SOS 2005 Preliminary Version

Adding recursion to DPI (Extended abstract)

Samuel Hym

PPS, Université Paris 7 & CNRS

Matthew Hennessy

Department of Informatics, University of Sussex

Abstract

DPI is a distributed version of the PI-CALCULUS, in which processes are explicitly located, and a migration construct may be used for moving between locations. We argue that adding a recursion operator to the language increases significantly its descriptive power. But typing recursive processes requires the use of potentially infinite types.

We show that the capability-based typing system of DPI can be extended to *co-inductive types* so that recursive processes can be successfully supported. We also show that, as in the PI-CALCULUS, recursion can be implemented via iteration. This translation improves on the standard ones by being compositional but still comes with a significant migration overhead in our distributed setting.

1 Introduction

The PI-CALCULUS, [7,8], is a well-known formal calculus for describing, and reasoning about, the behaviour of concurrent processes which interact via two-way communication channels. DPI, [4], is one of a number of extensions in which processes are *located*, and may migrate between locations, or sites, by executing an explicit migrate command; the agent $\mathsf{goto}\, k.P$, executing at a site l, will continue with the execution of P at the site k. This extension comes equipped with a sophisticated capability-based type system, and a co-inductive behavioural theory which takes into account the constraints imposed by these types, [3,4]. The types informally correspond to sets of *capabilities*, and the use a process may make of an entity, such as a location or a channel, depends on the current type at which the process owns the entity. Moreover this type may change over time, reflecting the fact that processes may gradually accumulate capabilities over entities.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

The most common formulations of the pi-calculus use iteration (written $* P$ for the iteration of P) to describe repetitive processes. Thus

$$* \, c\,?(x)\, d\,!\langle x\rangle$$

represents a process which repeatedly inputs a value on channel c and outputs it at d. An alternative would be to use an explicit recursion operator, leading to definitions such as

$$\mathsf{rec}\; Z.\; c\,?(x)\, d\,!\langle x\rangle\; Z$$

But it has been argued that explicit recursion is unnecessary, because it offers no extra convenience over iteration; indeed it is well-known that such a recursion operator can easily be implemented using iteration; see pages 132–138 in [8].

However the situation changes when we move to the distributed world of Dpi. In Section 2 we demonstrate that the addition of explicit recursion leads to powerful programming techniques; in particular it leads to simple natural descriptions of processes for searching the underlying network for sites with particular properties.

Unfortunately this increase in descriptive power is obtained at a price. In order for these recursive processes to be accommodated within the typed framework of Dpi, we need to extend the type system with *co-inductive types*, that is types of potentially infinite depth.

The purpose of this paper is to

- demonstrate the descriptive power of recursion when added to Dpi;
- develop a system of co-inductive types which support recursive processes;
- prove that at the cost of significant migration costs recursion in Dpi can still be implemented by purely iterative processes, in the absence of network failures.

In Section 2 we describe the extension to Dpi, called recDpi, and demonstrate the power of recursion by a series of prototypical examples. This is followed in Section 3 with an outline of how the co-inductive types are defined, and how the typing system for Dpi can be easily extended to handle these new types. The translation of recursive processes into iterative processes is explained in Section 4, and we outline the proof of correctness in Section 5. This requires the use of a *typed* bisimulation equivalence to accommodate the typed labelled transition system for recDpi.

The paper relies heavily on existing work on Dpi, and the reader is referred to papers such as [3,4] for detailed explanations of both the semantics of Dpi and its typing system.

Fig. 1 Syntax of recDpi

M, N ::=	*Systems*
$l[\![P]\!]$	Located Process
$M \mid N$	Composition
$(\mathsf{new}\, e : \mathsf{E})\, M$	Name Creation
$\mathbf{0}$	Termination

P, Q ::=	*Processes*
$u\,!\langle V\rangle\, P$	Output
$u\,?(X : \mathsf{T})\, P$	Input
$\mathsf{goto}\, v.P$	Migration
$\mathsf{if}\ u_1 = u_2\ \mathsf{then}\ P\ \mathsf{else}\ Q$	Matching
$(\mathsf{newc}\, c : \mathsf{C})\, P$	Channel creation
$(\mathsf{newreg}\, n : \mathsf{N})\, P$	Global name creation
$(\mathsf{newloc}\, k : \mathsf{K})\, P$	Location creation
$P \mid Q$	Composition
stop	Termination
$*\, P$	Iteration
$\mathsf{here}\, [x]\, P$	**Location look up**
$\mathsf{rec}\ (Z : \mathsf{R}).\ P$	**Recursion**
Z	**Recursion variable**

2 The language recDpi

The syntax of recDpi is given in Figure 1, and is a simple extension of that of Dpi; the new constructs are highlighted in bold font. As usual it assumes a set of *names*, ranged over by letters such as $a, b, c,\ k, l, \ldots$, and a separate set of *variables*, ranged over by $x, y, z, \ldots$; to handle recursive processes we use another set of *recursion variables*, ranged over by $X, Y, Z, \ldots$ The values in the language include *identifiers*, that is names or variables, and *addresses*, of the form $u@w$; intuitively w stands for a location and u a channel located there. In the paper we will consider only closed terms, where all variables (recursion included) are bound.

The most important new construct is that for typed recursive processes, $\mathsf{rec}\ (Z : \mathsf{R}).\ P$; as we shall see the type R dictates the requirements on any site wishing to host this process. We also have a new construct $\mathsf{here}\, [x]\, P$, which allows a process to know its current location.

Example 2.1 [Searching a network] Consider the following recursive process, which searches a network for certain values satisfying some unspecified predicate p:

$$\begin{aligned}\mathsf{Search} \triangleq \mathsf{rec}\ Z : \mathsf{S}.\ & \mathit{test}\,?(x) \mathsf{if}\ p(x)\ \mathsf{then}\ \mathsf{goto}\, \mathsf{home}.\mathit{report}\,!\langle x\rangle \\ & \mathsf{else}\ \mathit{neigh}\,?(y)\, \mathsf{goto}\, y.Z\end{aligned}$$

When placed at a specific site such as k, giving the system

$$k[\![\mathsf{Search}]\!],$$

the process first gets the local value from the channel *test*. If it satisfies the test the search is over; the process returns home, and *reports* the value. Otherwise it uses the local channel *neigh* to find a neighbour to the current site, migrates there and launches a recursive call at this new site. ■

We refrain from burdening the reader with a formal reduction semantics for recDpi, as it is a minor extension of that of Dpi. However in Section 4 we give a typed labelled transition system for the language, the τ-moves of which provides our reduction semantics (see Figure 3). For the current discussion we can focus on the following rules:

(lts-here)
$$k[\![\mathsf{here}\,[x]\,P]\!] \xrightarrow{\tau} k[\![P[^k/_x]]\!]$$
(lts-iter)
$$k[\![* \,P]\!] \xrightarrow{\tau} k[\![* \,P]\!] \mid k[\![P]\!]$$
(lts-rec)
$$k[\![\mathsf{rec}\ (Z:\mathsf{R}).\ P]\!] \xrightarrow{\tau} k[\![P\{^{\mathsf{rec}\ (Z:\mathsf{R}).\ P}/_Z\}]\!]$$

The first simply implements the capture of the current location by the construct here. The second states that the iterative process at k, $k[\![* \,P]\!]$ can spawn a new copy $k[\![P]\!]$, while retaining the iterated process. This means that every new copy of this process will be located in k. The final one, (lts-rec), implements recursion in the standard manner by unwinding the body, which is done by replacing every free occurrence of the recursion variable Z in P by the recursive process itself. This takes an explicit τ-reduction to match the rule (lts-iter).

Example 2.2 [Self-locating processes] We give an example to show why the construct here is particularly interesting for recursive processes. Consider the system $k[\![\mathsf{Quest}]\!]$ where

$$\begin{aligned}\mathsf{Quest} \triangleq \mathsf{rec}\ Z:\mathsf{R}.\ \mathsf{here}\,[x]\,(\mathsf{newc}\ ans)\ \ neigh\,?(y:\mathsf{R})\\ (ans\,?(news)\ldots \mid \mathsf{goto}\ y.req\,!\langle \mathsf{data}, ans@x\rangle\ Z)\end{aligned}$$

After determining its current location x, this process generates a new local channel *ans* at the current site k, and sets up a listener on this channel to await news. Concurrently it finds a neighbour, via the local channel *neigh*. It then migrates to this neighbour and poses a question there, via the channel *req*, and fires a new recursive call, this time at the neighbouring site. The neighbour's request channel *req* requires some data, data, and a return address, which in this case is given via the value $ans@x$.

Note that at runtime the occurrence of x in the value proffered to the channel *req* is substituted by the originating site k. After the first three steps

Fig. 2 Recursive pre-types

Base Types:	$\mathsf{B} ::= \textbf{int} \mid \textbf{bool} \mid \textbf{unit} \ldots$
Local Channel Types:	$\mathsf{A} ::= \text{R}\langle \mathsf{U}\rangle \mid \text{W}\langle \mathsf{T}\rangle \mid \text{RW}\langle \mathsf{U}, \mathsf{T}\rangle$
Capability Types:	$\mathsf{C} ::= u : \mathsf{A}$
Location Types:	$\mathsf{K} ::= \text{LOC}[\mathsf{C}_1, \ldots, \mathsf{C}_n], n \geq 0 \mid \mu \mathbf{Y}.\mathsf{K} \mid \mathbf{Y}$
Registered Name Types:	$\mathsf{G} ::= \text{RC}\langle \mathsf{A}\rangle$
Value Types:	$\mathsf{V} ::= \mathsf{B} \mid \mathsf{A} \mid (\tilde{\mathsf{A}})@u \mid (\tilde{\mathsf{A}})@\mathsf{K}$
Transmission Types:	$\mathsf{T}, \mathsf{U} ::= (\mathsf{V}_1, \ldots, \mathsf{V}_n), n \geq 0$

in the reduction of the system $k[\![\mathsf{Quest}]\!]$, we get to

$$(\mathsf{new}\, ans)\, k[\![neigh\,?(y : \mathsf{R})\,(\,\mathsf{goto}\, y.req\,!\langle \mathsf{data}, ans@k\rangle\, \mathsf{Quest} \mid ans\,?(news) \ldots)]\!]$$

If k's neighbour is l, this further reduces to (up to some reorganisation)

$$\begin{aligned}&(\mathsf{new}\, ans)\, k[\![ans\,?(news)\ldots]\!] \mid l[\![Q]\!]\\ &\mid (\mathsf{new}\, ans')\, l[\![neigh\,?(y : \mathsf{R})\,(\,\mathsf{goto}\, y.req\,!\langle \mathsf{data}, ans'@l\rangle\, \mathsf{Quest} \mid ans'\,?(news) \ldots)]\!]\end{aligned}$$

with Q some code running at l to answer the request brought by Quest.

The here construct can also be used to write a process initialising a doubly linked list starting from a simply linked one. We assume for this that the cells are locations containing two specific channels: n to get the name of the next cell in the list, p for the previous. The initial state of our system is

$$l_0[\![n\,!\langle l_1\rangle]\!] \mid l_1[\![n\,!\langle l_2\rangle]\!] \mid \ldots$$

and we run the following code in the first cell of this network to initialise the list:

$$\mathsf{rec}\; Z : R.\; n\,?(n')\; \mathsf{here}\,[p']\,(n\,!\langle n'\rangle \mid \mathsf{goto}\, n'.(p\,!\langle p'\rangle \mid Z))$$

■

Now we need to look more closely at the types, like R, involved in the recursive construct.

3 Co-inductive types for recDpi

There is a well-established capability-based type system for DPI, [4], which we can adapt to RECDPI.

3.1 The Types

In this type system local channels have read/write types of the form $\text{R}\langle \mathsf{U}\rangle$, $\text{W}\langle \mathsf{T}\rangle$, or $\text{RW}\langle \mathsf{U}, \mathsf{T}\rangle$ (meaning that values are written at type T and read at

type U on a channel of that type), provided the object types U and T "agree", as will be explained later. Locations have record types, of the form

$$\textsc{loc}[u_1 : \mathsf{A}_1, \ldots, u_n : \mathsf{A}_n]$$

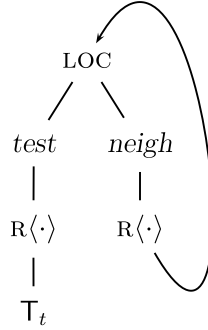
indicating that the local channels u_i may be used at the corresponding type A_i.

However with recursive processes it turns out that we need to consider *infinite* location types. To see this consider again the searching process Search from Example 2.1. Any site, such as k, which can support this process needs to have a local channel called *neigh* from which values can be read. These values must be locations, and let us consider their type, that is the object type of *neigh*. These locations must have a local channel called *test*, of an appropriate type, and a local channel called *neigh*; the object type of this local channel must be in turn the same as the type we are trying to describe. Using a recursion operator μ, this type can be described as

$$\mu\mathbf{Y}.\textsc{loc}[\mathit{test} : \textsc{r}\langle \mathsf{T}_t \rangle, \ \ \mathit{neigh} : \textsc{r}\langle \mathbf{Y} \rangle]$$

which will be used as the type S in the definition of Search; it describes precisely the requirements on any site wishing to host this process.

The set of recursive pre-types is given in Figure 2, and is obtained by adding the operator $\mu\mathbf{Y}.\mathsf{K}$ as a constructor to the type formation rules for Dpi. Following [8] we can associate with each recursive pre-type T a co-inductive pre-type denoted $\mathsf{Tree}(\mathsf{T})$, which takes the form of a finite-branching, but possibly infinite, tree whose nodes are labelled by the type constructors. For example $\mathsf{Tree}(\mathsf{S})$ is the infinite tree represented by the following graph:



Definition 3.1 [Contractive and Tree pre-type] We call a recursive pre-type S *contractive* if for every $\mu\mathbf{Y}.\mathsf{S}'$ it contains, $\mathbf{Y}$ can only appear in S' under an occurrence of $\textsc{loc}$. In the paper we will only consider contractive pre-types.

For every contractive S we can define $\mathsf{Tree}(\mathsf{S})$, the unique tree satisfying by the following equations:

- unwinding recursive pre-types $\mathsf{Tree}(\mu\mathbf{Y}.\mathsf{S}') = \mathsf{Tree}(\mathsf{S}'\{\!\!\{^{\mu\mathbf{Y}.\mathsf{S}'}/_{\mathbf{Y}}\}\!\!\})$
- not modifying any other construct; for instance $\mathsf{Tree}(\textsc{r}\langle \mathsf{U} \rangle) = \textsc{r}\langle \mathsf{Tree}(\mathsf{U}) \rangle$

We call $\mathsf{Tree}(\mathsf{S})$ the *tree pre-type* associated with the recursive pre-type S. ■

Note that $\mathsf{Tree}(\mathsf{S})$ might not be defined when the recursive pre-type S is not contractive.

To go from pre-types to types, we need to get rid of meaningless pre-types like $\text{RW}\langle \text{R}\langle\rangle, \mathbf{int}\rangle$, which would be the type of a channel on which integers are written but channels are read. This is achieved using a notion of subtype, and demanding that, in types of the form $\text{RW}\langle \mathsf{U}, \mathsf{T}\rangle$, T must be a subtype of U.

In Figure A.1 (in the appendix) we give the standard set of rules which define the subtyping relation used in DPI; a typical rule, an instance of (SUB-CHAN), takes the form

$$\frac{\mathsf{T} <: \mathsf{U} <: \mathsf{U}'}{\text{RW}\langle \mathsf{U}, \mathsf{T}\rangle <: \text{R}\langle \mathsf{U}'\rangle}$$

However here we interpret these rules co-inductively, [2]. Formally they give rise to a transformation on relations over tree pre-types. If $\mathcal{R}$ is such a relation, then $\mathsf{Sub}(\mathcal{R})$ is the relation given by:

$$\begin{aligned} \mathsf{Sub}(\mathcal{R}) = &\{(\mathbf{base}, \mathbf{base})\} \\ &\cup \{(u : \mathsf{A}, u : \mathsf{B}) \text{ if } (\mathsf{A}, \mathsf{B}) \text{ is in } \mathcal{R}\} \\ &\cup \{((\widetilde{\mathsf{C}}), (\widetilde{\mathsf{C}'})) \text{ if } (\mathsf{C}_i, \mathsf{C}'_i) \text{ is in } \mathcal{R} \text{ for all } i\} \\ &\cup \ldots \end{aligned}$$

Intuitively, if the hypotheses of any rule of Figure A.1 are in $\mathcal{R}$ the conclusion is in $\mathsf{Sub}(\mathcal{R})$.

Definition 3.2 [Subtyping and types] We define the *subtyping* relation between tree pre-types to be the greatest fixpoint of the function Sub, written $\nu\mathsf{Sub}$. For convenience we often write $\mathsf{T} <: \mathsf{T}'$ to mean that $(\mathsf{T}, \mathsf{T}')$ is in $\nu\mathsf{Sub}$.

Then a tree pre-type is called a *tree type* if every occurrence of $\text{RW}\langle \mathsf{U}, \mathsf{T}\rangle$ it contains satisfies $\mathsf{T} <: \mathsf{U}$.

Finally this is lifted to recursive pre-types. A pre-type T from Figure 2 is called a *recursive type* if $\mathsf{Tree}(\mathsf{T})$ is a tree type. ■

The co-inductive definition of subtyping gives rise to a natural co-inductive proof method, the dual of the usual inductive proof method used for sub-typing in DPI. This can be employed to derive many of the required properties of sub-typing in RECDPI. Here is a typical example. Let us write $\mathsf{T}_1 \downarrow \mathsf{T}_2$ to mean that there is some T such that $\mathsf{T} <: \mathsf{T}_1$ and $\mathsf{T} <: \mathsf{T}_2$, that is T_1 and T_2 are compatible.

Lemma 3.3 *The set of tree types, ordered by $<:$, has partial meets. That is $\mathsf{T}_1 \downarrow \mathsf{T}_2$ implies T_1 and T_2 have a meet, denoted $\mathsf{T}_1 \sqcap \mathsf{T}_2$.*

In the full version of the paper, [5], we go on to show that this result also applies to recursive types, and moreover give a procedure for calculating the meet of any two compatible recursive types.

3.2 Typing Systems

With these types we can now adapt the typing system for Dpi to recDpi. At the system level the judgements take the form

$$\Gamma \vdash M$$

and the rules used are identical to those for Dpi, see Figure 11 in the full version of this paper, [5], where they are restated. The main rule, there, is

$$\text{(t-proc)}\quad \frac{\Gamma \vdash_k P}{\Gamma \vdash k[\![P]\!]}$$

which in turn requires a set of inference rules for the judgements

$$\Gamma \vdash_k P$$

indicating that the process P is well-typed to run at location k. Once more most of these rules are inherited from Dpi, see Figure 12 in [5], and we concentrate here on explaining the three new rules required for recursion and the here construct. The latter is straightforward:

$$\text{(t-here)}\quad \frac{\Gamma \vdash_w P[{}^{w}\!/\!_{x}]}{\Gamma \vdash_w \mathsf{here}\,[x]\,P}$$

However in order to derive judgements about recursive processes, such as

$$\Gamma \vdash_k \mathsf{rec}\,(Z : \mathsf{R}).\,P \tag{1}$$

we need to augment the type environments used in Dpi with entries for recursion variables. Recall that here the type R is a location type, such as $\text{LOC}[u_1 : \mathsf{A}_1, \ldots u_n : \mathsf{A}_n]$, indicating the minimal requirements on any location wishing to host a call to the recursive procedure. So in some way we want to consider recursion variables in the same manner as locations. But we must be careful as subtyping can not be allowed on these variables, unlike locations. Therefore we only allow unique entries of the form $Z : \mathsf{K}$, where K is a location type, in type environments. Then the natural rule for typechecking a recursive call, that is an occurrence of a recursion variable, is given by:

$$\text{(t-recvar)}\quad \frac{\Gamma \vdash w : \Gamma(Z)}{\Gamma \vdash_w Z}$$

In order to typecheck a recursive definition, such as (1) above, we need to

- check that k has at least the capabilities required in R, that is $\Gamma \vdash k : \mathsf{R}$;

- ensure that the body P only uses the resources given in R.

To check this second point we again look at recursion variables as *locations*, and check that P is well-typed to run "in the location Z", which has all the resources mentioned in the type R. The final rule is

$$\frac{\begin{array}{l}\text{(T-REC)}\\ \Gamma \vdash w : \mathsf{R} \\ \Gamma, \langle\!\langle Z : \mathsf{R} \rangle\!\rangle \vdash_Z P\end{array}}{\Gamma \vdash_w \mathsf{rec}\ (Z : \mathsf{R}).\ P}$$

where $\Gamma, \langle\!\langle Z : \mathsf{R} \rangle\!\rangle$ is a notation extending Γ with the information that Z has all the capabilities in R.

Example 3.4 Referring back to Example 2.1 let us see how these rules can be used to infer $\Gamma \vdash_k \mathsf{Search}$, assuming that Γ knows about locations home, k, etc. and their channels. So, by (T-REC), this will amount to:

$$\begin{array}{l}\Gamma, \langle\!\langle Z : \mathsf{S} \rangle\!\rangle \vdash_Z \mathit{test}?(x) \mathsf{if}\ p(x)\ \mathsf{then}\ \mathsf{goto}\ \mathsf{home}.\mathit{report}!\langle x \rangle \\ \qquad\qquad\qquad\qquad\qquad \mathsf{else}\ \mathit{neigh}?(y)\ \mathsf{goto}\ y.Z\end{array}$$

which, in turn, will mainly consist of proving:

$$\begin{array}{l}\Gamma, \langle\!\langle Z : \mathsf{S} \rangle\!\rangle \vdash \mathit{test} : \mathrm{R}\langle \mathsf{T}_t \rangle @ Z \\ \Gamma, \langle\!\langle Z : \mathsf{S} \rangle\!\rangle, x : \mathsf{T}_t \vdash \mathit{neigh} : \mathrm{R}\langle \mathsf{S} \rangle @ Z \\ \Gamma, \langle\!\langle Z : \mathsf{S} \rangle\!\rangle, x : \mathsf{T}_t, y : \mathsf{S} \vdash_y Z\end{array}$$

These statements are provable since S is $\mu\mathbf{Y}.\mathrm{LOC}[\mathit{test} : \mathrm{R}\langle \mathsf{T}_t \rangle,\ \ \mathit{neigh} : \mathrm{R}\langle \mathbf{Y} \rangle]$; the first one is obvious, while the second will be true by unfolding once the recursive type, which means in turn that the third will also be true. ■

The partial view of recursion variables as locations complicates somewhat the formal rules for the construction of valid environments. In this extended abstract we do not go into the details. But for completeness sake we give the formation rules in the appendix, in Figure A.2, together with these for value typing, in Figure A.3. Notice that value typing rules allow statements of the form $\Gamma \vdash Z : \mathrm{LOC}$, required when typing a process "at Z", even if, syntactically, recursion variables cannot be used as values.

The main new technical property of the type inference system is given by:

Lemma 3.5 (Recursion Variable Substitution) *Let us suppose that* $\Gamma \vdash_w$ *rec* $Z : R.\ P$. *Then* $\Gamma \vdash_w P\{^{\mathit{rec}\ Z:R.\ P}/_Z\}$.

This in turn leads to:

Lemma 3.6 (Subject Reduction) $\Gamma \vdash M$ *and* $M \xrightarrow{\tau} M'$ *implies that* $\Gamma \vdash M'$.

4 Implementing recursion using iteration

The problem of implementing recursion using iteration in Dpi, contrary to the pi-calculus, is that any code of the form $k[\![* P]\!]$ will force every instance of P to be launched at the originating site k; this is in contrast to $k[\![\mathsf{rec}\ (Z : \mathsf{R}).\ P]\!]$ where the initial instance of the body P is launched at k but subsequent instances may be launched at arbitrary sites, provided they are appropriately typed.

Nevertheless, at the expense of repeated migrations, we can mimic the behaviour of a recursive process using iteration by designating a *home base* to which the process must return before a new instance is launched. For example if *home* is deemed to be the home base then we can implement our example $k[\![\mathsf{Search}]\!]$ using

$$home[\![* \ \mathsf{IterSearch}]\!] \mid k[\![\mathsf{FireOne}]\!]$$

where

$$\begin{aligned} \mathsf{IterSearch} &\triangleq ping?(l)\, \mathsf{goto}\, l.test?(x)\ \mathsf{if}\ p(x)\ \mathsf{then}\ \mathsf{goto}\, \mathsf{home}.report!\langle x\rangle \\ &\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ neigh?(y)\, \mathsf{goto}\, y.\mathsf{FireOne} \\ \mathsf{FireOne} &\triangleq \mathsf{here}\,[l]\, \mathsf{goto}\, home.ping!\langle l\rangle \end{aligned}$$

With this example, we can easily see how the translation will mimic the original process step by step: the body of the process is left unmodified, only the recursion parts are changed, by implementing the recursive call with a few reductions. FireOne is the "translation" for the recursive calls, which means going to the home base and firing a new instance. This shows why the construct here is necessary: the translation for recursive calls needs to detect its current location to indeed trigger the new instance in the "proper" context. Then the replicated IterSearch starts off by migrating to the actual location where it will run.

This approach underlies our general translation of recursive processes into iterative processes, which we now explain.

As we want to ensure that our translation will be compositional, we will have to dynamically generate the home bases for iterative processes where, in the example IterSearch, the home base and the replicated process were already set up. We will also dynamically generate the registered channel *ping* used to provide to a new instance of the process the name of the location where the recursive call took place. The last thing to do when the recursion is unwound for the first time is to start the iterative process, which means two things: move the code that will be replicated to its home base and fire the first instance. As we explained with the example, the replicated code will just have to wait for the name of a location when the recursion is unwound, go

there and behave as the recursive process. So our translation looks like this:

$$\begin{aligned}
\textsc{unrec}(\mathsf{rec}\ Z : \mathsf{R}.\ P) = {} & (\mathsf{newreg}\ \mathit{ping}_Z : \textsc{rc}\langle\textsc{rw}\langle\mathsf{R}\rangle\rangle) \\
& \quad (\mathsf{newloc}\ \mathit{home}_Z : \textsc{loc}[\mathit{ping}_Z : \textsc{rc}\langle\textsc{rw}\langle\mathsf{R}\rangle\rangle]) \\
& \qquad (\textsc{unrec}(Z) \mid \\
& \qquad\quad \mathsf{goto}\ \mathit{home}_Z.*\ \mathit{ping}_Z?(l : \mathsf{R})\ \mathsf{goto}\ l.\textsc{unrec}(P)) \\
\textsc{unrec}(Z) = {} & \mathsf{here}\,[x]\ \mathsf{goto}\ \mathit{home}_Z.\mathit{ping}_Z!\langle x\rangle
\end{aligned}$$

Of course, any construct other than recursion is left unmodified by this translation; for example $\textsc{unrec}(u!\langle V\rangle\, P) = u!\langle V\rangle\, \textsc{unrec}(P)$.

We stress the fact that this translation heavily relies on migration to mimic the original process. We conjecture that in a Dpi setting where locations or links can fail, like in [1], it would not be possible to get a reasonable encoding of recursion into iteration.

We could also give another translation, which would be closer to the one proposed for the pi-calculus in [8] by:

- closing the free names of recursive processes, and then communicating their actual values through the channel *ping*, at the same time as the location;
- creating all the home bases at the top-level of the process, once and for all.

But such an approach would not be compositional.

Now that we have described our translation, we want to prove that the translation and the original process are "equivalent", in some sense. Since we are in a typed setting, the first property we need to check is the following.

Lemma 4.1 $\Gamma \vdash M$ *if and only if* $\Gamma \vdash \textsc{unrec}(M)$

We can also show that the behaviour of M and that of its translation $\textsc{unrec}(M)$ are closely related. Intuitively we want to show that whenever $\Gamma \vdash M$ then any observer, or indeed other system, which uses names according to the type constraints given in Γ can not differentiate between M and $\textsc{unrec}(M)$. This idea has been formalised in [3] as a typed version of *reduction barbed congruence*, giving rise to the judgements

$$\Gamma \models M \cong_{rbc} N$$

The reader is referred to [3] for the formal details.

Theorem 4.2 *Suppose* $\Gamma \vdash M$. *Then* $\Gamma \models M \cong_{rbc} \textsc{unrec}(M)$.

The proof uses a characterisation of this relation as a bisimulation equivalence in a labelled transition system in which:

- the states are *configurations* of the form $\Gamma \triangleright M$;
- the actions take the form $\Gamma \triangleright M \xrightarrow{\mu} \Gamma' \triangleright M'$; these are based on the labelled transitions system given in Figure 3 and 4.

Fig. 3 Labelled transition semantics. Internal actions.

(LTS-GO)
$\Gamma \triangleright k[\![\mathsf{goto}\, l.P]\!] \xrightarrow{\tau}_\beta \Gamma \triangleright l[\![P]\!]$

(LTS-SPLIT)
$\Gamma \triangleright k[\![P \mid Q]\!] \xrightarrow{\tau}_\beta \Gamma \triangleright k[\![P]\!] \mid k[\![Q]\!]$

(LTS-ITER)
$\Gamma \triangleright k[\![* \; P]\!] \xrightarrow{\tau}_\beta \Gamma \triangleright k[\![* \; P]\!] \mid k[\![P]\!]$

(LTS-HERE)
$\Gamma \triangleright k[\![\mathsf{here}\,[x]\; P]\!] \xrightarrow{\tau}_\beta \Gamma \triangleright k[\![P[^k\!/\!_x]]\!]$

(LTS-REC)
$\Gamma \triangleright k[\![\mathsf{rec}\; (Z:\mathsf{R}).\; P]\!] \xrightarrow{\tau}_\beta \Gamma \triangleright k[\![P\{^{\mathsf{rec}\,(Z:\mathsf{R}).\,P}\!/\!_Z\}]\!]$

(LTS-L-CREATE)
$\Gamma \triangleright k[\![(\mathsf{newloc}\, l:\mathsf{L})\; P]\!] \xrightarrow{\tau}_\beta \Gamma \triangleright (\mathsf{new}\, l:\mathsf{L})\; k[\![P]\!] \mid l[\![C]\!]$

(LTS-N-CREATE)
$\Gamma \triangleright k[\![(\mathsf{newreg}\, n:\mathsf{N})\; P]\!] \xrightarrow{\tau}_\beta \Gamma \triangleright (\mathsf{new}\, n:\mathsf{N})\; k[\![P]\!]$

(LTS-C.CREATE)
$\Gamma \triangleright k[\![(\mathsf{newc}\, c:\mathsf{C})\; P]\!] \xrightarrow{\tau}_\beta \Gamma \triangleright (\mathsf{new}\, c:\mathsf{C}@k)\; k[\![P]\!]$

(LTS-EQ)
$\Gamma \triangleright k[\![\mathsf{if}\; u = u\; \mathsf{then}\; P\; \mathsf{else}\; Q]\!] \xrightarrow{\tau}_\beta \Gamma \triangleright k[\![P]\!]$

(LTS-NEQ)
$\Gamma \triangleright k[\![\mathsf{if}\; u = v\; \mathsf{then}\; P\; \mathsf{else}\; Q]\!] \xrightarrow{\tau}_\beta \Gamma \triangleright k[\![Q]\!] \qquad \text{when } u \neq v$

(LTS-COMM)
$$\frac{\Gamma_M \triangleright M \xrightarrow{(\tilde{n}:\tilde{\mathsf{T}})k.a!V} \Gamma'_M \triangleright M' \qquad \Gamma_N \triangleright N \xrightarrow{(\tilde{n}:\tilde{\mathsf{U}})k.a?V} \Gamma'_N \triangleright N'}{\begin{array}{l}\Gamma \triangleright M \mid N \xrightarrow{\tau} \Gamma \triangleright (\mathsf{new}\,\widetilde{n}:\widetilde{\mathsf{T}})\; M' \mid N' \\ \Gamma \triangleright N \mid M \xrightarrow{\tau} \Gamma \triangleright (\mathsf{new}\,\widetilde{n}:\widetilde{\mathsf{T}})\; N' \mid M'\end{array}} \quad \tilde{n} \cap \mathsf{fn}(N) = \emptyset$$

Definition 4.3 [Actions] For configurations $\mathcal{C}$ of the form $(\Gamma \triangleright M)$, we say that they can do the following actions:

- $\mathcal{C} \xrightarrow{\tau} \mathcal{C}'$ or $\mathcal{C} \xrightarrow{(\tilde{n}:\tilde{\mathsf{T}})k.a?V} \mathcal{C}'$ if we can prove so with a derivation in the LTS;
- $\mathcal{C} \xrightarrow{(\tilde{n})k.a!V} \mathcal{C}'$ if there exists some derivation proving $\mathcal{C} \xrightarrow{(\tilde{m}:\tilde{\mathsf{T}}')k.a!V} \mathcal{C}'$ in the LTS with $(\tilde{n})$ the names that are both in V and $(\tilde{m})$. ■

Again we refer the reader to [3] for further motivation; this paper also contains the result that

$$(\Gamma \triangleright M) \approx_{bis} (\Gamma \triangleright N) \text{ implies } \Gamma \models M \cong_{rbc} N$$

whenever $\Gamma \vdash M$ and $\Gamma \vdash N$. So we establish Theorem 4.2 by showing

$$\Gamma \vdash M \text{ implies } (\Gamma \triangleright M) \approx_{bis} (\Gamma \triangleright \textsc{unrec}(M)) \tag{2}$$

Fig. 4 Labelled transition semantics. External actions.

(LTS-OUT)
$$\frac{\Gamma \vdash k : \text{LOC} \quad a : \text{R}\langle \mathsf{T}\rangle_{@}k \in \Gamma \quad \Gamma,\langle V : \mathsf{T}\rangle_{@}k \vdash \mathbf{env}}{\Gamma \triangleright k[\![a\,!\langle V\rangle\, P]\!] \xrightarrow{k.a!V} \Gamma,\langle V : \mathsf{T}\rangle_{@}k \triangleright k[\![P]\!]}$$

(LTS-IN)
$$\frac{\Gamma \vdash k : \text{LOC} \quad a : \text{W}\langle \mathsf{U}\rangle_{@}k \in \Gamma \quad \Gamma \vdash V : \mathsf{U}_{@}k}{\Gamma \triangleright k[\![a\,?(X : \mathsf{T})\, P]\!] \xrightarrow{k.a?V} \Gamma \triangleright k[\![P\{\!|^V/x|\!\}]\!]}$$

(LTS-NEW)
$$\frac{\Gamma \triangleright M \xrightarrow{\mu} \Gamma' \triangleright M'}{\Gamma \triangleright (\mathsf{new}\, n : \mathsf{T})\, M \xrightarrow{\mu} \Gamma' \triangleright (\mathsf{new}\, n : \mathsf{T})\, M'}\; n \notin \mu$$

(LTS-OPEN)
$$\frac{\Gamma \triangleright M \xrightarrow{(\tilde{n}:\tilde{\mathsf{T}})k.a!V} \Gamma' \triangleright M'}{\Gamma \triangleright (\mathsf{new}\, n : \mathsf{T})\, M \xrightarrow{(n\tilde{n}:\mathsf{T}\tilde{\mathsf{T}})k.a!V} \Gamma' \triangleright M'}\; \begin{array}{l} n \notin \{a, k\} \\ n \in \mathsf{fn}(V) \cup \mathsf{n}(\tilde{\mathsf{T}}) \end{array}$$

(LTS-WEAK)
$$\frac{\Gamma,\langle n : \mathsf{T}\rangle \triangleright M \xrightarrow{(\tilde{n}:\tilde{\mathsf{T}})k.a?V} \Gamma' \triangleright M'}{\Gamma \triangleright M \xrightarrow{(n:\mathsf{T},\tilde{n}:\tilde{\mathsf{T}})k.a?V} \Gamma' \triangleright M'}\; n \notin \{a, k\}$$

(LTS-PAR)
$$\frac{\Gamma \triangleright M \xrightarrow{\mu} \Gamma' \triangleright M'}{\Gamma \triangleright M \mid N \xrightarrow{\mu} \Gamma' \triangleright M' \mid N}\; \mathsf{bn}(\mu) \cap \mathsf{fn}(N) = \emptyset$$

5 Proof of recursion implementability

Let us see the problems encountered in trying to prove the equation (2) on an example. For this, let us consider a parameterised server version of our Search process that would be exploring a binary tree instead of a list:

$$\begin{aligned} \mathsf{PSearch} \triangleq\; & \mathit{search_req}\,?(x, \mathit{client}) \\ & \quad \mathsf{goto}\, k_0.\mathsf{rec}\, Z : \mathsf{S}.\ \mathit{test}\,?(y)\mathsf{if}\ p(x, y)\ \mathsf{then}\ \mathsf{goto}\,\mathit{client}.\mathit{report}\,!\langle y\rangle \\ & \qquad\qquad\qquad\qquad \mathsf{else}\ \mathit{neigh}\,?(n_1, n_2)\,\mathsf{goto}\, n_1.Z \mid \mathsf{goto}\, n_2.Z \end{aligned}$$

used in the system $\mathsf{Server}[\![* \mathsf{PSearch}]\!]$. So this sets up a search server, at Server; but the difference with Search from Example 2.1 is the fact that the data to search for in the network is given in the search request on *search_req*, and is subsequently used as a parameter by the testing predicate p.

Our translation of this process gives the following Dpi code:

$$\mathsf{IPSearch} \triangleq \mathit{search_req}?(x, \mathit{client}) \quad \mathsf{goto}\, k_0.\, (\mathsf{newreg}\, \mathit{ping})\ (\mathsf{newloc}\, \mathit{base})\, \mathsf{F} \mid \mathsf{goto}\, \mathit{base}.\, *\ \mathsf{Inst}$$

$$\mathsf{Inst} \triangleq \mathit{ping}?(k)\, \mathsf{goto}\, k.\mathit{test}?(y)\ \ \mathsf{if}\ p(x,y)\ \mathsf{then}\ \mathsf{goto}\, \mathit{client}.\mathit{report}!\langle y\rangle \ \mathsf{else}\ \mathit{neigh}?(n_1,n_2)\, \mathsf{goto}\, n_1.\mathsf{F} \mid \mathsf{goto}\, n_2.\mathsf{F}$$

$$\mathsf{F} \triangleq \mathsf{here}\,[l]\,\mathsf{goto}\, \mathit{base}.\mathit{ping}!\langle l\rangle$$

with Inst an instance of the iterative process, and F the triggering process, written FireOne is the example in the previous section.

Since IPSearch is replicated, it will generate a new home base for Inst for every request on *search_req*. This means that, after servicing a number of such requests we will end up with a system of the form:

$$(\mathsf{new}\, \mathit{ping}_1)\, (\mathsf{new}\, \mathit{base}_1)\, (\mathsf{new}\, \mathit{ping}_2)\, (\mathsf{new}\, \mathit{base}_2) \ldots \quad \mathsf{Server}[\![\ldots]\!] \mid \mathit{base}_1[\![\ldots]\!] \mid k_1^1[\![\ldots \mathsf{F}_1]\!] \mid k_1^2[\![\ldots \mathsf{F}_1]\!] \mid \ldots \mid \mathit{base}_2[\![\ldots]\!] \mid k_2^1[\![\ldots \mathsf{F}_2]\!] \ldots \tag{3}$$

Of course, this will correspond to the recDpi system:

$$\mathsf{Server}[\![\ldots]\!] \mid k_1^1[\![\ldots \mathsf{rec}\ Z.\ P]\!] \mid k_1^2[\![\ldots \mathsf{rec}\ Z.\ P]\!] \mid \ldots \mid k_2^1[\![\ldots \mathsf{rec}\ Z.\ P]\!] \ldots$$

On this example, we can see quite clearly the main difference at runtime between our translation and the standard, but non-compositional, one used in the pi-calculus we previously mentioned (see [8]), which arises because of the replication of rec $Z.\ P$. A translation following the lines of that in [8], would give rise to the following state, corresponding to (3) above:

$$(\mathsf{new}\, \mathit{ping})\, (\mathsf{new}\, \mathit{base})\, \mathsf{Server}[\![* \ \mathit{search_req}?(x, \mathit{client})\, \mathsf{goto}\, k_0.\mathsf{F}(x, \mathit{client})]\!] \mid \mathit{base}[\![* \ \mathit{ping}?(k, x, \mathit{client})\, \mathsf{goto}\, k.\mathit{test}?(y) \ldots]\!] \mid k_1^1[\![\ldots \mathsf{F}(x_1, \mathit{client}_1)]\!] \mid k_1^2[\![\ldots \mathsf{F}(x_1, \mathit{client}_1)]\!] \mid \ldots \mid k_2^1[\![\ldots \mathsf{F}(x_2, \mathit{client}_2)]\!] \ldots$$

$$\mathsf{F}(x, \mathit{client}) \triangleq \mathsf{here}\,[l]\,\mathsf{goto}\, \mathit{base}.\mathit{ping}!\langle l, x, \mathit{client}\rangle$$

Note that here all the free names used in the recursive process are closed and the actual parameters are obtained when an instance is called via *ping*. But more importantly only one home base is ever created. Thus the loss of compositionality allows an easier proof of equivalence, since there is only one *base* per recursion variable.

To return to the discussion of our translation, we have here a recDpi process containing a number of recursive constructs but the way they are to be translated to get the Dpi system (3) depends on the system history. That is why our proof of (2) is based on an extended version of the translation in which we specify whether a given occurrence of rec $Z.\ P$ has already been attributed a home base. If not, it should generate a new one; if it has, then the

actual home base needs to be recorded. In the example, we need to attribute the same home base to the $\mathsf{rec}\ Z.\ P$ in every k_1^i, and different ones for the other k_j^i.

Let us write $\textsc{unrec}_{\mathcal{P}}(M)$ for the translation of M parameterised by $\mathcal{P}$, with $\mathcal{P}$ specifying how each $\mathsf{rec}\ Z.\ P$ should be translated in M. Then to prove the equation (2), we would like to exhibit a bisimulation containing the pair

$$(\Gamma \triangleright M,\ \ \Gamma \triangleright \textsc{unrec}_{\mathcal{P}}(M))$$

for any term M, any environment Γ such that $\Gamma \triangleright M$ is well-formed configuration and any parameter $\mathcal{P}$.

But we still have a major problem in trying to exhibit this bisimulation: there are a lot of possible states in the translation corresponding to only one state in the original process, since we introduce numerous extra reductions. To deal with those extra steps, we will resort to a proof technique given in [6], namely bisimulation up-to-β. This is based on the remark that, among the reductions added by the translation, only the communication on the channel *ping* is "dangerous", because it could fail if one of the two agents involved in the communication were absent. Every other step is a so-called β-move, written $\xrightarrow{\tau}_{\beta}$ in the LTS, in Figure 3. Thanks to bisimulations up-to-β we can focus only on the communication moves, by using a new variant of the translation, written $\textsc{unrec}^{\beta}_{\mathcal{P}}(M)$, meaning that every β-move required for the bisimulation has been done. Then by considering that the *ping*-communication (which is a τ-move) in the translation corresponds to the recursion unwinding in recDpi, we can prove that the relation

$$(\Gamma \triangleright M, \Gamma \triangleright \textsc{unrec}^{\beta}_{\mathcal{P}}(M))$$

is a bisimulation-up-to-β.

6 Conclusion

In this paper we gave an extension of the Dpi-calculus with recursive processes. In particular we described why this construct was more suited to programming in the distributed setting, by allowing the description of agents migrating through network, visiting and interrogating different locations. We also gave a typing system for this extended calculus, which involved recursive types, dealt with by using co-inductive proof techniques, and showed that Subject Reduction remains valid. Finally we showed how to encode our recursive processes into standard Dpi which uses iteration, by resorting to the addition of extra migrations in the network. The encoding was proved to be sound and complete, in the sense that the original and translated processes are indistinguishable in a typed version of reduction barbed congruence.

It would now be interesting to study the behaviour of recursive processes in a setting where some parts of the network could fail (either locations or links),

since failures are of major importance in the study of distributed computations. We conjecture that in such a setting there is no translation of recursive processes into iterative ones, which preserve their behaviour.

References

[1] Francalanza, A. and M. Hennessy, *Location and link failure in a distributed pi-calculus*, Computer Science Report 2005:01, University of Sussex (2005).

[2] Gapeyev, V., M. Levin and B. Pierce, *Recursive subtyping revealed*, Journal of Functional Programming **12** (2003), pp. 511–548, preliminary version in *International Conference on Functional Programming (ICFP)*, 2000. Also appears as Chapter 21 of *Types and Programming Languages* by Benjamin C. Pierce (MIT Press, 2002).

[3] Hennessy, M., M. Merro and J. Rathke, *Towards a behavioural theory of access and mobility control in distributed systems*, Theoretical Computer Science **322** (2003), pp. 615–669.

[4] Hennessy, M. and J. Riely, *Resource access control in systems of mobile agents*, Information and Computation **173** (2002), pp. 82–120.

[5] Hym, S. and M. Hennessy, *Adding recursion to Dpi*, Computer science report, University of Sussex (2005).

[6] Jeffrey, A. and J. Rathke, *A theory of bisimulation for a fragment of concurrent ml with local names*, Theoretical Computer Science **323** (2004), pp. 1–48.

[7] Milner, R., "Communicating and Mobile Systems: the π-Calculus," Cambridge University Press, 1999.

[8] Sangiorgi, D. and D. Walker, "The π-calculus," Cambridge University Press, 2001.

A Dpi subtyping and typing rules

Fig. A.1 Subtyping rules

(SUB-BASE)

$$\frac{}{\textbf{base} <: \textbf{base}}$$

(SUB-CAP)

$$\frac{\mathsf{A} <: \mathsf{B}}{u : \mathsf{A} <: u : \mathsf{B}}$$

(SUB-TUPLE)

$$\frac{\mathsf{C}_i <: \mathsf{C}'_i}{(\widetilde{\mathsf{C}}) <: (\widetilde{\mathsf{C}'})}$$

(SUB-HOM)

$$\frac{\mathsf{A}_1 <: \mathsf{A}_2 \quad \mathsf{K}_1 <: \mathsf{K}_2}{\begin{array}{l}\mathsf{A}_1{@}\mathsf{K}_1 <: \mathsf{A}_2{@}\mathsf{K}_2 \\ \mathsf{A}_1{@}u <: \mathsf{A}_2{@}u \\ \mathrm{RC}\langle \mathsf{A}_1 \rangle <: \mathrm{RC}\langle \mathsf{A}_2 \rangle\end{array}}$$

(SUB-CHAN)

$$\frac{\mathsf{T}_2 <: \mathsf{T}_1 <: \mathsf{U}_1 <: \mathsf{U}_2}{\begin{array}{l}\mathrm{W}\langle \mathsf{T}_1 \rangle <: \mathrm{W}\langle \mathsf{T}_2 \rangle \\ \mathrm{R}\langle \mathsf{U}_1 \rangle <: \mathrm{R}\langle \mathsf{U}_2 \rangle \\ \mathrm{RW}\langle \mathsf{U}_1, \mathsf{T}_1 \rangle <: \mathrm{R}\langle \mathsf{U}_2 \rangle \\ \mathrm{RW}\langle \mathsf{U}_1, \mathsf{T}_1 \rangle <: \mathrm{W}\langle \mathsf{T}_2 \rangle \\ \mathrm{RW}\langle \mathsf{U}_1, \mathsf{T}_1 \rangle <: \mathrm{RW}\langle \mathsf{U}_2, \mathsf{T}_2 \rangle\end{array}}$$

(SUB-LOC)

$$\frac{\mathsf{A}_i <: \mathsf{A}'_i, \quad 1 \leq i \leq n}{\mathrm{LOC}[u_1 : \mathsf{A}_1, \ldots, u_n : \mathsf{A}_n, \ldots, u_{n+p} : \mathsf{A}_{n+p}] <: \mathrm{LOC}[u_1 : \mathsf{A}'_1, \ldots, u_n : \mathsf{A}'_n]}$$

Fig. A.2 Well-formed environments

(e-base)
$$\frac{\Gamma \vdash \mathbf{env}}{\Gamma, u : \mathbf{base} \vdash \mathbf{env}} \quad \Gamma(u) \downarrow \mathbf{base}$$

(e-empty)
$$\vdash \mathbf{env}$$

(e-new-lchan)
$$\frac{\Gamma \vdash \mathbf{env} \quad \Gamma \vdash w : \text{LOC} \quad \Gamma(u) = \{\mathsf{A}_i@v_i\}}{\Gamma, u : \mathsf{A}@w \vdash \mathbf{env}} \quad \begin{array}{l}\{v_i, w\} \text{ contains at most one location} \\ \{\mathsf{A}_i\} \downarrow \mathsf{A}\end{array}$$

(e-ref-lchan)
$$\frac{\Gamma \vdash \mathbf{env} \quad \Gamma \vdash w : \text{LOC} \quad \Gamma \vdash u : \text{RC}\langle \mathsf{B} \rangle, \quad \mathsf{B} <: \mathsf{A}}{\Gamma, u : \mathsf{A}@w \vdash \mathbf{env}}$$

(e-rchan)
$$\frac{\Gamma \vdash \mathbf{env}}{\Gamma, u : \text{RC}\langle \mathsf{A} \rangle \vdash \mathbf{env}} \quad u \notin \Gamma$$

(e-loc)
$$\frac{\Gamma \vdash \mathbf{env}}{\Gamma, v : \text{LOC} \vdash \mathbf{env}} \quad \Gamma(v) \downarrow \text{LOC}$$

(e-rec)
$$\frac{\Gamma \vdash \mathbf{env} \quad \Gamma(u_i) = \{\text{RC}\langle \mathsf{A}_{ij} \rangle, \mathsf{A}_{ik}@v_k\}}{\Gamma, Z : \text{LOC}[(u_i : \mathsf{A}_i)] \vdash \mathbf{env}} \quad \begin{array}{l} Z \notin \Gamma \\ \{\mathsf{A}_{ij}, \mathsf{A}_{ik}\} \downarrow \mathsf{A}_i \end{array}$$

(e-dec-at-rec)
$$\frac{\Gamma \vdash \mathbf{env} \quad \Gamma(Z) = \text{LOC}[\ldots, u : \mathsf{A}, \ldots]}{\Gamma, u : \mathsf{A}@Z \vdash \mathbf{env}}$$

Fig. A.3 Typing values

(v-meet)
$$\frac{\Gamma \vdash u : \mathsf{T}_1 \quad \Gamma \vdash u : \mathsf{T}_2}{\Gamma \vdash u : \mathsf{T}_1 \sqcap \mathsf{T}_2}$$

(v-name)
$$\frac{\Gamma, u : \mathsf{T}, \Gamma' \vdash \mathbf{env}}{\Gamma, u : \mathsf{T}, \Gamma' \vdash u : \mathsf{T}'} \quad \mathsf{T} <: \mathsf{T}'$$

(v-tuple)
$$\frac{\Gamma \vdash u_i : \mathsf{T}_i}{\Gamma \vdash (\widetilde{u}) : (\widetilde{\mathsf{T}})}$$

(v-located-channel)
$$\frac{\Gamma \vdash u_i : \mathsf{A}_i@v \quad \Gamma \vdash v : \mathsf{K}}{\Gamma \vdash (\widetilde{u})@v : (\widetilde{\mathsf{A}})@\mathsf{K}}$$

(v-loc)
$$\frac{\Gamma \vdash v : \text{LOC} \quad \Gamma \vdash u_i : \mathsf{A}_i@v}{\Gamma \vdash v : \text{LOC}[u_1 : \mathsf{A}_1, \ldots, u_n : \mathsf{A}_n]}$$

(v-dec-loc)
$$\frac{\Gamma \vdash v : \text{LOC} \quad \Gamma \vdash u_i : \mathsf{A}_i@v \quad \Gamma \vdash u_i : \text{RC}\langle \mathsf{D}_i \rangle, \mathsf{D}_i <: \mathsf{A}_i}{\Gamma \vdash_{\text{dec}} v : \text{LOC}[u_1 : \mathsf{A}_1, \ldots, u_n : \mathsf{A}_n]}$$

(v-located-tuple)
$$\frac{\Gamma \vdash u_i : \mathsf{A}_i@k}{\Gamma \vdash (\widetilde{u}) : (\widetilde{\mathsf{A}})@k}$$