follows:

$$\begin{array}{lcll} U & := & (VW) \mid (\mathsf{ref}_\rho V) \mid (\mathbf{!}\,V) \mid (V := W) & \textit{redexes} \\ & \mid & (\mathsf{local}\ \rho\ \mathsf{in}\ M) \mid (V \backslash \varrho) \mid (\mathsf{thread}\, M) & \\ \mathbf{E} & ::= & [] \mid \mathbf{E}[\mathbf{F}] & \textit{evaluation contexts} \\ \mathbf{F} & := & ([]\,N) \mid (V[]) \mid (\mathsf{ref}_\rho[]) \mid (\mathbf{!}\,[]) & \textit{frames} \\ & \mid & ([] := N) \mid (V := []) \mid ([] \backslash \varrho) & \end{array}$$

Notice that our notion of evaluation context is slightly different from the usual one. We write for instance $[][([]N)][(V[])]$ for what is usually written $((V[])N)$. Indeed, an evaluation context in our sense always has the form of a stack, namely $[][\mathbf{F}_0]\cdots[\mathbf{F}_n]$. As usual, we denote by $\mathbf{E}[M]$ the expression obtained by putting M into the context $\mathbf{E}$. This is defined as follows:

$$\begin{array}{rcl} [][M] & = & M \\ \mathbf{E}[\mathbf{F}][M] & = & \mathbf{E}[\mathbf{F}[M]] \end{array}$$

where $\mathbf{F}[M]$ is defined in the obvious way: $([]N)[M] = (MN)$, and so on. It is easy to check the following property, by induction on the syntax:

Lemma 2.2. *For any expression M either M is a value or there is a (unique) evaluation context $\mathbf{E}$ and a redex U such that $M = \mathbf{E}[U]$.*

We shall use in the operational semantics the set $\lceil \mathbf{E} \rceil$ of regions which are *masked* by the

[4] This terminology is slightly incorrect here since a "redex" in our sense can be an irreducible expression like $(()V)$ or $(() := V)$. We will not have to explicitly consider *faulty* expressions, as in [41], and a redex here rather is an expression that cannot be decomposed, but has to be reduced.

evaluation context $\mathbf{E}$. This is defined as follows:

$$\begin{array}{rcl} \lceil [] \rceil & = & \emptyset \\ \lceil \mathbf{E}[\mathbf{F}] \rceil & = & \begin{cases} \{\varrho\} \cup \lceil \mathbf{E} \rceil & \text{if } \mathbf{F} = ([] \backslash \varrho) \\ \lceil \mathbf{E} \rceil & \text{otherwise} \end{cases} \end{array}$$

The operational semantics will be defined as a transition relation between configurations, that involve in particular the current expression to evaluate, and a pool of threads waiting for execution. In order to get a *fair* scheduling strategy, we split this pool of threads into two parts, or more precisely two *turns*, that are multisets of expressions. Then a *configuration* is a tuple $C = (F, \delta, M, T, S)$, where

- F is a pair (R, L) of a finite set R of regions and a finite set L of locations that are used in the computation,
- δ is the memory,
- M is the currently evaluated expression (the active thread),
- T is the multiset of threads in the *current turn* of execution,
- S is the multiset of threads waiting for the *next turn*.

The first component F in a configuration is used in the generation of new names. In particular, we shall suppose given a function $\mathsf{fresh\_loc}$ from the set $\mathcal{P}_f(\mathcal{L}oc)$ of finite sets of locations to $\mathcal{L}oc$, such that for any L we have $\mathsf{fresh\_loc}(L) \notin L$.

The memory δ in a configuration is a mapping from a finite subset $\mathsf{dom}(\delta)$ of $\mathcal{R}ef$ to the set $\mathcal{V}al$ of values, such that to each memory address is assigned only one region, that is

$$u_{\rho_0} \in \mathsf{dom}(\delta) \ \& \ u_{\rho_1} \in \mathsf{dom}(\delta) \ \Rightarrow \ \rho_0 = \rho_1$$

Then we could actually regard a memory as a mapping from a finite set of locations to pairs made of a region and a value (where the region component is immutable). We define

$$\mathsf{im}(\delta) =_{\text{def}} \{\, V \mid \exists u_\rho \in \mathsf{dom}(\delta).\ V = \delta(u_\rho) \,\}$$

As usual, we denote by $\delta[u_\rho := V]$ the memory obtained from δ by updating the value of the reference u_ρ (which is assumed to belong to the domain of δ) by V.

As regards multisets, our notations are as follows. Given a set X, a *multiset* over X is a mapping E from X to the set $\mathbb{N}$ of non-negative integers, indicating the *multiplicity* $E(x)$ of an element. We denote by $\mathsf{0}$ the empty multiset, such that $\mathsf{0}(x) = 0$ for any x, and by x the singleton multiset such that $x(y) = (\mathsf{if}\ y = x\ \mathsf{then}\ 1\ \mathsf{else}\ 0)$. Multiset union $E + E'$ is given by $(E + E')(x) = E(x) + E'(x)$. In the following we only consider multisets of expressions, which are ranged over by $S, T \ldots$ We extend the notations for the sets of regions and references to multisets, that is $\mathsf{reg}(T)$ and $\mathsf{ref}(T)$, in the obvious way, i.e. $\mathsf{reg}(S + T) = \mathsf{reg}(S) \cup \mathsf{reg}(T)$ and $\mathsf{ref}(S + T) = \mathsf{ref}(S) \cup \mathsf{ref}(T)$, and also to configurations, with

$$\begin{array}{rcl} \mathsf{reg}(F, \delta, M, T, S) & = & \mathsf{reg}(\mathsf{dom}(\delta)) \cup \mathsf{reg}(\mathsf{im}(\delta)) \cup \mathsf{reg}(M) \cup \mathsf{reg}(T) \cup \mathsf{reg}(S) \\ \mathsf{ref}(F, \delta, M, T, S) & = & \mathsf{ref}(\mathsf{im}(\delta)) \cup \mathsf{ref}(M) \cup \mathsf{ref}(T) \cup \mathsf{ref}(S) \end{array}$$

In what follows we shall use the notion of a *well-formed* configuration. This is defined as follows:

$$
\begin{array}{rcl@{\quad}l@{\quad}l}
(F,\delta,\mathbf{E}[(\lambda x M V)]) & \underset{\emptyset}{\to} & (F,\delta,\mathbf{E}[\{x\mapsto V\}M],0,0) & & \\
(F,\delta,\mathbf{E}[(\mathsf{rec}\,f(x)MV)]) & \underset{\circlearrowright}{\to} & (F,\delta,\mathbf{E}[\{x\mapsto V\}\{f\mapsto \mathsf{rec}\,f(x)M\}M],0,0) & & \\
((R,L),\delta,\mathbf{E}[(\mathsf{ref}_\rho V)]) & \underset{e}{\to} & ((R,L\cup\{u\}),\delta\cup\{u_\rho\mapsto V\},\mathbf{E}[u_\rho],0,0) & u=\mathsf{fresh\_loc}(L) & (*) \\
(F,\delta,\mathbf{E}[(!\,u_\rho)]) & \underset{e}{\to} & (F,\delta,\mathbf{E}[V],0,0) & V=\delta(u_\rho) & (*) \\
(F,\delta,\mathbf{E}[(u_\rho := V)]) & \underset{e}{\to} & (F,\delta[u_\rho := V],\mathbf{E}[()],0,0) & & (*) \\
((R,L),\delta,\mathbf{E}[(\mathsf{local}\,\rho\,\mathsf{in}\,M)]) & \underset{\emptyset}{\to} & ((R\cup\{\varrho\},L),\delta,\mathbf{E}[(\{\rho\mapsto\varrho\}M\backslash\varrho)],0,0) & \varrho\in\mathcal{R}eg_{un}-R & \\
(F,\delta,\mathbf{E}[(V\backslash\varrho)]) & \underset{\emptyset}{\to} & (F,\delta,\mathbf{E}[V],0,0) & & \\
(F,\delta,\mathbf{E}[(\mathsf{thread}\,M)]) & \underset{\emptyset}{\to} & (F,\delta,\mathbf{E}[()],M,0) & & \\
(F,\delta,\mathbf{E}[(\nabla y M V)]) & \underset{\emptyset}{\to} & (F,\delta,(),0,\mathbf{E}[\{y\mapsto\nabla y M\}M]) & &
\end{array}
$$

$(*)$ where $e=\{\rho\}-\lceil\mathbf{E}\rceil$.

Figure 1: Operational Semantics (Sequential)

Definition (Well-Formed Configurations) 2.3.
A configuration $C=((R,L),\delta,M,T,S)$ *is* well-formed, *written* $C\,\mathsf{wf}$, *if and only if*
(a) *the* R *component includes all the regions that are mentioned in the configuration, that is* $\mathsf{reg}(C)\subseteq R$,
(b) *all the references that are mentioned in the configuration belong to* $L\times R$, *that is* $\mathsf{ref}(C)\subseteq L\times R$, *and*
(c) *all the references occurring in the configuration are bound to a value in the memory, that is* $\mathsf{ref}(C)\subseteq\mathsf{dom}(\delta)$.

To evaluate a closed expression M of the source language, that does not contain memory locations, i.e. $\mathsf{ref}(M)=\emptyset$, we start from the initial configuration $((\mathsf{reg}(M),\emptyset),\emptyset,M,0,0)$. Such a configuration is obviously well-formed. We are now ready to define the operational semantics of our language.

3. Operational semantics

3.1 The transition relation

In order to define the transition relation between configurations, we first define the sequential evaluation of expressions. This is given by an annotated transition relation

$$(F,\delta,M)\underset{e}{\to}(F',\delta',M',T,S)$$

where T and S are the multisets (which actually are either empty or a singleton) of threads spawned at this step, for execution in the current and next turn respectively, and e is the *effect* performed at this step. As usual, (imperative) effects record the regions in which an expression may operate, either by creating, reading or udpating a reference.[5] We shall also record as an effect the application of a recursive function, which we call *unfolding*, and denote $\circlearrowright$ (assuming that this symbol does not belong to $\mathcal{R}eg$). This is reminiscent of the fact that a recursive function is implemented by means of a circular reference (which does

[5] We conjecture that for the language without the $(\mathsf{local}\ \rho\ \mathsf{in}\ M)$ construct, we would not have to take the reading effect in consideration.

$$\frac{(F,\delta,M) \xrightarrow[e]{} (F',\delta',M',T',S')}{(F,\delta,M,T,S) \xrightarrow[e]{} (F',\delta',M',T+T',S+S')} \quad (\textsc{Exec})$$

$$\frac{}{(F,\delta,V,N+T,S) \xrightarrow[\emptyset]{} (F,\delta,N,T,S)} \quad (\textsc{Sched 1})$$

$$\frac{}{(F,\delta,V,\mathsf{0},Q+T) \xrightarrow[\emptyset]{} (F,\delta,Q,T,\mathsf{0})} \quad (\textsc{Sched 2})$$

Figure 2: Operational Semantics (Concurrent)

not explicitly appear in our operational semantics). In what follows it will not be necessary to distinguish different kinds of imperative effects, and therefore an effect is simply a (finite) subset of $\mathcal{R}eg \cup \{\circlearrowleft\}$. We denote by $\mathcal{E}\!f\!f$ the set of effects, that is $\mathcal{E}\!f\!f = \mathcal{P}_f(\mathcal{R}eg \cup \{\circlearrowleft\})$. We should point out here that the effect that we record in a transition is just an annotation, decorating the transition: there is no constraint associated with this effect, which would simply be ignored in an implementation. This annotation is, at each step, either empty or a singleton (which we abusively write as its single element).

The sequential part of the operational semantics is given in Figure 1. This part is quite standard, except as regards the looping construct ∇yM. An expression $\mathbf{E}[(\nabla yMV)]$ instantly terminates, returning $()$, while spawning as a thread the unfolding $\{y \mapsto \nabla yM\}M$ of the loop, in its evaluation context $\mathbf{E}$. One should notice that the thread $\mathbf{E}[\{y \mapsto \nabla yM\}M]$ created by means of the looping construct is delayed to be executed at the *next* turn, whereas with the construct $(\mathsf{thread}\, M)$ the new thread M is to be executed during the *current* turn (and moreover creating a new thread does not terminate the evaluation of the active expression). Then in order to execute immediately (and recursively) some task M, one should rather use the following construct:

$$\mu yM =_{\text{def}} \{y \mapsto \nabla yM\}M$$

For instance one can define $(\mathsf{loop}\, M) = \mu y(M\, ;\, (y()))$, which repeatedly starts executing M until termination, and then resumes at the next turn. To code a service $(\mathsf{repeat}\, M)$ that has to execute some task M at every turn, like continuously processing requests to a server for instance, one would write – using standard conventions for saving some parentheses:

$$(\mathsf{repeat}\, M) =_{\text{def}} \mu y.(\mathsf{thread}\, y())\, ;\, M$$

Anticipating on the concurrency semantics, we can describe the behaviour of this expression as follows: it spawns a new thread $N = (\nabla y((\mathsf{thread}\,(y()))\, ;\, M)())$ in the current turn of execution, and starts M. Whenever the thread N comes to be executed, during the current turn, a thread performing $(\mathsf{repeat}\, M)$ is spawned for execution at the next turn.

Our concurrency semantics, which, together with the "yield-and-loop" construct, is the main novelty of this work, is defined in Figure 2, which we now comment. We see from the (Exec) rule that the active expression keeps executing, possibly spawning new threads, till termination (we could also formulate this rule using a big-step sequential semantics). When this expression is terminated, a scheduling operation occurs: if there is some thread waiting for execution in the current turn, the value returned by the previously active thread is discarded, and a thread currently waiting is non-deterministically elected for becoming active, as stated by rule (SCHED 1). Otherwise, by the rule (SCHED 2), one chooses to execute a thread that was waiting for the next turn, if any, and simultaneously the other "next-turn" threads all become "current-turn" ones. If there is no waiting thread, the execution stops. One should notice that the termination of the active thread may be temporary. This is the case when the thread is actually performing a looping operation. Indeed, if we define

$$\mathsf{yield} = (\nabla y()())$$

then the execution of a thread $\mathbf{E}[\mathsf{yield}]$ stops, returning $()$, and will resume executing $\mathbf{E}[()]$ at the next turn. That is, we have

$$(F, \delta, \mathbf{E}[\mathsf{yield}], T, S) \xrightarrow[\emptyset]{} (F, \delta, (), T, S + \mathbf{E}[()])$$

It should be obvious that reduction preserves well-formedness, and that no illegal expression, in the sense of Definition 2.1, is created by reduction, since substitution is capture-avoiding.

3.2 Some definitions

Let us first define a kind of reflexive and transitive closure $\xrightarrow[e]{*}$ of the transition relation, as follows:

$$\frac{}{C \xrightarrow[\emptyset]{*} C} \qquad \frac{C \xrightarrow[e]{} C'' \xrightarrow[e']{*} C'}{C \xrightarrow[e \cup e']{*} C'}$$

In order to state our main result, we have to introduce some further notations and definitions. We shall denote by $\rightarrow^a$ the transition relation between configurations that only involves the active expression, that is, the transition relation defined as $\rightarrow$, but without using the rules (SCHED 1) and (SCHED 2). Similarly, we denote by $\rightarrow^c$ the transitions that occur in the current turn of execution. This is defined as $\rightarrow$, but without using (SCHED 2). We shall denote $\xrightarrow[e]{*}{}^a$ and $\xrightarrow[e]{*}{}^c$ the relations defined in the same way as $\xrightarrow[e]{*}$, based on $\rightarrow^a$ and $\rightarrow^c$ respectively. The sequences of $\rightarrow$ transitions can be decomposed into a sequence of $\rightarrow^c$ transitions, then possibly an application of the (SCHED 2) rule, then again a sequence of $\rightarrow^c$ transitions, and so on. Following the terminology of synchronous or reactive programming [16, 27, 35], a maximal sequence of $\rightarrow^c$ transitions may be called an *instant*. Then a property we wish to ensure is that all the instants in the execution of a program are finite. However, we shall guarantee this only in the case where there is no divergence resulting from calling ordinary recursive functions. We regard this kind of divergence as a programming error, but we do not address here the problem of ruling out such errors. Notice that it would be such an error to indefinitely spawn new threads, like with

$$(\mathsf{rec}\, f(x)(\mathsf{thread}\,(fx))())$$

Then we define:

DEFINITION (REACTIVITY) 3.1. *A well-formed closed configuration C is* reactive *if for any maximal sequence of $\rightarrow^c$-transitions*

$$C = (F_0, \delta_0, M_0, T_0, S_0) \xrightarrow[e_0]{}{}^c \cdots \xrightarrow[e_{n-1}]{}{}^c (F_n, \delta_n, M_n, T_n, S_n) \cdots$$

then either this sequence is infinite and $\forall i\, \exists j \geqslant i.\ e_j = \{\circlearrowleft\}$ or $\exists k.\ M_k \in \mathcal{V}al\ \&\ T_k = 0$.

In order to show our fairness property, we need to take into account the possible interleavings of threads in the current turn. We will have in particular to use the fact that if an expression of the form (MN) is fair in some sense, then both M and N terminate in the current turn, including the execution of the threads that these subexpressions may (hereditarily) create. Since we cannot assume that the threads spawned by M are executed before the ones spawned by N, we have to take into account in the notion of fairness (of M) the fact that the created threads may start in the context of a memory that is not necessarily the result of executing M or one of its "descendants." Then, given a set $\mathcal{M}$ of memories, we define a transition relation $\rightarrow^{c,\mathcal{M}}$ which is given as $\rightarrow^{c}$, except that in the case where a scheduling occurs, the new thread may be started in the context of any memory from $\mathcal{M}$:

$$\frac{(F,\delta,M) \underset{e}{\rightarrow} (F',\delta',M',T',S')}{(F,\delta,M,T,S) \underset{e}{\rightarrow}^{c,\mathcal{M}} (F',\delta',M',T+T',S+S')}$$

$$\frac{F \subseteq F' \quad \delta' \in \mathcal{M} \quad (F',\delta',N,T,S)\ \mathsf{wf}}{(F,\delta,V,N+T,S) \underset{\emptyset}{\rightarrow}^{c,\mathcal{M}} (F',\delta',N,T,S)}$$

We shall also use the relation $\overset{*}{\underset{e}{\rightarrow}}{}^{c,\mathcal{M}}$, defined in the same way as $\overset{*}{\underset{e}{\rightarrow}}{}^{c}$. The following definition is the crucial one, which will be used in the realizability interpretation. It states the kind of termination property we are seeking: a program M is said to be *fair* with respect to a set $\mathcal{M}$ of memories if, unless it calls some recursive functions that cause divergence (but our informal assumption here is that this is a programming error that does not occur), all the execution sequences in the current turn starting from P and a memory from $\mathcal{M}$ are finite, including the executions of the threads hereditarily created by M (and started in the context of an arbitrary memory in $\mathcal{M}$). Moreover these executions terminate in a cooperative way, that is in a state where the current turn is terminated. The formal statement is as follows:

Definition (Fairness) 3.2. *Given a set $\mathcal{M}$ of memories, a closed expression M is* fair *w.r.t. $\mathcal{M}$, in notation $M \searrow_{\mathcal{M}}$ if and only if, for all F and $\delta \in \mathcal{M}$ such that $C = (F,\delta,M,0,0)$ is well-formed, any maximal sequence of transitions*

$$C = (F_0,\delta_0,M_0,T_0,S_0) \xrightarrow[e_0]{}{}^{c,\mathcal{M}} \cdots \xrightarrow[e_{n-1}]{}{}^{c,\mathcal{M}} (F_n,\delta_n,M_n,T_n,S_n) \cdots$$

starting from C is either infinite, with $\forall i\, \exists j \geqslant i.\ e_j = \{\circlearrowleft\}$, or there exists k such that $M_k \in \mathcal{V}al$ & $T_k = 0$.

This definition is extended to multisets of expressions, as follows: $T \searrow_{\mathcal{M}}$ if and only if $T = M + T'$ implies that the statement of the previous definition holds for $C = (F,\delta,M,T',0)$.

To conclude this section we introduce a notion of *bisimulation*, to deal with the management of regions. One has noticed that, whereas the choice of a new location is made in a deterministic way, via the $\mathsf{fresh\_loc}$ function, the generation of new region names is left arbitrary. This is done for technical convenience only. We shall have to relate executions of a given expression starting with different initial sets R of given region names. It should be intuitively obvious that such executions are the same up to a (dynamically evolving) permutation of names. To make this precise, we adapt the notion of a "progressive bisimulation" from [1]. We let Ψ denote the set of partial, finite injections on $\mathcal{R}eg$. That is, $\psi \in \Psi$

if and only if ψ is a mapping from a finite subset $\mathsf{dom}(\psi)$ of $\mathcal{R}eg$ to a finite subset $\mathsf{im}(\psi)$ of $\mathcal{R}eg$, such that $\rho \neq \rho' \;\Rightarrow\; \psi(\rho) \neq \psi(\rho')$. We say that a pair (ρ, ρ') is *consistent* with ψ, in notation $(\rho, \rho') \uparrow \psi$ if and only if $(\rho, \rho') \in \psi$ or $\rho \notin \mathsf{dom}(\psi)$ and $\rho' \notin \mathsf{im}(\psi)$. A progressive bisimulation is a Ψ-indexed family $\mathcal{R} = \{\, \mathcal{R}_\psi \mid \psi \in \Psi \,\}$ of binary relations over the set $\mathcal{C}onfig$ of configurations, satisfying the conditions given below. Such a family of relations is said to be *symmetric* if and only if $\mathcal{R}_{\psi^{-1}} = (\mathcal{R}_\psi)^{-1}$ for all $\psi \in \Psi$.

DEFINITION (BISIMULATIONS) 3.3. *A* progressive bisimulation *is a symmetric family* $\mathcal{R} = \{\, \mathcal{R}_\psi \mid \psi \in \Psi \,\}$ *of binary relations over* $\mathcal{C}onfig$ *satifying: for all* $\psi \in \Psi$ *if* $C_0 \,\mathcal{R}_\psi\, C_1$ *and* $C_0 \xrightarrow[e]{} C_0'$ *then there exist* e', C_1' *and* ψ' *such that*

(i) $C_1 \xrightarrow[e']{} C_1'$ *and* $C_0' \,\mathcal{R}_{\psi'}\, C_1'$, *with* $\psi \subseteq \psi'$

(ii) $e = \{\circlearrowleft\} \;\Rightarrow\; e' = \{\circlearrowleft\}$

(iii) $e = \{\rho\} \;\Rightarrow\; e' = \{\rho'\} \;\&\; (\rho, \rho') \uparrow \psi'$

3.3 Some operational properties

Let us define $\delta \backslash R$ where R is a set of regions:

$$\delta \backslash R = \delta \upharpoonright (\mathsf{dom}(\delta) - \mathcal{L}oc \times R)$$

We also extend the notation $\lceil \cdot \rceil$ to expressions, as follows:

$$\begin{array}{rcl} \lceil V \rceil & = & \emptyset \\ \lceil \mathbf{E}[U] \rceil & = & \lceil \mathbf{E} \rceil \end{array}$$

The following should be obvious:

REMARK 3.4. *If* $((R, L), \delta, M) \xrightarrow[e]{} ((R', L'), \delta', M', T, S)$ *then* $\lceil M' \rceil \subseteq \lceil M \rceil \cup (R' - R)$ *and* $\delta' \backslash (e \cap \mathcal{R}eg) \cup \lceil M \rceil = \delta \backslash (e \cap \mathcal{R}eg) \cup \lceil M \rceil$

COROLLARY 3.5. *For any expression* M, *if* $((R, L), \delta, M, T, S) \xrightarrow[e]{*}{}^{a} ((R', L'), \delta', M', T', S')$ *and* $u_\rho \in \mathsf{dom}(\delta')$ *with* $\rho \notin e \cup (R' - R)$ *then* $u_\rho \in \mathsf{dom}(\delta)$ *and* $\delta'(u_\rho) = \delta(u_\rho)$.

PROOF: by induction on the length of the transition sequence, using the previous remark. ❑

The following should be clear:

REMARK 3.6. *If* $M \searrow_{\mathcal{M}}$ *and* $(F, \delta, M, 0, 0) \xrightarrow[e]{*}{}^{a} (F', \delta', V, T, S)$ *where* $(F, \delta, M, 0, 0)$ wf *and* $\delta \in \mathcal{M}$ *then* $T \searrow_{\mathcal{M}}$.

We can show that the definition of fairness for multisets of expressions is compatible with (finite) multiset union:

LEMMA 3.7. *If* T *and* S *are finite multisets of expressions, that is* $T = M_1 + \cdots + M_m$ *and* $S = N_1 + \cdots + N_n$ *then* $T \searrow_{\mathcal{M}} \;\&\; S \searrow_{\mathcal{M}} \Rightarrow\; (T + S) \searrow_{\mathcal{M}}$

PROOF (SKETCH): for this proof we introduce a refinement of the transition relation $\rightarrow^{c,\mathcal{M}}$. First, we consider configurations where with each expression is associated an index i with $1 \leqslant i \leqslant m{+}n$. The meaning is that an expression indexed by i originates in M_i if $1 \leqslant i \leqslant m$, and from N_j if $i = j + m$ with $1 \leqslant j \leqslant n$. We denote by M^i the expression M decorated with index i, and, for any multiset T of expressions, by T^i the multiset of the expressions of T decorated by i. Then the decorated transition relation is defined as follows, where Θ denotes a multiset of decorated expressions, and $1 \leqslant h \leqslant m + n$:

$$\frac{(F, \delta, M) \underset{e}{\rightarrow} (F', \delta', N, T, S')}{(F, \delta, M^h, \Theta, S) \overset{h}{\underset{e}{\longrightarrow}}{}^{c,\mathcal{M}} (F', \delta', N^h, \Theta + T^h, S + S')}$$

$$\frac{F \subseteq F' \quad \delta' \in \mathcal{M} \quad (F', \delta', M^h, \Theta, S)\ \mathsf{wf}}{(F, \delta, V^k, M^h + \Theta, S) \overset{h}{\underset{\emptyset}{\longrightarrow}}{}^{c,\mathcal{M}} (F', \delta', M^h, \Theta, S)}$$

It should be clear that if $M_1 + \cdots + M_m + N_1 + \cdots + N_n = P + T$ then any sequence of $\rightarrow^{c,\mathcal{M}}$ transitions from $(F, \delta, M, T, 0)$ wf can be lifted into a sequence of decorated $\rightarrow^{c,\mathcal{M}}$ transitions, where the steps decorated by i determine a sequence of $\rightarrow^{c,\mathcal{M}}$ transitions from $(F, \delta, M_i, 0, 0)$ (if $1 \leqslant i \leqslant m$) or from $(F, \delta, N_j, 0, 0)$ (if $i = j + m$ with $1 \leqslant j \leqslant n$). Then we use the hypotheses $T \searrow_{\mathcal{M}}$ and $S \searrow_{\mathcal{M}}$ to conclude. ❏

In the rest of this section, we show the operational property that motivated the definition of bisimulations, namely that the various executions from configurations which only differ in the initial set of given regions are the same, up to the renaming of regions. For any $\psi \in \Psi$, regarded as a (simultaneous) region substitution, let us define:

$$M \asymp_\psi M' \quad \Leftrightarrow_{\text{def}} \quad (\mathsf{reg}(M) - \mathsf{dom}(\psi)) \cap \mathsf{im}(\psi) = \emptyset \ \&\ M' = \psi M$$

It should be clear that $M \asymp_\psi M' \ \Rightarrow\ M' \asymp_{\psi^{-1}} M$. This relation is extended to multisets of expressions, in the obvious way, and to configurations, as follows: if $C = ((R, L), \delta, M, T, S)$ and $C' = ((R', L'), \delta', M', T', S')$ then $C \asymp_\psi C'$ if and only if $\mathsf{dom}(\psi) \subseteq R$, $(R - \mathsf{dom}(\psi)) \cap \mathsf{im}(\psi) = \emptyset$ and $R' = \psi(R)$, $L' = L$, $M \asymp_\psi M'$, $T \asymp_\psi T'$, $S \asymp_\psi S'$ and $u_\rho \in \mathsf{dom}(\delta) \Leftrightarrow \psi u_\rho \in \mathsf{dom}(\delta')$ and $\delta'(\psi u_\rho) = \psi\delta(u_\rho)$ for all $u_\rho \in \mathsf{dom}(\delta)$.

Lemma 3.8.

(i) *The family $\{ \asymp_\psi \mid \psi \in \Psi \}$ is a progressive bisimulation.*

(ii) If C and C' are two well-formed configurations such that $C \asymp_\psi C'$ then C is reactive if and only if C' is reactive.

Proof:

(i) It is clear that the family $\{ \asymp_\psi \mid \psi \in \Psi \}$ is symmetric. If $C_0 \asymp_\psi C_1$ and $C_0 \underset{e}{\rightarrow} C'_0$, we show (i)-(iii) of Definition 3.3 by case on the transition. Let us just examine two cases.

• If $C_0 = ((R, L), \delta, \mathbf{E}[(\mathsf{ref}_\rho V)], T, S)$ and $C'_0 = ((R, L \cup \{u\}), \delta \cup \{u_\rho \mapsto V\}, \mathbf{E}[u_\rho], T, S)$ with $u = \mathsf{fresh\_loc}(L)$ and $e = \{\rho\} - \lceil \mathbf{E} \rceil$, then we have $C_1 = ((R', L), \delta', (\psi\mathbf{E})[(\mathsf{ref}_{\rho'} \psi V)], T', S')$ where $\rho' = \psi(\rho)$ if $\rho \in \mathsf{dom}(\psi)$, and $\rho' = \rho$ otherwise. In both cases, we have $(\rho, \rho') \uparrow \psi$, and $C_1 \underset{e'}{\rightarrow} C'_1$ where $e' = \psi(e)$ and $C'_1 = ((R', L \cup \{u\}), \delta \cup \{u_{\rho'} \mapsto \psi V\}, \psi(\mathbf{E}[u_\rho]), T', S')$, and clearly $C'_0 \asymp_\psi C'_1$.

• If $C_0 = ((R, L), \delta, \mathbf{E}[(\mathsf{local}\ \rho\ \mathsf{in}\ M)], T, S)$ then $C'_0 = ((R \cup \{\varrho\}, L), \mathbf{E}[(\{\rho \mapsto \varrho\} M \backslash \varrho)], T, S)$

where $\varrho \in \mathcal{R}eg_{un} - R$, and $e = \emptyset$. Then $C_1 = ((R', L), \delta', \psi(\mathbf{E})[(\mathsf{local}\ \rho'\ \mathsf{in}\ \psi\{\rho \mapsto \rho'\}M)], T', S')$ where $\rho' \notin (\mathsf{dom}(\psi) \cup \mathsf{im}(\psi))$, and $C_1 \xrightarrow[\emptyset]{} C_1'$ where

$$C_1' = ((R \cup \{\varrho'\}, L), \delta', \psi(\mathbf{E})[(\{\rho' \mapsto \varrho'\}(\psi\{\rho \mapsto \rho'\}M) \backslash \varrho')], T', S')$$

with $\varrho' \in \mathcal{R}eg_{un} - R'$. Since $\mathsf{dom}(\psi) \subseteq R$ and $R' = \psi(R)$, we have $(\varrho, \varrho') \uparrow \psi$, and therefore, if we let $\psi' = \psi \cup \{(\varrho, \varrho')\}$, we have

$$\{\rho' \mapsto \varrho'\}(\psi\{\rho \mapsto \rho'\}M) = \psi'\{\rho \mapsto \varrho\}M$$

and it is then easy to conclude that $C_0' \asymp_{\psi'} C_1'$.

(ii) Let

$$C = C_0 \xrightarrow[e_0]{}{}^{c} \cdots \xrightarrow[e_{n-1}]{}{}^{c} C_n \cdots$$

be a maximal sequence of $\rightarrow^c$-transitions. If $C \asymp_\psi C'$, then by the previous point there exists a sequence

$$C' = C_0' \xrightarrow[e_0']{}{}^{c} \cdots \xrightarrow[e_{n-1}']{}{}^{c} C_n' \cdots$$

and $\{\, \psi_n \mid n \in \mathbb{N} \,\}$ with $\psi_0 = \psi$, such that $C_i \,\mathcal{R}_{\psi_i}\, C_i'$. If the former sequence is finite, and ends up with a configuration of the form $(F_k, \delta_k, V, \mathsf{0}, S_k)$ then the same holds for C', since obviously $V \asymp_{\psi_k} N \;\Rightarrow\; N \in \mathcal{V}al$ and $\mathsf{0} \asymp_{\psi_k} T \;\Rightarrow\; T = \mathsf{0}$. Otherwise, we have $e_i = \{\circlearrowleft\}$ for an infinite number of steps, and the same holds for C', thanks to the previous point and the clause (ii) of Definition 3.3. ❏

4. The type and effect system

4.1 Types

The types are

$$\tau,\ \sigma,\ \theta \ldots \in \mathcal{T}ype \ ::=\ \mathbb{1} \ \mid\ \theta\,\mathsf{ref}_\rho \ \mid\ (\tau \xrightarrow{e} \sigma)$$

The type $\mathbb{1}$ is also denoted unit (and sometimes improperly void). As usual, in the functional types $(\tau \xrightarrow{e} \sigma)$ we record the latent effect, that is the effect a value of this type may have when applied to an argument. We define the size $|\tau|$ and the set $\mathsf{reg}(\tau)$ of regions that occur in a latent effect in τ as follows:

$$\begin{array}{rcl@{\qquad}rcl}
|\mathbb{1}| & = & 0 & \mathsf{reg}(\mathbb{1}) & = & \emptyset \\
|\theta\,\mathsf{ref}_\rho| & = & 1 + |\theta| & \mathsf{reg}(\theta\,\mathsf{ref}_\rho) & = & \mathsf{reg}(\theta) \\
|\tau \xrightarrow{e} \sigma| & = & 1 + |\tau| + |\sigma| & \mathsf{reg}(\tau \xrightarrow{e} \sigma) & = & \mathsf{reg}(\tau) \cup (e \cap \mathcal{R}eg) \cup \mathsf{reg}(\sigma)
\end{array}$$

We shall say that a type τ is *pure* if it does not mention any imperative effect, that is $\mathsf{reg}(\tau) = \emptyset$.

In order to rule out from the memory the circularities that may cause divergence in computations, we assign a type to each region, in such a way that the region cannot be reached by using a value stored in that region. This is achieved, as in dependent type systems [8], by introducing the notion of a *well-formed type* with respect to a type assignment to regions. A *region typing context* Δ is a sequence $\rho_1 : \theta_1, \ldots, \rho_n : \theta_n$ of assignments of types to regions. We denote by $\mathsf{dom}(\Delta)$ the set of regions where Δ is defined, that is $\{\rho_1, \ldots, \rho_n\}$. Then we define by simultaneous induction two predicates $\Delta \vdash$, for "the context Δ is well-formed", and $\Delta \vdash \tau$, for "the type τ is well-formed in the context of Δ", as follows:

$$\frac{}{\emptyset \vdash} \qquad \frac{\Delta \vdash \theta}{\Delta, \rho : \theta \vdash}\ \rho \notin \mathsf{dom}(\Delta)$$

$$\frac{\Delta \vdash}{\Delta \vdash \mathbb{1}} \qquad \frac{\Delta \vdash \quad \Delta(\rho) = \theta}{\Delta \vdash \theta\,\mathsf{ref}_\rho} \qquad \frac{\Delta \vdash \tau \quad \Delta \vdash \sigma \quad e \cap \mathcal{R}eg \subseteq \mathsf{dom}(\Delta)}{\Delta \vdash (\tau \xrightarrow{e} \sigma)}$$

For any well-formed region typing context Δ, we denote by $\mathcal{ET}(\Delta)$ the set of pairs (e, τ) of an effect and a type such that $e \cap \mathcal{R}eg \subseteq \mathsf{dom}(\Delta)$ and $\Delta \vdash \tau$. One may observe that if $\rho_1 : \theta_1, \ldots, \rho_n : \theta_n \vdash$ then $i \neq j \Rightarrow \rho_i \neq \rho_j$. Moreover, it is easy to see that

$$\Delta \vdash \tau \ \Rightarrow\ \mathsf{reg}(\tau) \subseteq \mathsf{dom}(\Delta)$$

and therefore

$$\Delta \vdash \theta\,\mathsf{ref}_\rho \ \Rightarrow\ \rho \notin \mathsf{reg}(\theta) \tag{2}$$

because a proof of $\Delta \vdash \theta\,\mathsf{ref}_\rho$ must have the following form:

$$\cfrac{\cfrac{\vdots}{\cfrac{\cfrac{\vdots}{\Delta_0 \vdash \theta}}{\Delta_0, \rho : \theta \vdash}}\ \vdots}{\cfrac{\Delta_0, \rho : \theta, \Delta_1 \vdash}{\Delta_0, \rho : \theta, \Delta_1 \vdash \theta\,\mathsf{ref}_\rho}}$$

where $\rho \notin \mathsf{dom}(\Delta_0)$.

The important clause in the definition of well-formedness is the last one: to be well-formed in the context of Δ, the type $(\tau \xrightarrow{e} \sigma)$ of a function with side-effects must be such that all the regions involved in the latent effect e are already recorded in Δ (this is vacuously true if there are no such regions, and in particular if the functional type is pure). This is the way we will avoid "dangerous" circularities in the memory. For instance, if $\Delta \vdash (\tau \xrightarrow{e} \sigma)$ and $\rho \in e$, then the type $(\tau \xrightarrow{e} \sigma)\,\mathsf{ref}_\rho$ is not well-formed in the context of Δ, thanks to the remark (2) above.

4.2 Typing

The judgements of the type and effect system for our language have the form $\Delta; \Gamma \vdash M : e, \tau$, where Γ is a typing context in the usual sense, that is a mapping from a finite set $\mathsf{dom}(\Gamma)$ of variables to types. We omit this context when it has an empty domain, writing $\Delta; \vdash M : e, \tau$ in this case. We denote by $\Gamma, x : \tau$ the typing context which is defined as Γ, except for x, to which is assigned the type τ. We extend the well-formedness of types predicate to typing contexts, as follows:

$$\Delta \vdash \Gamma \ \Leftrightarrow_{\mathrm{def}}\ \Delta \vdash \quad \& \quad \forall x \in \mathsf{dom}(\Gamma).\ \Delta \vdash \Gamma(x)$$

$$\frac{\Delta \vdash \Gamma \quad \Gamma(x) = \tau}{\Delta;\Gamma \vdash x : \emptyset, \tau} \qquad \frac{\Delta;\Gamma, x : \tau \vdash M : e, \sigma \quad \Delta \vdash (\tau \xrightarrow{e} \sigma)}{\Delta;\Gamma \vdash \lambda x M : \emptyset, (\tau \xrightarrow{e} \sigma)}$$

$$\frac{\Delta;\Gamma, x : \tau, f : (\tau \xrightarrow{e'} \sigma) \vdash M : e, \sigma}{\Delta;\Gamma \vdash \mathsf{rec}\, f(x) M : \emptyset, (\tau \xrightarrow{e'} \sigma)}\ e' = \{\circlearrowright\} \cup e \qquad \frac{\Delta \vdash \Gamma \quad \Delta(\rho) = \theta}{\Delta;\Gamma \vdash u_\rho : \emptyset, \theta\,\mathsf{ref}_\rho}$$

$$\frac{\Delta \vdash \Gamma}{\Delta;\Gamma \vdash () : \emptyset, \mathbb{1}} \qquad \frac{\Delta;\Gamma, y : (\mathbb{1} \xrightarrow{e} \mathbb{1}) \vdash M : e, \mathbb{1} \quad \Delta \vdash}{\Delta;\Gamma \vdash \nabla y M : \emptyset, (\mathbb{1} \xrightarrow{e} \mathbb{1})}$$

$$\frac{\Delta;\Gamma \vdash M : e, (\tau \xrightarrow{e''} \sigma) \quad \Delta;\Gamma \vdash N : e', \tau}{\Delta;\Gamma \vdash (MN) : e \cup e' \cup e'', \sigma} \qquad \frac{\Delta;\Gamma \vdash M : e, \theta \quad \Delta(\rho) = \theta}{\Delta;\Gamma \vdash (\mathsf{ref}_\rho M) : \{\rho\} \cup e, \theta\,\mathsf{ref}_\rho}$$

$$\frac{\Delta;\Gamma \vdash M : e, \theta\,\mathsf{ref}_\rho}{\Delta;\Gamma \vdash (!\, M) : \{\rho\} \cup e, \theta} \qquad \frac{\Delta;\Gamma \vdash M : e, \theta\,\mathsf{ref}_\rho \quad \Delta;\Gamma \vdash N : e', \theta}{\Delta;\Gamma \vdash (M := N) : \{\rho\} \cup e \cup e', \mathbb{1}}$$

$$\frac{\Delta, \Delta' \vdash \Gamma, \tau \quad \Delta, \rho : \theta, \Delta';\Gamma \vdash M : e, \tau}{\Delta, \Delta';\Gamma \vdash (\mathsf{local}\ \rho\ \mathsf{in}\ M) : e - \{\rho\}, \tau}\ \rho \in \mathcal{R}eg_{un} \qquad \frac{\Delta \backslash \varrho \vdash \Gamma, \tau \quad \Delta;\Gamma \vdash M : e, \tau}{\Delta;\Gamma \vdash (M \backslash \varrho) : e - \{\varrho\}, \tau}$$

$$\frac{\Delta \upharpoonright \mathcal{R}eg_{sh};\Gamma \vdash M : e, \mathbb{1} \quad \Delta \vdash}{\Delta;\Gamma \vdash (\mathsf{thread}\, M) : e, \mathbb{1}} \qquad \frac{\Delta \vdash \tau \quad \Delta;\Gamma \vdash M : e, \sigma}{\Delta;\Gamma, x : \tau \vdash M : e, \sigma}\ x \notin \mathsf{dom}(\Gamma)$$

Figure 3: Type and Effect System

The rules of the type and effect system for expressions of the language are given in Figure 3. Most of the typing rules are standard, except for the fact that we check well-formedness with respect to the region typing context Δ. One can see that the expression Υ of Example (1) in the Introduction is not typable, since to type it, the variable f should have type $(\tau \xrightarrow{\{\rho\}} \sigma)\mathsf{ref}_\rho$, but this type is not well-formed (the fact that Υ is not typable will be a consequence of our type safety result). One may notice that some circularities in the memory are still permitted by the type system. For instance, if y is of type $(\tau \xrightarrow{\emptyset} \tau)\mathsf{ref}_\rho$ then the statement $y := \lambda x(\lambda z x y)$ is typable, introducing a value for y in the memory that contains the reference y itself. One can also see that some expressions that read above their type are typable, like $((!\, u_\rho)N)$ where $\Delta(\rho) = (\tau \xrightarrow{\emptyset} \sigma)$ and N is of type τ (a more interesting example was suggested in the Introduction, namely that of a function from lists of integers to integers that reads a map function from the memory). As a matter of fact, it is always safe to read functions of a pure type from the memory. Moreover, it is easy to see that our type system extends the usual simple type system for pure functions.

Notice that the effects of a thread are recorded as effects of the expression that spawns it. In the rule for the (thread M) construct, $\Delta \upharpoonright \mathcal{R}eg_{sh}$ stands for the restriction of Δ to the regions in $\mathcal{R}eg_{sh}$. Then in $(\mathsf{thread}\, M)$, the newly created thread M is assumed to (publicly) use only *shared* regions of the memory. We have explained this restriction in the previous section: we wish to know which references may be shared, and which are not. To compensate for this restriction, we have included into the language the local region declaration construct ($\mathsf{local}\ \rho\ \mathsf{in}\ M$), which allows a thread to use its own local memory. Notice that in the typing rule for ($\mathsf{local}\ \rho\ \mathsf{in}\ M$), the hypothesis $\Delta, \Delta' \vdash \Gamma, \tau$ ensures that ρ does not occur in Γ or τ. As usual, we have the following derived typing for sequential composition:

$$\frac{\Delta; \Gamma \vdash M : e, \tau \quad \delta; \Gamma \vdash N : e', \sigma}{\Delta; \Gamma \vdash M; N : e \cup e', \sigma}$$

Then for instance we have $\Delta; \Gamma \vdash M; () : e, \mathbb{1}$ if $\Delta; \Gamma \vdash M : e, \tau$ for some τ. In order to state our type safety result, we extend the typing to configurations and, first, to memories:

$$\Delta; \Gamma \vdash \delta \quad \Leftrightarrow_{\mathrm{def}} \quad \forall u_\rho.\ u_\rho \in \mathsf{dom}(\delta) \Rightarrow \begin{cases} \rho \in \mathsf{dom}(\Delta)\ \& \\ \Delta; \Gamma \vdash \delta(u_\rho) : \emptyset, \Delta(\rho) \end{cases}$$

The typing, or more accurately the effect system, is also extended to multisets of expressions, with judgements $\Delta; \Gamma \vdash T : e$, as follows:

$$\frac{\Delta \vdash \Gamma}{\Delta; \Gamma \vdash \mathsf{0} : \emptyset} \qquad \frac{\Delta; \Gamma \vdash M : e, \tau \quad \Delta; \Gamma \vdash T : e'}{\Delta; \Gamma \vdash M + T : e \cup e'}$$

Then we define

$$\Delta; \Gamma \vdash ((R, L), \delta, M, T, S) : e \ \Leftrightarrow_{\mathrm{def}} \ \exists e_M, e_T, e_S. \begin{cases} \mathsf{dom}(\Delta) \subseteq R\ \&\ \Delta; \Gamma \vdash \delta\ \& \\ e = e_M \cup e_T \cup e_S\ \& \\ \exists \tau.\ \Delta; \Gamma \vdash P : e_P, \tau\ \& \\ \Delta; \Gamma \vdash T : e_T\ \&\ \Delta; \Gamma \vdash S : e_S \end{cases}$$

4.3 Some properties

First we notice some obvious facts:

Remark 4.1.

(i) $\Delta; \Gamma \vdash M : e, \tau \ \Rightarrow\ \Delta \vdash\ \&\ (e, \tau) \in \mathcal{ET}(\Delta)$

(ii) $\Delta; \Gamma \vdash M : e, \tau \ \Rightarrow\ \mathsf{reg}(M) \subseteq \mathsf{dom}(\Delta)$

(iii) $V \in \mathcal{V}al\ \&\ \Delta; \Gamma \vdash V : e, \tau \ \Rightarrow\ e = \emptyset$

Now we show a subject reduction property, stating, roughly, that typability of configurations is preserved along the computations. Since the proof of this property follows the usual steps (see [41]), and is, for a large part, quite standard, we shall not give it in full details.

Proposition (Subject Reduction) 4.2. *Let C be a well-formed configuration. If $\Delta; \Gamma \vdash C : e$ and $C \xrightarrow[e']{*} C'$ then $e' \subseteq e$ and $\Delta'; \Gamma \vdash C' : e''$ for some $e'' \subseteq e$ and $\Delta' \supseteq \Delta$.*

Clearly it is enough to prove

Lemma 4.3. *Let $C = (F, \delta, M, T, S)$ be a well-formed configuration, such that $\Delta; \Gamma \vdash C : e$ and $(F, \delta, M) \xrightarrow[e']{} (F', \delta', M', T', S')$. Then $e' \subseteq e$ and there exist $\Delta' \supseteq \Delta$ and $e'' \subseteq e$ such that $\Delta'; \Gamma \vdash (F', \delta', M', T + T', S + S') : e''$.*

Proof (Sketch): by definition of the typing of configurations, we have $\Delta; \Gamma \vdash \delta(u_\rho) : \emptyset, \Delta(\rho)$ for all $u_\rho \in \mathsf{dom}(\delta)$, $\Delta; \Gamma \vdash M : e_M, \tau$ for some e_M and τ, and $\Delta; \Gamma \vdash T : e_T$ and

$\Delta;\Gamma \vdash S : e_S$ with $e = e_M \cup e_T \cup e_S$. We sketch the proof of the lemma by cases on the transition

$$(F,\delta,M) \xrightarrow[e']{} (F',\delta',M',T',S') \tag{3}$$

where $M = \mathbf{E}[U]$. To deal with the evaluation context $\mathbf{E}$, we use the following property – similar to the Replacement Lemma in [41]:

Lemma (Replacement). *If $\Delta;\Gamma \vdash \mathbf{E}[M] : e,\tau$ then there exist e_0 and σ such that $\Delta;\Gamma \vdash M : e_0,\sigma$ with $e_0 \subseteq e \cup \lceil\mathbf{E}\rceil$, and for any N if $\Delta';\Gamma \vdash N : e_1,\sigma$ with $\Delta \subseteq \Delta'$ and $e_1 \subseteq e_0$ then there exist $e' \subseteq e$ such that $\Delta';\Gamma \vdash \mathbf{E}[N] : e',\tau$.*

(The proof is by induction on $\mathbf{E}$, and by cases on $\mathbf{F}$ when $\mathbf{E} = \mathbf{E}'[\mathbf{F}]$.) Now we examine a few cases regarding the reduction (3).

- If the transition is $(F,\delta,\mathbf{E}[(\lambda xNV)]) \xrightarrow[\emptyset]{} (F,\delta,\mathbf{E}[\{x \mapsto V\}N],0,0)$, we have, using Remark 4.1(iii):

$$\dfrac{\dfrac{\vdots}{\Delta;\Gamma \vdash \mathbf{E}[(\lambda xNV)] : e_M,\tau}}{} \quad\text{from}\quad \dfrac{\dfrac{\dfrac{\vdots}{\Delta;\Gamma,x:\theta \vdash N : e_0,\sigma} \quad \dfrac{}{\Delta \vdash (\theta \xrightarrow{e_0} \sigma)}}{\Delta;\Gamma \vdash \lambda xN : \emptyset,(\theta \xrightarrow{e_0} \sigma)} \quad \dfrac{\vdots}{\Delta;\Gamma \vdash V : \emptyset,\theta}}{\dfrac{\Delta;\Gamma \vdash (\lambda xNV) : e_0,\sigma}{\dfrac{\vdots}{\Delta;\Gamma \vdash \mathbf{E}[(\lambda xNV)] : e_M,\tau}}}$$

Then we use

Lemma (Substitution). *If $\Delta;\Gamma,x:\tau \vdash M : e,\sigma$ and $\Delta;\Gamma \vdash V : \emptyset,\tau$ then $\Delta;\Gamma \vdash \{x \mapsto V\}M : e,\sigma$.*

(see [41]) to conclude $\Delta;\Gamma \vdash \{x \mapsto V\}N : e_0,\sigma$, and then use the Replacement Lemma. The proof is similar for $U = (\mathsf{rec}\, f(x)MV)$ and $U = (\nabla yMV)$.

- If the transition is $((R,L),\delta,\mathbf{E}[(\mathsf{ref}_\rho V)]) \xrightarrow[e']{} ((R,L \cup \{u\}),\delta \cup \{u_\rho \mapsto V\},\mathbf{E}[u_\rho],0,0)$ where $e' = \{\rho\} - \lceil\mathbf{E}\rceil$ and $u = \mathsf{fresh\_loc}(L)$, we have

$$\dfrac{\dfrac{\vdots}{\Delta;\Gamma \vdash V : \emptyset,\theta} \quad \dfrac{}{\Delta(\rho) = \theta}}{\dfrac{\Delta;\Gamma \vdash (\mathsf{ref}_\rho V) : \{\rho\},\theta\,\mathsf{ref}_\rho}{\dfrac{\vdots}{\Delta;\Gamma \vdash \mathbf{E}[(\mathsf{ref}_\rho V)] : e_M,\theta\,\mathsf{ref}_\rho}}}$$

Since $\Delta(\rho) = \theta \;\Rightarrow\; \Delta;\Gamma \vdash u_\rho : \emptyset,\theta\,\mathsf{ref}_\rho$, we conclude $\Delta;\Gamma \vdash \mathbf{E}[u_\rho] : e_0,\theta\,\mathsf{ref}_\rho$ with $e_0 \subseteq e_M$ using the Replacement Lemma, and therefore $\Delta;\Gamma \vdash (F,\delta \cup \{u_\rho \mapsto V\},\mathbf{E}[u_\rho],T,S) : e_0 \cup e_T \cup e_S$. The cases of $U = (!\,u_\rho)$ and $U = (u_\rho := V)$ are similar.

- If the transition is $((R,L),\delta,\mathbf{E}[(\mathsf{local}\ \rho\ \mathsf{in}\ N)]) \underset{\emptyset}{\rightarrow} ((R\cup\{\varrho\},L),\delta,\mathbf{E}[(\{\rho\mapsto\varrho\}N\backslash\varrho)],0,0)$ where $\varrho\in\mathcal{R}eg_{un}-R$, we have

$$\dfrac{\dfrac{\dfrac{}{\Delta_0,\Delta_1\vdash\Gamma,\sigma}\qquad\dfrac{\vdots}{\Delta_0,\rho:\theta,\Delta_1;\Gamma\vdash N:e_0,\sigma}}{\Delta_0,\Delta_1;\Gamma\vdash(\mathsf{local}\ \rho\ \mathsf{in}\ N):e_0-\{\rho\},\sigma}\ \rho\in\mathcal{R}eg_{un}}{\dfrac{\vdots}{\Delta_0,\Delta_1;\Gamma\vdash\mathbf{E}[(\mathsf{local}\ \rho\ \mathsf{in}\ N)]:e_M,\tau}}$$

with $\Delta=\Delta_0,\Delta_1$. Since $\Delta\vdash\Gamma,\sigma$ the region name ρ does not occur in this statement, and $\varrho\notin\mathsf{dom}(\Delta)$ since $\mathsf{dom}(\Delta)\subseteq R$, and we use

Lemma (Region Substitution). 4.4. *If $\Delta,\rho:\theta,\Delta';\Gamma\vdash M:e,\tau$ with $\rho\in\mathcal{R}eg_{un}$ and $\Delta,\Delta'\vdash\Gamma,\tau$ then, for any $\rho'\in\mathcal{R}eg_{un}-\mathsf{dom}(\Delta,\Delta')$, the judgement $\Delta,\rho':\theta,\Delta';\Gamma\vdash\{\rho\mapsto\rho'\}M:\{\rho\mapsto\rho'\}e,\tau$ is provable, with a proof having the same structure as the one of $\Delta,\rho:\theta,\Delta';\Gamma\vdash M:e,\tau$.*

(The proof is by induction on the typing judgement.) Let $\Delta'=\Delta_0,\varrho:\theta,\Delta_1$. Then we have $\Delta';\Gamma\vdash\{\rho\mapsto\varrho\}N:\{\rho\mapsto\varrho\}e_0,\sigma$, and therefore $\Delta';\Gamma\vdash(\{\rho\mapsto\varrho\}N\backslash\varrho):e_0-\{\rho\},\sigma$, hence $\Delta';\Gamma\vdash\mathbf{E}[(\{\rho\mapsto\varrho\}N\backslash\varrho)]:e_1,\tau$ with $e_1\subseteq e_M$ by the Replacement Lemma.

The other cases are left to the reader. ❏

5. The termination property

5.1 The realizability interpretation

In this section we define the *realizability predicate* which, given a well-formed region typing context Δ, states that an expression M realizes the effect e and the type τ in the context of Δ, in notation $\Delta\models M:e,\tau$. This is defined by induction on e and τ, with respect to a well-founded ordering that we now introduce. First, for each region typing Δ and type τ, we define the set $\mathsf{Reg}_\Delta(\tau)$, which intuitively is the set of regions in $\mathsf{dom}(\Delta)$ that are involved in a proof that τ is well-formed, in the case where $\Delta\vdash\tau$. This includes in particular the regions of $\mathsf{Reg}_\Delta(\theta)$ whenever τ is a functional type, and θ is the type assigned in Δ to a region that occurs in the latent effect of τ. Then, overloading the notation, we also define $\mathsf{Reg}_\Delta(e)\subseteq\mathcal{R}eg$ for $e\in\mathcal{E}\!f\!f$. The definition of $\mathsf{Reg}_\Delta(\tau)$ and $\mathsf{Reg}_\Delta(e)$ is by simultaneous induction on (the length of) Δ. For any given Δ, $\mathsf{Reg}_\Delta(\tau)$ is defined by induction on τ, in a uniform way:

$$\begin{aligned}\mathsf{Reg}_\Delta(\mathbb{1}) &= \emptyset\\ \mathsf{Reg}_\Delta(\theta\ \mathsf{ref}_\rho) &= \mathsf{Reg}_\Delta(\theta)\\ \mathsf{Reg}_\Delta(\tau\xrightarrow{e}\sigma) &= \mathsf{Reg}_\Delta(\tau)\cup\mathsf{Reg}_\Delta(\sigma)\cup\mathsf{Reg}_\Delta(e)\end{aligned}$$

Then $\mathsf{Reg}_\Delta(e)$ is given by:

$$\begin{aligned}\mathsf{Reg}_\emptyset(e) &= \emptyset\\ \mathsf{Reg}_{\Delta,\rho:\theta}(e) &= \begin{cases}\{\rho\}\cup\mathsf{Reg}_\Delta(e-\{\rho\})\cup\mathsf{Reg}_\Delta(\theta) & \text{if } \rho\in e\\ \mathsf{Reg}_\Delta(e) & \text{otherwise}\end{cases}\end{aligned}$$

It is easy to see that $\mathsf{reg}(\tau) \subseteq \mathsf{Reg}_\Delta(\tau) \subseteq \mathsf{dom}(\Delta)$ if $\Delta \vdash \tau$, and that $R \subseteq \mathsf{Reg}_\Delta(R)$ if $R \subseteq \mathsf{dom}(\Delta)$. Moreover, if τ is pure, then $\mathsf{Reg}_\Delta(\tau) = \emptyset$. The following is an equally easy but crucial remark:

Lemma 5.1. *If $\Delta \vdash$ and $\theta = \Delta(\rho)$, where $\rho \in \mathsf{dom}(\Delta)$, then $\mathsf{Reg}_\Delta(\theta) \subset \mathsf{Reg}_\Delta(\{\rho\})$.*

(The proof, by induction on Δ, is trivial, since $\Delta, \rho : \theta \vdash$ implies $\rho \notin \mathsf{dom}(\Delta)$ and $\Delta \vdash \theta$.) The realizability interpretation is defined by induction on a strict ordering on the pairs (e, τ), namely the lexicographic ordering on $(\mathsf{Reg}_\Delta(e) \cup \mathsf{Reg}_\Delta(\tau), |\tau|)$. More precisely, we define:

Definition (Effect and Type Strict Ordering) 5.2. *Let Δ be a well-formed region typing context. The relation $\prec_\Delta$ on $\mathcal{ET}(\Delta)$ is defined as follows: $(e, \tau) \prec_\Delta (e', \tau')$ if and only if*

(i) $\mathsf{Reg}_\Delta(e) \cup \mathsf{Reg}_\Delta(\tau) \subset \mathsf{Reg}_\Delta(e') \cup \mathsf{Reg}_\Delta(\tau')$, *or*

(ii) $\mathsf{Reg}_\Delta(e) \cup \mathsf{Reg}_\Delta(\tau) = \mathsf{Reg}_\Delta(e') \cup \mathsf{Reg}_\Delta(\tau')$ *and* $|\tau| < |\tau'|$.

We notice two facts about this ordering:

1. for pure types, this ordering is the usual one, that is $(\emptyset, \tau) \prec_\Delta (\emptyset, \sigma)$ if and only if $|\tau| < |\sigma|$;
2. the pure types are always smaller than impure ones, that is $(\emptyset, \tau) \prec_\Delta (\emptyset, \sigma)$ if $\mathsf{reg}(\tau) = \emptyset \neq \mathsf{reg}(\sigma)$.

The strict ordering $\prec_\Delta$ is *well-founded*, that is, there is no infinite sequence $(e_n, \tau_n)_{n \in \mathbb{N}}$ in $\mathcal{ET}(\Delta)$ such that $(e_{n+1}, \tau_{n+1}) \prec_\Delta (e_n, \tau_n)$ for all n. This allows us to use the principle of noetherian induction with respect to this strict ordering, namely that if a subset X of $\mathcal{ET}(\Delta)$ has the property

$$\big(\forall (e', \tau').\ (e', \tau') \prec_\Delta (e, \tau) \ \Rightarrow\ (e', \tau') \in X\big) \ \Rightarrow\ (e, \tau) \in X$$

then $X = \mathcal{ET}(\Delta)$. Notice that, by the lemma above, we have in particular $(\emptyset, \theta) \prec_\Delta (e, \tau)$ if $\theta \in \Delta(e)$.

Our realizability interpretation states that if an expression realizes a type, then in particular it is fair in the context of suitable memories. As explained in the Introduction, realizability has to be defined for the types of values that the expression may read or modify in the memory, and this is what we mean by "suitable." The portion of the memory that has to be "suitable" may be restricted to the regions where the expression may have a side-effect (as approximated by the type and effect system). In the following definition we write $\Delta \models V : \tau$ for $\Delta \models V : \emptyset, \tau$, and we let, for $e \in \mathcal{E}\!f\!f$:

$$\Delta \models \delta \restriction e \quad \Leftrightarrow_{\mathrm{def}} \quad \forall \rho \in e \cap \mathsf{dom}(\Delta).\ \forall u_\rho \in \mathsf{dom}(\delta).\ \Delta \models \delta(u_\rho) : \Delta(\rho)$$

Clearly, $\Delta \models \delta \restriction e$ is vacuously true if $e \subseteq \{\circlearrowleft\}$.

Definition (Realizability) 5.3. *The closed expression M* realizes *e, τ in the context of Δ, in notation $\Delta \models M : e, \tau$, if and only if the following holds, where $\mathcal{M} = \{\, \delta \mid \Delta \models \delta \restriction e \,\}$:*

(i) $(e, \tau) \in \mathcal{ET}(\Delta)$;

(ii) $M \searrow_{\mathcal{M}}$;

(iii) *if* $\mathsf{dom}(\Delta) \subseteq R$ *and* $\delta \in \mathcal{M}$ *are such that* $((R, L), \delta, M, 0, 0)$ wf *then*

$$((R, L), \delta, M, 0, 0) \xrightarrow[e']{*}{}^{c,\mathcal{M}} (F, \delta', P, T, S) \ \Rightarrow\ e' \subseteq e \ \&\ \delta' \in \mathcal{M}$$

(iv) *with the same hypothesis on* $((R, L), \delta, M, 0, 0)$ *as in* (iii)*, if*

$$((R, L), \delta, M, 0, 0) \xrightarrow[e']{*}{}^{a} (F, \delta', V, T, 0)$$

then

(a) *if* $\tau = \mathbb{1}$ *then* $V = ()$,

(b) *if* $\tau = \theta\,\mathsf{ref}_\rho$ *then* $V \in \mathcal{L}oc_\rho$,

(c) *if* $\tau = (\theta \xrightarrow{e''} \sigma)$ *then* $\forall W.\ \Delta \models W : \theta \ \Rightarrow\ \Delta \models (VW) : e'', \sigma.$

This is extended to open expressions as follows: if $\mathsf{fv}(M) \subseteq \mathsf{dom}(\Gamma)$ *where* $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ *then* $\Delta; \Gamma \models M : e, \tau$ *if and only if*

$$\forall i\,\forall V_i.\ \Delta \models V_i : \tau_i \Rightarrow \Delta \models \{x_1 \mapsto V_1, \ldots, x_n \mapsto V_n\}M : e, \tau$$

This definition is well-founded. Indeed, with the statement $\Delta \models \delta \upharpoonright e$ the definition of $\Delta \models M : e, \tau$ calls for $\Delta \models V : \emptyset, \theta$ where $\theta = \Delta(\rho)$ for some ρ in $\eta \cup e$ (if there is any such region), and we have seen that $(\emptyset, \theta) \prec_\Delta (e, \tau)$ in this case. If $\tau = (\theta \xrightarrow{e''} \sigma)$, the definition calls for $\Delta \models W : \emptyset, \theta$ and $\Delta \models M : e'', \sigma$. It is clear that, in this case, $(\emptyset, \theta) \prec_\Delta (e, \theta \xrightarrow{e''} \sigma)$ since $\mathsf{Reg}_\Delta(\theta) \subseteq \mathsf{Reg}_\Delta(\theta \xrightarrow{e''} \sigma)$ and $|\theta| < |\theta \xrightarrow{e''} \sigma|$. Moreover, since $\mathsf{Reg}_\Delta(e'') \subseteq \mathsf{Reg}_\Delta(\theta \xrightarrow{e''} \sigma)$, it is obvious that $(e'', \sigma) \prec_\Delta (e, \theta \xrightarrow{e''} \sigma)$.

Notice that the hypothesis of the item (iv) of the definition means in particular that reducing M does not end up performing a call to a recursive process $\nabla y N$. Then it is not difficult to see that an expression of the form $(\nabla y M V)$ realizes any type and effect, and therefore that $\Delta \models \nabla y M : e, (\tau \xrightarrow{e'} \sigma)$ for any Δ, η etc. We observe that, restricted to values, the realizability interpretation is quite standard (see [30]).

Remark 5.4. *Let* V *be a closed value. Then* $\Delta \models V : \tau$ *if and only if* $\Delta \vdash \tau$ *and*

(a) *if* $\tau = \mathbb{1}$ *then* $V = ()$,

(b) *if* $\tau = \theta\,\mathsf{ref}_\rho$ *then* $V \in \mathcal{L}oc_\rho$,

(c) *if* $\tau = (\theta \xrightarrow{e} \sigma)$ *then* $\forall W.\ \Delta \models W : \tau \ \Rightarrow\ \Delta \models (VW) : e, \sigma.$

One can also see that as regards the functional fragment of the language, the realizability interpretation is as one might expect.

5.2 Continuity

In the following we shall prove the soundness of the type system with respect to the realizability interpretation, that is, $\Delta; \Gamma \vdash M : e, \tau$ implies $\Delta; \Gamma \models M : e, \tau$. The proof of the soundness property proceeds by induction on the inference of the typing judgement $\Delta; \Gamma \vdash M : e, \tau$. In the cases where this inference uses a typing assumption about variables, namely in the cases of functional values, recursive or not, we see that we have to substitute for the variables some values realizing appropriate types. However, in the case of recursive

$$\begin{array}{rcl}
\mathcal{A}pp_n(x) & = & \{x\} \\
\mathcal{A}pp_n(\lambda x M) & = & \{\, \lambda x N \mid N \in \mathcal{A}pp_n(M)\,\} \\
\mathcal{A}pp_n(\mathsf{rec}\, f(x)M) & - & \{\, \mathsf{rec}^k f(x)N \mid k \leqslant n \ \&\ N \in \mathcal{A}pp_n(M)\,\} \\
\mathcal{A}pp_n(\mathsf{rec}^m f(x)M) & - & \{\, \mathsf{rec}^k f(x)N \mid k \leqslant \mathsf{min}(m,n) \ \&\ N \in \mathcal{A}pp_n(M)\,\} \\
\mathcal{A}pp_n(\nabla y M) & = & \{\, \nabla y N \mid N \in \mathcal{A}pp_n(M)\,\} \\
\mathcal{A}pp_n(MN) & = & \{\, (M'N') \mid M' \in \mathcal{A}pp_n(M) \ \&\ N' \in \mathcal{A}pp_n(N)\,\} \\
 & \vdots &
\end{array}$$

Figure 4: Approximants

functions, that is $\mathsf{rec}\, f(x)M$ (we have seen that the case of ∇yM is trivial), there is a circularity in this argument, since we would like to show that such an expression realizes its type by substituting $\mathsf{rec}\, f(x)M$ for f in M, with the hypothesis that the substituted value, that is $\mathsf{rec}\, F(x)M$, realizes the same type. Then in this case we shall resort to a *continuity* argument, as in [30]: we shall define, for each expression, a set of approximants, and we shall show that if all approximants of an expression realize a given type, then the expression realizes this type. Our approach however differs from the one of [30]: we introduce new *auxiliary values* in the language,

$$V \ ::= \ \cdots \ \mid \ \mathsf{rec}^n f(x)M$$

for any $n \in \mathbb{N}$, with the following operational semantics:

$$\begin{array}{rcl}
(F, \delta, \mathbf{E}[(\mathsf{rec}^0 f(x)MV)]) & \underset{\circlearrowleft}{\longrightarrow} & (F, \delta, \mathbf{E}[(\mathsf{rec}^0 f(x)MV)], 0, 0) \\
(F, \delta, \mathbf{E}[(\mathsf{rec}^{n+1} f(x)MV)]) & \underset{\circlearrowleft}{\longrightarrow} & (F, \delta, \mathbf{E}[\{x \mapsto V\}\{f \mapsto \mathsf{rec}^n f(x)M\}M], 0, 0)
\end{array}$$

and the typing

$$\frac{\Delta; \Gamma, x : \tau, f : (\tau \xrightarrow{e'} \sigma) \vdash M : e, \sigma}{\Delta; \Gamma \vdash \mathsf{rec}^n f(x)M : \emptyset, (\tau \xrightarrow{e'} \sigma)} \quad e' = \{\circlearrowleft\} \cup e$$

We then define the notion of an *approximant* of an expression, which is obtained by replacing each occurrence of rec in the expression with rec^n for some n. More precisely, we define

$$\mathcal{A}pp(M) =_{\text{def}} \bigcup \{\, \mathcal{A}pp_n(M) \mid n \in \mathbb{N}\,\}$$

where $\mathcal{A}pp_n(M)$ is defined by structural induction on M, in an obvious way, see Figure 4. We define on the extended language, with the auxiliary values $\mathsf{rec}^n f(x)M$, the *approximation ordering* $\sqsubseteq$, as the precongruence generated by

$$\begin{array}{rcl}
\mathsf{rec}^n f(x)M & \sqsubseteq & \mathsf{rec}\, f(x)M \\
\mathsf{rec}^n f(x)M & \sqsubseteq & \mathsf{rec}^{n+1} f(x)M
\end{array}$$

It is easy to see that this is actually an ordering, that is, an antisymmetric relation. One should notice that if $M \sqsubseteq N$, then the expression M has exactly the same structure as N, where some nodes rec or rec^n are replaced with rec^k for some k ($\leqslant n$). This will allow us to establish quite easily a direct correspondence between the transitions of an expression and those of its approximants, without having to reformulate the operational semantics, as this is done in [30]. The approximation ordering is extended pointwise to memories, that is

$$\delta' \sqsubseteq \delta \ \Leftrightarrow_{\mathrm{def}} \begin{cases} \mathsf{dom}(\delta') = \mathsf{dom}(\delta)\ \& \\ \forall u_\rho \in \mathsf{dom}(\delta).\ \delta'(u_\rho) \sqsubseteq \delta(u_\rho) \end{cases}$$

and to multisets of expressions, as the congruence with respect to multiset sum generated by $\sqsubseteq$. It should be clear that the following holds:

Remark 5.5.

(i) $V \sqsubseteq W\ \&\ M \sqsubseteq N \ \Rightarrow\ \{x \mapsto V\}M \sqsubseteq \{x \mapsto W\}N$

(ii) $M_0 \sqsubseteq N\ \&\ M_1 \sqsubseteq N \ \Rightarrow\ \exists M_0 \sqcup M_1 \sqsubseteq N.\ M_0 \sqsubseteq M_0 \sqcup M_1 \&\ M_1 \sqsubseteq M_0 \sqcup M_1$

One should also notice that

$$M \in \mathcal{A}pp(N) \ \Rightarrow\ M \sqsubseteq N$$
$$M \sqsubseteq N \ \Rightarrow\ M \in \mathcal{V}al \Leftrightarrow M \in \mathcal{V}al$$
$$M \sqsubseteq N \ \Rightarrow\ \Delta; \Gamma \vdash_\eta M : e, \tau \Leftrightarrow \Delta; \Gamma \vdash_\eta N : e.\tau$$

Lemma 5.6. $M \in \mathcal{A}pp(\{x \mapsto V\}N) \ \Rightarrow\ \exists W \in \mathcal{A}pp(V)\, \exists N' \in \mathcal{A}pp(N).\ M \sqsubseteq \{x \mapsto W\}N'$

Proof: by induction on N, using Remark 5.5(ii). ❏

Now we aim at showing that if M approximates N, and N converges, then M either diverges or converges towards a value that approximates the one of N, and that if M converges, then N converges too, towards a better value. A first step is:

Lemma 5.7. *Let N be a closed expression.*

(i) *If $(F, \delta, N) \underset{e}{\rightarrow} (F', \delta', N', T, S)$ and $M \sqsubseteq N$ and $\delta_0 \sqsubseteq \delta$ then either $e = \{\circlearrowleft\}$ and $(F, \delta_0, M) \underset{e}{\rightarrow} (F, \delta_0, M, 0, 0)$, or there exist $M' \sqsubseteq N'$, $\delta_1 \sqsubseteq \delta'$, $T' \sqsubseteq T$ and $S' \sqsubseteq S$ such that $(F, \delta_0, M) \underset{e}{\rightarrow} (F', \delta_1, M', T', S')$.*

(ii) *If $M \sqsubseteq N$ and $\delta_0 \sqsubseteq \delta$ and $(F, \delta_0, M) \underset{e}{\rightarrow} (F', \delta_1, M', T, S)$ then either $e = \{\circlearrowleft\}$ and $(F', \delta_1, M', T, S) = (F, \delta_0, M, 0, 0)$, or $(F, \delta, N) \underset{e}{\rightarrow} (F', \delta', N', T', S')$ for some δ', N', T' and S' such that $(\delta_1, M', T, S) \sqsubseteq (\delta', N', T', S')$.*

Proof: we prove (i) by case on the transition $(F, \delta, N) \underset{e}{\rightarrow} (F', \delta', N', T, S)$. Let us just examine the case $N = \mathbf{E}[(\mathsf{rec}\, f(x)MV)]$. In this case we have $e = \{\circlearrowleft\}$, $F' = F$, $\delta' = \delta$, $T = 0 = S$ and $N' = \mathbf{E}[\{x \mapsto V\}\{f \mapsto \mathsf{rec}\, f(x)M\}M]$, and either $M = \mathbf{E}'[(\mathsf{rec}\, f(x)M'V')]$ or $M = \mathbf{E}'[(\mathsf{rec}^n f(x)M'V')]$ with $\mathbf{E}' \sqsubseteq \mathbf{E}$, $M' \sqsubseteq M$ and $V' \sqsubseteq V$. In the second case, we have $(F, \delta_0, M) \underset{\circlearrowleft}{\rightarrow} (F, \delta_0, M, 0, 0)$ if $n = 0$, and otherwise

$$(F, \delta_0, M) \underset{\circlearrowleft}{\rightarrow} (F, \delta_0, \mathbf{E}'[\{x \mapsto V'\}\{f \mapsto \mathsf{rec}\, f(x)M'\}M', 0, 0)$$

or

$$(F, \delta_0, M) \underset{\circlearrowleft}{\rightarrow} (F, \delta_0, \mathbf{E}'[\{x \mapsto V'\}\{f \mapsto \mathsf{rec}\, f^{n-1}(x)M'\}M', 0, 0)$$

and we use Remark 5.5(i) to conclude. The proof of (ii) is similar. ❏

Let us denote by $(F,\delta,M)\Uparrow$ the fact that there exists a sequence of transitions from $(F,\delta,M,\mathsf{0},\mathsf{0})$ of the following form:

$$(F,\delta,M,\mathsf{0},\mathsf{0}) \xrightarrow[e]{*}{}^{c} (F',\delta',M',T,S) \xrightarrow[\circlearrowright]{}{}^{a} (F',\delta',M',T,S)$$

Then an obvious consequence of the previous lemma is:

Corollary 5.8. *Let N be a closed expression and $\mathcal{M}$ a set of memories.*
(i) If $(F,\delta,N,\mathsf{0},\mathsf{0}) \xrightarrow[e]{*}{}^{c,\mathcal{M}} (F',\delta',V,T,S)$ and $M \sqsubseteq N$ and $\delta_0 \sqsubseteq \delta$ then either $\circlearrowright\, \in e$ and $(F,\delta_0,M)\Uparrow$ or there exist W, δ_1, T' and S' such that $(F,\delta_0,M,\mathsf{0},\mathsf{0}) \xrightarrow[e]{*}{}^{c,\mathcal{M}} (F',\delta_1,W,T',S')$ with $W \sqsubseteq V$, $\delta_1 \sqsubseteq \delta'$ and $T' \sqsubseteq T$.
(ii) If $M \sqsubseteq N$ and $\delta_0 \sqsubseteq \delta$ and $(F,\delta_0,M,\mathsf{0},\mathsf{0})$ has a $\xrightarrow[e]{*}{}^{a}$ transition (resp. a $\xrightarrow[e]{*}{}^{c,\mathcal{M}}$ transition) to (F',δ_1,V,T,S) then there exist δ', W, T' and S' such that $(F,\delta,N,\mathsf{0},\mathsf{0})$ has a $\xrightarrow[e]{*}{}^{a}$ transition (resp. a $\xrightarrow[e]{*}{}^{c,\mathcal{M}}$ transition) to (F',δ',W,T',S') with $(\delta_1,V,T,S) \sqsubseteq (\delta',W,T',S')$.

as announced, and therefore

Corollary 5.9. *Let N be a closed expression. Then for all M and $\mathcal{M}$ if $M \sqsubseteq N$ then $M \searrow_{\mathcal{M}} \Leftrightarrow\ N \searrow_{\mathcal{M}}$.*

Now we show that if N approximates M, and M realizes e and τ, then N realizes the same effect and type.

Proposition 5.10. *Let M be a closed expression. If $\Delta \models M : e,\tau$ and $N \sqsubseteq M$ then $\Delta \models N : e,\tau$.*

Proof: by induction on (e,τ), ordered by $\prec_\Delta$. The points (i) and (ii) of Definition 5.3 are obvious, using Corollary 5.9. Let $\mathcal{M} = \{\,\delta \mid \Delta \models \delta \upharpoonright e\,\}$, $R \subseteq \mathcal{R}eg$ such that $\mathsf{dom}(\Delta) \subseteq R$, and $\delta \in \mathcal{M}$. If

$$((R,L),\delta,N,\mathsf{0},\mathsf{0}) \xrightarrow[e]{*}{}^{c,\mathcal{M}} (F,\delta',M,T,S)$$

then

$$((R,L),\delta,M,\mathsf{0},\mathsf{0}) \xrightarrow[e]{*}{}^{c,\mathcal{M}} (F,\delta'',N,T',S')$$

with $\delta' \sqsubseteq \delta''$ by Lemma 5.7(ii). Since $\Delta \models M : e,\tau$, we have $\delta'' \in \mathcal{M}$, hence also $\delta' \in \mathcal{M}$ by induction hypothesis. This shows (iii) of Definition 5.3. Let us now show (iv). If

$$((R,L),\delta,N,\mathsf{0},\mathsf{0}) \xrightarrow[e]{*}{}^{a} (F,\delta',V,T,\mathsf{0})$$

then by Corollary 5.8(ii) we have

$$((R,L),\delta,M,\mathsf{0},\mathsf{0}) \xrightarrow[e]{*}{}^{a} (F,\delta'',V',T',\mathsf{0})$$

with $V \sqsubseteq V'$. Since $\Delta \models M : e,\tau$, we have $\Delta \models V' : \tau$. We examine the possible cases. If $\tau = \mathbb{1}$ or $\tau = \theta\,\mathsf{ref}_\rho$ we have $V' = ()$ or $V' \in \mathcal{L}oc_\rho$, hence $V = V'$, and we are done in these cases. Otherwise, that is if $\tau = (\theta \xrightarrow{e''} \sigma)$, let W be such that $\Delta \models W : \theta$. Then $\Delta \models (V'W) : e'',\sigma$, and therefore, since $(VW) \sqsubseteq (V'W)$ we have $\Delta \models (VW) : e'',\sigma$ by induction hypothesis, because $(e'',\sigma) \prec_\Delta (e,\tau)$. ❏

The evaluation of an expression can always be approximated by the evaluation of an approximant of the expression, if we chose n large enough in approximating a sub-expression $\mathsf{rec}\, f(x)N$ as $\mathsf{rec}^n f(x)N'$. In the following lemma the definition of approximants is extended to evaluation contexts, and (pointwise) to memories.

Lemma 5.11. *Let N be a closed expression. If $(F, \delta, N) \underset{e}{\rightarrow} (F', \delta', N', T, S)$ and $(\delta'', M', T', S') \in \mathcal{A}pp(\delta'', N', T, S)$ then there exist $(\delta_0, M) \in \mathcal{A}pp(\delta, N)$ such that $(F, \delta_0, M) \underset{e}{\rightarrow} (F', \delta_1, M'', T'', S'')$ with $(\delta'', M', T', S') \sqsubseteq (\delta_1, M'', T'', S'')$.*

Proof (Sketch): by case on the transition $(F, \delta, N) \underset{e}{\rightarrow} (F', \delta', N', T, S)$. Let us just examine the case $N = \mathbf{E}[(\mathsf{rec}\, f(x)MV)]$. In this case we have $e = \{\circlearrowleft\}$, $R' = R$, $\delta' = \delta$, $T = \mathsf{0} = S$ and $N' = \mathbf{E}[\{x \mapsto V\}\{y \mapsto \mathsf{rec}\, f(x)M\}M]$. By Lemma 5.6 there exist $W \in \mathcal{A}pp(V)$, $\mathbf{E}' \in \mathcal{A}pp(\mathbf{E})$, $N_0, N_1 \in \mathcal{A}pp(M)$ and n such that $M' \sqsubseteq \mathbf{E}'[\{x \mapsto W\}\{f \mapsto \mathsf{rec}^n f(x)N_0\}N_1]$, and we also have $M' \sqsubseteq \mathbf{E}'[\{x \mapsto W\}\{f \mapsto \mathsf{rec}^n f(x)\overline{N}\}\overline{N}]$ where $\overline{N} = N_0 \sqcup N_1$. Then we may let $N = \mathbf{E}'[(\mathsf{rec}^{n+1} f(x)\overline{N}W)]$ and $\delta_0 = \delta''$ in this case. The other cases are left to the reader. ❏

An obvious consequence of this lemma is:

Corollary (The Approximation Lemma) 5.12. *Let N be a closed expression. Then for any set $\mathcal{M}$ of memories, if $(F, \delta, N, \mathsf{0}, \mathsf{0})$ has a $\xrightarrow[e]{*}{}^{c,\mathcal{M}}$ transition (resp. a $\xrightarrow[e]{*}{}^{a}$ transition) to (F', δ', N', T, S), and if $(\delta'', N'', T', S') \in \mathcal{A}pp(\delta', N', T, S)$ then there exists $(\delta_0, M) \in \mathcal{A}pp(\delta, N)$ such that $(F, \delta_0, M, \mathsf{0}, \mathsf{0})$ has a $\xrightarrow[e]{*}{}^{c,\mathcal{M}}$ transition (resp. a $\xrightarrow[e]{*}{}^{a}$ transition) to $(F', \delta_1, M', T'', S'')$ for some δ_1, M', T'' and S'' such that $(\delta'', N'', T', S') \sqsubseteq (\delta_1, M', T'', S'')$.*

Finally we can prove the continuity of the realizability interpretation:

Theorem (Continuity) 5.13.
$\big(\forall N \in \mathcal{A}pp(M).\ \Delta; \Gamma \models N : e, \tau\big) \ \Rightarrow\ \Delta; \Gamma \models M : e, \tau$

Proof: by induction on (e, τ). Let $\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$, and let $V_1, \ldots, V_n$ be closed values such that $\Delta \models V_i : \tau_i$ for all i. We let $\overline{M} = \{x_1 \mapsto V_1, \ldots, x_n \mapsto V_n\}M$. Let us show that $\Delta \models \overline{M} : e, \tau$. Since $\mathcal{A}pp(M) \neq \emptyset$ we have $(e, \tau) \in \mathcal{ET}(\Delta)$. Now let $\mathcal{M} = \{\, \delta \mid \Delta \models \delta{\restriction}e \,\}$. The fact that $\overline{M} \searrow_{\mathcal{M}}$ is a consequence of Corollary 5.9.

Now let R and δ be such that $\mathsf{dom}(\Delta) \subseteq R$, $((R, L), \delta, \overline{M}, \mathsf{0}, \mathsf{0})$ is weakly well-formed, and $((R, L), \delta, \overline{M}, \mathsf{0}, \mathsf{0}) \xrightarrow[e']{*}{}^{c,\mathcal{M}} (F, \delta', N, T, S)$, and let $(\delta'', N', T', S') \in \mathcal{A}pp(\delta', N, T, S)$. Then by Corollary 5.12 there exist $(\delta_0, N) \in \mathcal{A}pp(\delta, \overline{M})$ such that

$$((R, L), \delta_0, N, \mathsf{0}, \mathsf{0}) \xrightarrow[e']{*}{}^{c,\mathcal{M}} (F, \delta_1, M, T_1, S_1)$$

with $(\delta'', N', T', S') \sqsubseteq (\delta_1, M, T_1, S_1)$. Using Lemma 5.6 one can see that there exist $M_0 \in \mathcal{A}pp(M)$ and $W_i \in \mathcal{A}pp(V_i)$ for each i such that, if we let $\overline{M_0} = \{x_1 \mapsto W_1, \ldots, x_n \mapsto W_n\}M_0$, we have $N \sqsubseteq \overline{M_0} \in \mathcal{A}pp(\overline{M})$. By Corollary 5.8(ii) we have

$$(F, \delta_0, \overline{M_0}, \mathsf{0}, \mathsf{0}) \xrightarrow[e']{*}{}^{c,\mathcal{M}} (F', \delta_1', N'', T_1', S_1')$$

with $\delta_1 \sqsubseteq \delta_1'$. By Proposition 5.10 we have $\Delta \models W_i : \tau_i$ for all i and therefore $\Delta \models \overline{M_0} : e, \tau$, hence $\delta_1' \in \mathcal{M}$, which implies $\delta'' \in \mathcal{M}$ by Proposition 5.10 again. By induction hypothesis, we then have $\delta' \in \mathcal{M}$, and this shows (iii) of Definition 5.3.

Now assume that $((R,L),\delta,\overline{M},0,0) \xrightarrow[e']{*}{}^{a} (F,\delta',V,T,0)$. To show (iv) of Definition 5.3, we have to prove $\Delta \models V : \tau$. Let $V' \in \mathcal{App}(V)$. Then by Corollary 5.12 there exist $(\delta_0,N) \in \mathcal{App}(\delta,\overline{M})$ such that $((R,L),\delta_0,N,0,0) \xrightarrow[e']{*}{}^{a} (F,\delta_1,M,T_1,S_1)$ with $V' \sqsubseteq M$ and $0 \sqsubseteq S_1$. Then $M \in \mathcal{V}al$ and $S_1 = 0$, and therefore $\Delta \models M : \tau$, hence also $\Delta \models V' : \tau$ by Proposition 5.10. If $\tau = \mathbb{1}$ or $\tau = \theta\,\mathsf{ref}_\rho$ we have $V' = ()$ or $V' \in \mathcal{L}oc_\rho$, hence $V = V'$, and we are done in these cases. Otherwise, that is if $\tau = (\theta \xrightarrow{e''} \sigma)$, let W be such that $\Delta \models W : \theta$. Then for any $W' \in \mathcal{App}(W)$ we have $\Delta \models W' : \theta$ by Proposition 5.10 again, hence $\Delta \models (V'W') : e'',\sigma$, and therefore by induction hypothesis $\Delta \models (VW) : e'',\sigma$ for all W such that $\Delta \models W : \theta$. This shows $\Delta \models V : \tau$ in this case. ❏

5.3 Soundness and type safety

In this section we establish the *soundess* of the type system with respect to the realizability interpretation, namely that if an expression has effect e and type τ in some context, then in the same context it realizes e and τ (see [8, 23], and also [28], where soundness is called "the Basic Lemma"). We shall use a saturation property:

LEMMA (SATURATION) 5.14.

(i) $\Delta;\Gamma \models \{x \mapsto V\}M : e,\tau \;\Rightarrow\; \Delta;\Gamma \models (\lambda xMV) : e,\tau$

(ii) $\Delta;\Gamma \models \{x \mapsto V\}\{f \mapsto \mathsf{rec}\,f(x)M\}M : e,\tau \;\Rightarrow\; \Delta;\Gamma \models (\mathsf{rec}\,f(x)MV) : e,\tau$

(ii) $\Delta;\Gamma \models \{x \mapsto V\}\{f \mapsto \mathsf{rec}^m f(x)M\}M : e,\tau \;\Rightarrow\; \Delta;\Gamma \models (\mathsf{rec}^{m+1} f(x)MV) : e,\tau$

PROOF: if $\Gamma = x_1 : \tau_1,\ldots,x_n : \tau_n$ let $V_1,\ldots,V_n$ be such that $\Delta \models V_i : \tau_i$ for all i, and let

$$\begin{array}{rcl} N & = & \{x_1 \mapsto V_1,\ldots,x_n \mapsto V_n\}(\lambda xMV) \\ N' & = & \{x_1 \mapsto V_1,\ldots,x_n \mapsto V_n\}\{x \mapsto V\}M \end{array}$$

Since

$$(F,\delta,N,0,0) \xrightarrow[e']{+} (F',\delta',N'',T,S) \;\Leftrightarrow\; (F,\delta,N',0,0) \xrightarrow[e']{*} (F',\delta',N'',T,S)$$

it is clear that $\Delta \models N : e,\tau$ if and only if $\Delta \models N' : e,\tau$. The other cases are similar. ❏

PROPOSITION (SOUNDNESS) 5.15. $\Delta;\Gamma \vdash M : e,\tau \;\Rightarrow\; \Delta;\Gamma \models M : e,\tau$

PROOF: first we notice that the point (i) of Definition 5.3 is a consequence of Remark 4.1(i). Then we proceed by induction on the proof of $\Delta;\Gamma \vdash M : e,\tau$. This is trivial if M is a variable, a reference u_ρ or $()$, or if the last rule used in this proof is the weakening rule. We now examine the other cases.

$\mathbf{M = \lambda x N}$. If $M = \lambda xN$ with $e = \emptyset$, $\tau = (\theta \xrightarrow{e'} \sigma)$ and $\Delta;\Gamma,x : \theta \vdash N : e',\sigma$, let $W \in \mathcal{V}al$ be such that $\Delta \models W : \theta$. If $\Gamma = x_1 : \tau_1,\ldots,x_n : \tau_n$ and $\Delta \models V_i : \tau_i$ for all i we have, by induction hypothesis, $\Delta \models \{x_1 \mapsto V_1,\ldots,x_n \mapsto V_n\}\{x \mapsto W\}N : e',\sigma$, and therefore $\Delta \models (\{x_1 \mapsto V_1,\ldots,x_n \mapsto V_n\}MW) : e',\sigma$ by Lemma 5.14. This shows $\Delta;\Gamma \models M : e,\tau$ in this case, where the points (ii) and (iii) of Definition 5.3 are trivial since $\{x_1 \mapsto V_1,\ldots,x_n \mapsto V_n\}M \in \mathcal{V}al$.

$\mathbf{M = \mathsf{rec}^m f(x)N}$. If $M = \mathsf{rec}^m f(x)N$ with $e = \emptyset$, $\tau = (\theta \xrightarrow{e''} \sigma)$ and $\Delta;\Gamma,x : \theta, f : \tau \vdash N : e',\sigma$, where $e'' = e \cup \{\circlearrowleft\}$ and $\Gamma = x_1 : \tau_1,\ldots,x_n : \tau_n$, we proceed by induction on m.

Let $V_1, \ldots, V_n$ and V be closed values such that $\Delta \models V_i : \tau_i$ for all i and $\Delta \models V : \theta$. If we let $\overline{M} = \{x_1 \mapsto V_1, \ldots, x_n \mapsto V_n\}M = \mathsf{rec}^m f(x)\overline{N}$ where $\overline{N} = \{x_1 \mapsto V_1, \ldots, x_n \mapsto V_n\}N$, we have, in the case where $m = 0$,

$$(F, \delta, (\overline{M}V)) \underset{\circlearrowleft}{\rightarrow} (F, \delta, (\overline{M}V), \mathsf{0}, \mathsf{0})$$

for any δ, and therefore $\Delta \models (\overline{M}V) : e'', \sigma$, and this shows

$$\Delta; \Gamma \models \mathsf{rec}^0 f(x)N : \emptyset, (\theta \xrightarrow{e''} \sigma)$$

Now if $m = k + 1$ we have

$$(F, \delta, (\overline{M}V)) \underset{\circlearrowleft}{\rightarrow} (F, \delta, \{x \mapsto V\}\{f \mapsto \mathsf{rec}^k f(x)\overline{N}\}\overline{N}, \mathsf{0}, \mathsf{0})$$

Since $\Delta \models \mathsf{rec}^k f(x)\overline{N} : \tau$ by induction hypothesis (on m), and $\Delta; \Gamma, x : \theta, f : \tau \models N : e', \sigma$ by induction hypothesis (on the inference of the typing judgement $\Delta; \Gamma, x : \theta, f : \tau \vdash N : e', \sigma$), we have $\Delta; \Gamma \models \{x \mapsto V\}\{f \mapsto \mathsf{rec}^k f(x)N\}N : e', \sigma$, and we use Lemma 5.14 to conclude in this case.

$\mathbf{M} = \mathsf{rec}\,\mathbf{f(x)N}$ If $M = \mathsf{rec}\,f(x)N$ with $e = \emptyset$, $\tau = (\theta \xrightarrow{e''} \sigma)$ and $\Delta; \Gamma, x : \theta, f : \tau \vdash N : e', \sigma$ where $e'' = e' \cup \{\circlearrowleft\}$, we have $\Delta; \Gamma \models \mathsf{rec}^m f(x)N : \emptyset, \tau$ for all m, as we have just seen, and therefore $\Delta; \Gamma \models M' : \emptyset, \tau$ for all $M' \in \mathcal{App}(M)$ by Proposition 5.10, hence $\Delta; \Gamma \models M : \emptyset, \tau$ by Theorem 5.13.

$\mathbf{M = \nabla yN}$. If $M = \nabla yN$ then $e = \emptyset$, $\tau = (\mathbb{1} \xrightarrow{e'} \mathbb{1})$ and $\Delta; \Gamma, y : \tau \vdash N : e', \mathbb{1}$. If the typing context Γ is $x_1 : \tau_1, \ldots, x_n : \tau_n$, and $\Delta \models V_i : \tau_i$ for all i then $\{x_1 \mapsto V_1, \ldots, x_n \mapsto V_n\}M \in \mathcal{Val}$, and therefore the points (ii) and (iii) of Definition 5.3 are trivial in this case. Let $M' = \{x_1 \mapsto V_1, \ldots, x_n \mapsto V_n\}M$, and let $W \in \mathcal{Val}$ be such that $\Delta \models W : \mathbb{1}$, that is, $W = ()$. Then

$$(F, \delta, (M'W), \mathsf{0}, \mathsf{0}) \underset{\emptyset}{\rightarrow} (F, \delta, (), \mathsf{0}, N')$$

where $N' = \{x_1 \mapsto V_1, \ldots, x_n \mapsto V_n\}\{y \mapsto M\}N$, and therefore it is obvious that $\Delta \models (M'W) : e', \mathbb{1}$ (the expression $(M'W)$ actually realizes any type and effect). This shows $\Delta; \Gamma \models M : e, \tau$.

Since the context Γ plays no role in the other cases, for simplicity we shall assume for the rest of the proof that this context is empty, and in particular that M is closed (the proof for open terms is the same, except that it involves in each case appropriate substitutions of values).

$\mathbf{M = (M_0N)}$. If $M = (M'N)$ with $\Delta; \vdash M' : e_0, (\theta \xrightarrow{e_2} \tau)$ and $\Delta; \vdash N : e_1, \theta$ where $e = e_0 \cup e_1 \cup e_2$, we first show (ii) and (iii) of Definition 5.3. Let $\mathcal{M} = \{\, \delta \mid \Delta \models \delta \upharpoonright e \,\}$ and $\mathcal{M}_i = \{\, \delta \mid \Delta \models \delta \upharpoonright e_i \,\}$. Since $e_0 \subseteq e$ we have $\mathcal{M} \subseteq \mathcal{M}_0$, and therefore, by induction hypothesis, we have $M' \searrow_{\mathcal{M}}$, and similarly $N \searrow_{\mathcal{M}}$. Let $R \subseteq \mathcal{Reg}$, $L \subseteq \mathcal{Loc}$ and $\delta \in \mathcal{M}$ be such that $\mathsf{dom}(\Delta) \subseteq R$ and $C = ((R, L), \delta, M, \mathsf{0}, \mathsf{0})$ is weakly well-formed, and let us consider a maximal $\rightarrow^{c,\mathcal{M}}$-transition sequence from C:

$$C = (F_0, \delta_0, M_0, T_0, S_0) \xrightarrow[\varepsilon_0]{}^{c,\mathcal{M}} \cdots \xrightarrow[\varepsilon_{n-1}]{}^{c,\mathcal{M}} (F_n, \delta_n, M_n, T_n, S_n) \cdots \tag{4}$$

This sequence starts with a (possibly empty) sequence of computations of the function M', that is, there is a (maximal) sequence of $\rightarrow^{a}$-transitions

$$(F_0,\delta_0,M',T_0,S_0) \xrightarrow[\varepsilon_0]{}{}^{a} \cdots \xrightarrow[\varepsilon_{m-1}]{}{}^{a} (F_m,\delta_m,N_m,T_m,S_m)\cdots \tag{5}$$

such that $M_m = (N_m N)$. Since this is also a (possibly not maximal) sequence of $\rightarrow^{c,\mathcal{M}}$-transitions, we see, using $M' \Downarrow_{\mathcal{M}}$, that there are two cases:

(1) If this sequence is infinite, we have $\varepsilon_j = \{\circlearrowleft\}$ for an infinite number of indices j, since $M' \Downarrow_{\mathcal{M}}$. Then we are done in this case with (ii) of Definition 5.3. Moreover, we have $\delta_n \in \mathcal{M}_0$ for all n, by induction hypothesis. Let $\rho \in e - e_0$ and $u_\rho \in \mathsf{dom}(\delta_n)$. By induction hypothesis we have $\rho \notin \varepsilon_i \subseteq e_0$ for $i \leqslant n$. Moreover if $F_n = (R_n, L_n)$ we have $\rho \notin R_n - R$ since $e \subseteq \mathsf{dom}(\Delta) \subseteq R$. Then by Corollary 3.5 we have $u_\rho \in \mathsf{dom}(\delta_0)$ and $\delta_n(u_\rho) = \delta_0(u_\rho)$, hence $\delta_n \in \mathcal{M}$. This shows (iii) of Definition 5.3 in this case, where (iv) is vacuously true.

(2) Otherwise, the sequence (5) is finite, of length k, and $N_k \in \mathcal{V}al$. Then by induction hypothesis $\varepsilon_i \subseteq e_0$ and $\delta_i \in \mathcal{M}_0$ for $i \leqslant k$, and $\Delta \models N_k : (\theta \xrightarrow{e_2} \tau)$. Then, as above, if $u_\rho \in \mathsf{dom}(\delta_i)$ with $\rho \in e - e_0$, we have $u_\rho \in \mathsf{dom}(\delta_0)$ and $\delta_i(u_\rho) = \delta_0(u_\rho)$, and this shows $\delta_i \in \mathcal{M}$ for $i \leqslant k$. We distinguish two cases:

(2.1) If $S_k \neq \mathsf{0}$, then $N_k = ()$ and $S_k = \mathbf{E}[\{y \mapsto \nabla y M''\}M'']$ with $N_{k-1} = \mathbf{E}[\nabla y M'']$. In this case the sequence (4) above is

$$\begin{aligned}(F_0,\delta_0,(M'N),T_0,S_0) &\xrightarrow[\varepsilon_0]{}{}^{a} \cdots \xrightarrow[\varepsilon_{k-1}]{}{}^{a} (F_k,\delta_k,(),T_k,(\mathbf{E}[\{y \mapsto \nabla y M''\}M'']N))\\ &\xrightarrow[\varepsilon_k]{}{}^{c,\mathcal{M}} \cdots \xrightarrow[\varepsilon_{n-1}]{}{}^{c,\mathcal{M}} (F_n,\delta_n,M_n,T_n,S_n)\cdots\end{aligned}$$

in such a way that

$$\begin{aligned}(F_0,\delta_0,M',T_0,S_0) &\xrightarrow[\varepsilon_0]{}{}^{c,\mathcal{M}} \cdots \xrightarrow[\varepsilon_{k-1}]{}{}^{c,\mathcal{M}} (F_k,\delta_k,(),T_k,\mathbf{E}[\{y \mapsto \nabla y M''\}M''])\\ &\xrightarrow[\varepsilon_m]{}{}^{c,\mathcal{M}} \cdots \xrightarrow[\varepsilon_{n-1}]{}{}^{c,\mathcal{M}} (F_n,\delta_n,M_n,T_n,S_n)\cdots\end{aligned}$$

is a maximal sequence of $\rightarrow^{c,\mathcal{M}}$-transitions starting from $(F_0,\delta_0,M',T_0,S_0)$, and we easily conclude in this case, where (iv) is vacuously true, using the induction hypothesis, that is $\Delta \models M' : e_0, (\theta \xrightarrow{e_2} \tau)$.

(2.2) If $S_k = \mathsf{0}$, the function M' in $(M'N)$ evaluates to a value $V = N_k$ such that $\Delta \models V : (\theta \xrightarrow{e_2} \tau)$, by induction hypothesis. Then the computation of (MN) reduces into (VN), and continues with the evaluation of the argument N. That is, there exists a (maximal) sequence of $\rightarrow^{a}$-transitions

$$(F_k,\delta_k,N,\mathsf{0},\mathsf{0}) \xrightarrow[\varepsilon_k]{}{}^{a} \cdots \xrightarrow[\varepsilon_{k+l}]{}{}^{a} (F_{k+l},\delta_{k+l},N'_l,T'_l,S_{k+l})\cdots \tag{6}$$

such that $M_{k+i} = (VN'_i)$ and $T_{k+i} = T_k + T'_i$ for all i. If this sequence is infinite, or ends up with a yield operation (unfolding a recursive process), then we argue as above, using the induction hypothesis $\Delta \models N : e_1, \theta$. Otherwise, there exists h such that N'_h is a value W, and $S_i = \mathsf{0}$ for $i \leqslant k+h$. Then we have $\Delta \models W : \theta$ and $\delta_i \in \mathcal{M}_1$ for $k < i \leqslant k+l$ by induction hypothesis. By the same argument as above, we have $\delta_i \in \mathcal{M}$ for $i \leqslant k+l$ (see **(2)** above). In this case the sequence (4) is

$$\begin{aligned}(F_0, \delta_0, (M'N), T_0, S_0) &\xrightarrow[\varepsilon_0]{}{}^a \cdots \xrightarrow[\varepsilon_{k-1}]{}{}^a (F_k, \delta_k, (VN), T_k, 0)\\ &\xrightarrow[\varepsilon_k]{}{}^a \cdots \xrightarrow[\varepsilon_{k+h}]{}{}^a (F_{k+h}, \delta_{k+h}, (VW), T_k + T'_h, 0)\cdots\end{aligned}$$

Since $\Delta \models V : (\theta \xrightarrow{e_2} \tau)$ and $\Delta \models W : \theta$, we have, by definition, $\Delta \models (VW) : e_2, \tau$, and in particular $(VW) \searrow_{\mathcal{M}}$. Then there is a maximal sequence of $\rightarrow^a$-transitions

$$(F_{k+h}, \delta_{k+h}, (VW), 0, 0) \xrightarrow[\varepsilon_{k+h}]{}{}^a \cdots \xrightarrow[\varepsilon_n]{}{}^a (F_n, \delta_n, N''_{n-(k+h)}, T''_{n-(k+h)}, S_n)\cdots$$

such that $M_n = N''_{n-(k+h)}$ and $T_n = T_k + T'_h + T''_{n-(k+h)}$ for $n \geqslant k+h$. If this computation diverges, or ends up with a yield operation, we argue as above. If for some t we have $N''_t \in \mathcal{V}al$ with $S_{k+h+t} = 0$, then the computation (4) is

$$\begin{aligned}(F_0, \delta_0, (M'N), T_0, S_0) &\xrightarrow[\varepsilon_0]{}{}^a \cdots \xrightarrow[\varepsilon_{k-1}]{}{}^a (F_k, \delta_k, (VN), T_k, 0)\\ &\xrightarrow[\varepsilon_k]{}{}^a \cdots \xrightarrow[\varepsilon_{k+h-1}]{}{}^a (F_{k+h}, \delta_{k+h}, (VW), T_k + T'_h, 0)\\ &\xrightarrow[\varepsilon_{k+h}]{}{}^a \cdots \xrightarrow[\varepsilon_t]{}{}^a (F_t, \delta_t, N''_t, T_k + T'_h + T''_t, 0)\cdots\end{aligned}$$

and the computation continues executing some thread from $T_k + T'_h + T''_t$. By Lemma 3.7 we have $T_k + T'_h + T''_t \searrow_{\mathcal{M}}$, and this concludes the proof of (ii) in this case. To see that $\delta_i \in \mathcal{M}$ holds for $i > k+h+t$, one observes that δ_i is actually obtained in a sequence of $\rightarrow^{c,\mathcal{M}}$-transitions starting from M', or from N, or from (VW), and we use the induction hypotheses $\Delta \models M' : e_0, (\theta \xrightarrow{e_2} \tau)$ and $\Delta \models N : e_1, \theta$, and the fact that $\Delta \models (VW) : e_2, \tau$ to conclude $\delta_i \in \mathcal{M}$, thus showing (iii). Finally to show (iv) we use the fact that $\Delta \models (VW) : e_2, \tau$.

$\mathbf{M} = (\mathsf{ref}_\rho \mathbf{N})$. If $M = (\mathsf{ref}_\rho N)$ with $\Delta; \vdash N : e', \theta$ and $e = \{\rho\} \cup e'$, where $\theta = \Delta(\rho)$, we have $\Delta \models N : e', \theta$ by induction hypothesis, and in particular $N \searrow_{\mathcal{M}'}$ where $\mathcal{M}' = \{\, \delta \mid \Delta \models \delta \upharpoonright e' \,\}$. Let $\mathcal{M} = \{\, \delta \mid \Delta \models \delta \upharpoonright e \,\}$ and $\delta \in \mathcal{M}$ and (R, L) be such that $\mathsf{dom}(\Delta) \subseteq R$ and $C = ((R, L), \delta, M, 0, 0)$ is weakly well-formed. Let us consider a maximal sequence of $\rightarrow^{c,\mathcal{M}}$-transitions from this configuration:

$$C = (F_0, \delta_0, M_0, T_0, S_0) \xrightarrow[\varepsilon_0]{}{}^{c,\mathcal{M}} \cdots \xrightarrow[\varepsilon_{n-1}]{}{}^{c,\mathcal{M}} (F_n, \delta_n, M_n, T_n, S_n)\cdots \qquad (7)$$

This sequence starts with a (possibly empty) sequence of computations of N, that is, there is a (maximal) sequence of $\rightarrow^a$-transitions

$$(F_0, \delta_0, N, T_0, S_0) \xrightarrow[\varepsilon_0]{}{}^a \cdots \xrightarrow[\varepsilon_{m-1}]{}{}^a (F_m, \delta_m, N_m, T_m, S_m)\cdots$$

such that $M_m = (\mathsf{ref}_\rho N_m)$. Since $\mathcal{M} \subseteq \mathcal{M}'$ we have $N \searrow_{\mathcal{M}}$, and therefore this sequence either diverges, or ends up with a yield operation, or there exists k such that N_k is a value V and $S_k = 0$. In this latter case, the sequence (7) is

$$\begin{aligned}((R, L), \delta, (\mathsf{ref}_\rho N), 0, 0) &\xrightarrow[\varepsilon_0]{}{}^a \cdots \xrightarrow[\varepsilon_{k-1}]{}{}^a ((R_k, L_k), \delta_k, (\mathsf{ref}_\rho V), T_k, 0)\\ &\xrightarrow[\rho]{}{}^a ((R_k, L_k \cup \{u\}), \delta_k \cup u_\rho \mapsto V, u_\rho, T_k, 0)\end{aligned}$$

where $u = \mathsf{fresh\_loc}(L_k)$. By induction hypothesis, we have $\varepsilon_i \subseteq e'$ and $\delta_i \in \mathcal{M}'$. Assuming that $\rho \notin e'$, if $v_\rho \in \mathsf{dom}(\delta_i)$ we have $v_\rho \in \mathsf{dom}(\delta)$ and $\delta_i(v_\rho) = \delta(v_\rho)$ by Corollary 3.5, since $\rho \in R$, hence $\delta_i \in \mathcal{M}$ for all $i \leqslant k$. Moreover $\Delta \models V : \theta$ by induction hypothesis, and therefore $\delta_k \cup \{u_\rho \mapsto V\} \in \mathcal{M}$. In this case as well as the others, we conclude as in the case of application.

$\mathbf{M} = (\mathbf{!\,N})$. If $M = (!\,N)$ with $\Delta; \vdash N : e', \tau\,\mathsf{ref}_\rho$ and $e = \{\rho\} \cup e'$, we have $\Delta(\rho) = \tau$, and $\Delta \models N : e', \tau\,\mathsf{ref}_\rho$ by induction hypothesis, hence $N \Downarrow_{\mathcal{M}'}$ where $\mathcal{M}' = \{\, \delta \mid \Delta \models \delta \restriction e' \,\}$, hence also $N \Downarrow_{\mathcal{M}}$ where $\mathcal{M} = \{\, \delta \mid \Delta \models \delta \restriction e \,\}$. Given R and $\delta \in \mathcal{M}$ such that $\mathsf{dom}(\Delta) \subseteq R$ and $C = ((R, L), \delta, (!\,N), 0, 0)$ is weakly well-formed, any maximal $\rightarrow^{c,\mathcal{M}}$-computation of C starts with a maximal $\rightarrow^{a}$-computation of $((R, L), \delta, N, 0, 0)$. If this computation diverges, or end up with a yield operation, we argue as in the previous cases. Otherwise, there is a value V such that

$$((R, L), \delta, N, 0, 0) \xrightarrow[\varepsilon]{*}{}^{a} (F, \delta', V, T, 0)$$

and therefore $\Delta \models V : \tau\,\mathsf{ref}_\rho$ by induction hypothesis, that is, V is a reference u_ρ. By induction hypothesis, we also have $\delta' \in \mathcal{M}'$. If $\rho \notin \varepsilon$, we have $u_\rho \in \mathsf{dom}(\delta)$ and $\delta'(u_\rho) = \delta(u_\rho)$, hence $\Delta \models \delta'(u_\rho) : \tau$ in this case since $\delta \in \mathcal{M}$. Otherwise, that is if $\rho \in \varepsilon$, we have $\rho \in e'$ and therefore $\Delta \models \delta'(u_\rho) : \tau$ also in this case since $\delta \in \mathcal{M}'$. This shows (iv) of Definition 5.3 in this case. The remaining points of Definition 5.3 are shown as above. The proof is similar in the case where $M = (M' := N)$, and similar to the case of $M = (M'N)$.

$\mathbf{M} = (\mathsf{local}\ \rho\ \mathsf{in}\ \mathbf{N})$. In this case we have $\Delta = \Delta_0, \Delta_1$ and $\Delta_0, \rho : \theta, \Delta_1; \vdash N : e', \tau$ with $e = e' - \{\rho\}$. Reducing the expression M we get $(\{\rho \mapsto \varrho\}N \backslash \varrho)$, where $\varrho \in \mathcal{R}eg_{un} - R$, and $\{\rho \mapsto \varrho\}N$ has a typing $\Delta_0, \varrho : \theta, \Delta_1; \vdash \{\rho \mapsto \varrho\}N : \{\rho \mapsto \varrho\}e', \tau$ with the same structure as the typing of N. We can then use the induction hypothesis, that is $\Delta_0, \varrho : \theta, \Delta_1 \models \{\rho \mapsto \varrho\}N : \{\rho \mapsto \varrho\}e', \tau$ to conclude. The details are left to the reader, as well as the cases of $M = (N \backslash \varrho)$ and $M = (\mathsf{thread}\, N)$. ❏

We can now prove our main result, namely a Type Safety theorem, which improves upon the standard statement:

Theorem (Type Safety) 5.16. *Let M be a closed typable expression, with $\mathsf{ref}(M) = \emptyset$. Then the configuration $C = ((\mathsf{reg}(M), \emptyset), \emptyset, M, 0, 0)$ is reactive. Moreover, any configuration reachable from C is reactive.*

Proof: we prove that if $C = (F, \delta, M, T, S)$ is a closed, well-formed typable configuration then

(i) C is reactive, and

(ii) any configuration reachable from C is reactive.

We first show (i). We have $\Delta \vdash \delta$ and $\Delta; \vdash M : e_M, \tau$ for some type τ, $\Delta; \vdash T : e_T$ and $\Delta; \vdash S : e_S$ with $e = e_M \cup e_T \cup e_S$. Let $\mathcal{M} = \{\, \delta' \mid \Delta \models \delta' \restriction e \,\}$. Then $\delta \in \mathcal{M}$, and $M \searrow_{\mathcal{M}}$, and also $N \searrow_{\mathcal{M}}$ for any N in T by Proposition 5.15. Then $M + T \searrow_{\mathcal{M}}$ by Lemma 3.7. This shows that C is reactive.

(ii) If $C \xrightarrow[e']{*} C'$ then C' is closed and well-formed, and by Proposition 4.2 C' is typable, and therefore reactive by (i).

Now assume that M is a closed typable expression, that is $\Delta; \vdash M : e, \tau$ for some Δ, e and

τ. By Remark 4.1(ii) we have $\mathsf{reg}(M) \subseteq \mathsf{dom}(\Delta)$. Then we have $((\mathsf{reg}(M), \emptyset), \emptyset, M, \mathsf{0}, \mathsf{0}) \asymp_\emptyset ((\mathsf{dom}(\Delta), \emptyset), \emptyset, M, \mathsf{0}, \mathsf{0})$, and we use the previous points and Lemma 3.8(ii) to conclude. ❏

An obvious consequence of this result is that typable closed expressions written without recursion – either ordinary, that is $\mathsf{rec}\, f(x)M$, or using the yield-and-loop construct $\nabla y M$ – always terminate. This solves a problem raised in [15], regarding the typing of termination leaks. More precisely, we have shown in [15] that the typing of secure information flow (see [33] for a survey on this topic) may be largely improved if we know that some expressions (specifically, the branches in a conditional branching) are terminating, but we left open the problem of designing a typing technique to ensure termination. Our type and effect system provides a solution: for the language of [15], a typable expression that does not exhibit an unfolding effect $\circlearrowleft$ is guaranteed to terminate. The language of [15] did not contain the construct $\nabla y M$. To detect also the kind of non-termination introduced by this construct, the typing system should involve a specific (latent) effect in the typing of $\nabla y M$, similar to $\circlearrowleft$. We have not considered this refinement here, but it should be obvious that making explicit the yield effect does not cause any difficulty.

6. Conclusion

We have proposed a way of ensuring fairness in a cooperative, concurrent, higher-order imperative language, by introducing a specific construct for programming non-terminating processes. Our main technical contribution consists in designing a type and effect system for our language, that supports an extension of the classical realizability technique to show our termination property, namely fairness.

Our study was limited to a very simple core language, and clearly it should be extended to more realistic ones. The synchronization constructs of synchronous, or reactive programming for instance [14, 16, 19, 27, 32, 35] should be added. We believe this should not cause any difficulty. Indeed, the problem with termination in a concurrent higher-order imperative language is in the interplay between functions and store, and between recursion and thread creation.

Another topic that deserves to be investigated is whether the restriction imposed by our stratification of the memory is acceptable in practice. We believe that the restriction we have on the storable functional values is not too severe (in particular, any pure function can be stored), but obviously our design for the type system needs to be extended, and experimented on real applications, in order to assess more firmly this belief. We notice also that our approach does not seem to preclude the use of cyclic data structures. In OCaml for instance, one may define cyclic lists like – using standard notations for the list constructors:

$$(\mathsf{let\ rec}\ x = \mathsf{cons}(1, x)\ \mathsf{in}\ x)$$

Such a value, which is a list of integers, does not show any effect, and therefore it should be possible to extend our language and type and effect system to deal with such circular data structures.

Finally it would be interesting to see whether showing termination in a concurrent, higher-order imperative language may have other applications than the one which motivated our work (we have mentioned such an application in the previous section), and whether our realizabity interpretation could be generalized to logical relations (and to richer notions of type), in order to prove program equivalences for instance. This is left for further investigations.

References

[1] L. Aceto, *A static view of localities,* Formal Aspects of Computing Vol. 6 No. 2 (1994) 201-222.

[2] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, H. R. Douceur, *Cooperative task management without manual stack management or, Event-driven programming is not the opposite of threaded programming,* Usenix ATC (2002).

[3] A. Ahmed, *Step-indexed syntactic logical relations for recursive and quantified types,* ESOP'06, Lecture Notes in Comput. Sci. 3924 (2006) 69-83.

[4] A. J. Ahmed, A.W. Appel, R. Virga, *A stratified semantics of general references embeddable in higher-order logic,* LICS'02 (2002) 75-86.

[5] T. E. Anderson, B. N. Bershad, E. D. Lazowska, H. M. Levy, *Scheduler activations: effective kernel support for the user-level management of parallelism,* ACM Trans. on Computer Systems Vol. 10 No. 1 (1992) 53-79.

[6] A.W. Appel, D. McAllester, *An indexed model of recursive types for foundational proof-carrying code,* ACM TOPLAS Vol. 23 No. 5 (2001) 657-683.

[7] G. Banga, P. Druschel, J.C. Mogul, *Better operating sytem features for faster network servers,* ACM SIGMETRICS Performance Evaluation Review Vol. 26 No. 3 (1998) 23-30.

[8] H. Barendregt, *Lambda Calculi with Types,* in Handbook of Logic in Computer Science, Vol. 2 (S. Abramsky, Dov M. Gabbay & T.S.E. Maibaum, Eds.), Oxford University Press (1992) 117-309.

[9] R. von Berhen, J. Condit, E. Brewer, *Why events are a bad idea (for high-concurrency servers),* Proceedings of HotOS IX (2003).

[10] R. von Berhen, J. Condit, F. Zhou, G.C. Necula, E. Brewer, *Capriccio: scalable threads for Internet services,* SOSP'03 (2003).

[11] N. Benton, B. Leperchey, *Relational reasoning in a nominal semantics for storage,* TLCA'05, Lecture Notes in Comput. Sci. 3461 (2005) 86-101.

[12] L. Birkedal, R. Harper, *Relational interpretation of recursive types in an operational setting,* Information and Computation Vol. 155 No. 1-2 (1999) 3-63.

[13] N. Bohr, L. Birkedal, *Relational reasoning for recursive types and references,* APLAS'06, Lecture Notes in Comput. Sci. 4279 (2006) 79-96.

[14] G. Boudol, ULM, *a core programming model for global computing,* ESOP'04, Lecture Notes in Comput. Sci. 2986 (2004) 234-248.

[15] G. Boudol, *On typing information flow,* Intern. Coll. on Theoretical Aspects of Computing, Lecture Notes in Comput. Sci. 3722 (2005) 366-380.

[16] F. Boussinot, *FairThreads: mixing cooperative and preemptive threads in C,* Concurrency and Computation: Practice and Experience, Vol. 18 (2006) 445-469.

[17] K. Crary, R. Harper, *Syntactical logical relations for polymorphic and recursive types,* G. Plotkin's Festschrift, ENTCS Vol. 172 (2007) 259-299.

[18] F. Dabrowski, F.Boussinot, *Cooperative threads and preemptive computations,* Proceeding of TV'06, Workshop on Multithreading in Hardware and Software: Formal Approaches to Design and Verification, FLoC'06 (2006).

[19] S. Epardaud, *Mobile reactive programming in ULM,* Proc. of the Fifth ACM SIGPLAN Workshop on Scheme and Functional Programming (2004) 87-98.

[20] J.-C. Filliâtre, *Verification of non-functional programs using interpretations in type theory,* J. Functional Programming Vol. 13 No. 4 (2003) 709-745.

[21] J.-Y. Girard, Y. Lafont, P. Taylor, *Proofs and Types,* Cambridge Tracts in Theoretical Computer Science 7, Cambridge University Press (1989).

[22] S.C. Kleene, *On the interpretation of intuitionistic number theory,* J. of Symbolic Logic, Vol. 10 (1945) 109-124.

[23] J.-L. Krivine, *Lambda-Calcul: Types et Modèles,* Masson, Paris (1990). English translation "Lambda-Calculus, Types and Models", Ellis Horwood (1993).

[24] P.J. Landin, *The mechanical evaluation of expressions,* Computer Journal Vol. 6 (1964) 308-320.

[25] P.B. Levy, *Possible world semantics for general storage in call-by-value,* CSL'02, Lecture Notes in Comput. Sci. 2471 (2002) 232-246.

[26] J.M. Lucassen, D.K. Gifford, *Polymorphic effect systems,* POPL'88 (1988) 47-57.

[27] L. Mandel, M. Pouzet, *ReactiveML, a reactive extension to ML,* PPDP'05 (2005) 82-93.

[28] J.C. Mitchell, *Foundations for Programming Languages,* MIT Press (1996).

[29] J. Ousterhout, *Why threads are a bad idea (for most purposes),* presentation given at the 1996 Usenix ATC (1996).

[30] A. Pitts, I. Stark, *Operational reasoning for functions with local state,* in Higher-Order Operational Techniques in Semantics (A. Gordon and A. Pitts, Eds), Publications of the Newton Institute, Cambridge Univ. Press (1998) 227-273.

[31] G. Plotkin, *Lambda-definability and logical relations,* Memo SAI-RM-4, University of Edinburgh (1973).

[32] R. Pucella, *Reactive programming in Standard ML,* IEEE Intern. Conf. on Computer Languages (1998) 48-57.

[33] A. Sabelfeld, A.C. Myers, *Language-based information-flow security,* IEEE J. on Selected Areas in Communications Vol. 21 No. 1 (2003) 5-19.

[34] D. Sangiorgi, *Termination of processes,* Math. Struct. in Comp. Science Vol. 16 (2006) 1-39.

[35] M. Serrano, F. Boussinot, B. Serpette, *Scheme fair threads,* PPDP'04 (2004) 203-214.

[36] J.-P. Talpin, P. Jouvelot, *The type and effect discipline,* Information and Computation Vol. 111 (1994) 245-296.

[37] W. Tait, *Intensional interpretations of functionals of finite type I,* J. of Symbolic Logic 32 (1967) 198-212.

[38] W. Tait, *A realizability interpretation of the theory of species,* Logic Colloquium, Lecture Notes in Mathematics 453 (1975) 240-251.

[39] M. Tofte, J.-P. Talpin, *Region-based memory management,* Information and Computation Vol. 132, No. 2 (1997) 109-176.

[40] M. Welsh, D. Culler, E. Brewer, *SEDA: an architecture for well-conditioned, scalable internet services,* SOSP'01 (2001) 230-243.

[41] A. Wright, M. Felleisen, *A syntactic approach to type soundness,* Information and Computation Vol. 115 No. 1 (1994) 38-94.

[42] N. Yoshida, M. Berger, K. Honda, *Strong normalisation in the π-calculus,* Information and Computation Vol. 191 No. 2 (2004) 145-202.