Adding recursion to Dpi

Samuel Hym and Matthew Hennessy

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, [SW01], is a well-known formal calculus for describing, and reasoning about, the behaviour of concurrent processes which interact via two-way communication channels. Dpi, [HR02], 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, [HMR03, HR02]. The types informally correspond to sets of *capabilities*, and the use which a process may make of an entity, such as a location or a channel, depends on the current type at which process owns the entity. Moreover this type may change over time, reflecting the fact that processes may gradually accumulate capabilities over entities.

The most common formulations of the pi-calculus use iteration 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 [SW01].

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 5, and we give the proof of correctness in Section 6. 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 [HMR03, HR02] for detailed explanations of both the semantics of Dpi and its typing system.

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

FIGURE 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
$(\mathsf{newloc}\, k : \mathsf{K})\, P$	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**

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:

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

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 5 we give a typed labelled transition system for the language, the τ-moves of which provides our reduction semantics, see Figure 14. 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

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

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 $\mathit{ans}@x$.

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

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

Figure 2 Recursive pre-types

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

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

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

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, [HR02], which we can adapt to recDpi.

3.1 The Types

In this type system local channels have read/write types of the form $\mathrm{R}\langle\mathsf{U}\rangle$, $\mathrm{W}\langle\mathsf{T}\rangle$, or $\mathrm{RW}\langle\mathsf{U},\mathsf{T}\rangle$ (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

$$\mathrm{LOC}[u_1 : \mathsf{A}_1,\ldots,u_n : \mathsf{A}_n]$$

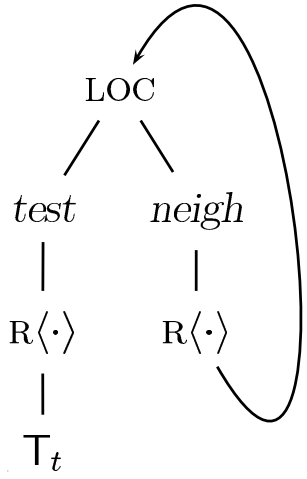
indicating that the local channels u_i may be used at the corresponding type A_i.

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

$$\mu\mathbf{Y}.\text{LOC}[\mathit{test} : \text{R}\langle\mathsf{T}_t\rangle,\ \ \mathit{neigh} : \text{R}\langle\mathbf{Y}\rangle]$$

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 [SW01] we can associate with each recursive pre-type T a co-inductive pre-type denoted Tree(T), which takes the form of a finite-branching, but possibly infinite, tree whose nodes are labelled by the type constructors. For example Tree(S) is the infinite tree



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

For every contractive S we can define Tree(S), the unique tree satisfying the following equations:

- *unwinding recursive pre-types* $\mathsf{Tree}(\mu\mathbf{Y}.S') = \mathsf{Tree}(S'\{\!|{}^{\mu\mathbf{Y}.S'}/_{\mathbf{Y}}|\!\})$

FIGURE 3 DPI subtyping rules

(SUB-BASE)

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

(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-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-HOM)

$$\frac{\mathsf{A}_1 <: \mathsf{A}_2 \quad \mathsf{K}_1 <: \mathsf{K}_2}{\mathsf{A}_1 @ \mathsf{K}_1 <: \mathsf{A}_2 @ \mathsf{K}_2}$$

(SUB-LOC)

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

- *not modifying any other construct; for instance*

$$\mathit{Tree}(\mathrm{R}\langle \mathsf{U}\rangle) = \mathsf{r}\langle \mathit{Tree}(\mathsf{U})\rangle$$

We call $\mathit{Tree}(\mathsf{S})$ *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 make clearer the distinction between tree types and recursive types, we slightly modify the notation and fonts of types. So the recursive type

$$\mu\mathbf{Y}.\mathrm{LOC}[\mathit{test} : \mathrm{R}\langle \mathsf{T}_t\rangle, \quad \mathit{neigh} : \mathrm{R}\langle \mathbf{Y}\rangle]$$

corresponds to the tree pre-type

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

To go from pre-types to types, we need to get rid of meaningless pre-types like $\mathsf{rw}\langle \mathsf{r}\langle\rangle, \mathbf{int}\rangle$, 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 $\mathsf{rw}\langle \mathbf{U}, \mathbf{T}\rangle$, $\mathbf{T}$ must be a subtype of $\mathbf{U}$.

In Figure 3 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

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

However here we interpret these rules co-inductively, [GLP03, Pie02]. 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:

$$\begin{aligned}
\mathsf{Sub}(\mathcal{R}) = {} & \{(\mathbf{base}, \mathbf{base})\} \\
& \cup \{(u : \mathbf{A}, u : \mathbf{B}) \text{ if } (\mathbf{A}, \mathbf{B}) \text{ is in } \mathcal{R}\} \\
& \cup \{((\widetilde{\mathbf{C}}), (\widetilde{\mathbf{C}'})) \text{ if } (\mathbf{C}_i, \mathbf{C}'_i) \text{ is in } \mathcal{R} \text{ for all } i\} \\
& \cup \{(\mathsf{w}\langle \mathbf{T}_1\rangle, \mathsf{w}\langle \mathbf{T}_2\rangle) \text{ if } (\mathbf{T}_2, \mathbf{T}_1) \text{ is in } \mathcal{R}\} \\
& \cup \{(\mathsf{r}\langle \mathbf{U}_1\rangle, \mathsf{r}\langle \mathbf{U}_2\rangle) \text{ if } (\mathbf{U}_1, \mathbf{U}_2) \text{ is in } \mathcal{R}\} \\
& \cup \{(\mathsf{rw}\langle \mathbf{U}_1, \mathbf{T}_1\rangle, \mathsf{r}\langle \mathbf{U}_2\rangle) \text{ if } (\mathbf{T}_1, \mathbf{U}_1) \text{ and } (\mathbf{U}_1, \mathbf{U}_2) \text{ are in } \mathcal{R}\} \\
& \cup \{(\mathsf{rw}\langle \mathbf{U}_1, \mathbf{T}_1\rangle, \mathsf{w}\langle \mathbf{T}_2\rangle) \text{ if } (\mathbf{T}_2, \mathbf{T}_1) \text{ and } (\mathbf{T}_1, \mathbf{U}_1) \text{ are in } \mathcal{R}\} \\
& \cup \{(\mathsf{rw}\langle \mathbf{U}_1, \mathbf{T}_1\rangle, \mathsf{rw}\langle \mathbf{U}_2, \mathbf{T}_2\rangle) \text{ if } (\mathbf{T}_2, \mathbf{T}_1), \\
& \qquad\qquad (\mathbf{T}_1, \mathbf{U}_1) \text{ and } (\mathbf{U}_1, \mathbf{U}_2) \text{ are in } \mathcal{R}\} \\
& \cup \{(\mathbf{A}_1 @ \mathbf{K}_1, \mathbf{A}_2 @ \mathbf{K}_2) \text{ if } (\mathbf{A}_1, \mathbf{A}_2) \text{ and } (\mathbf{K}_1, \mathbf{K}_2) \text{ are in } \mathcal{R}\} \\
& \cup \{(\mathsf{loc}[u_1 : \mathbf{A}_1, \ldots, u_{n+p} : \mathbf{A}_{n+p}], \mathsf{loc}[u_1 : \mathbf{A}'_1, \ldots, u_n : \mathbf{A}'_n]) \\
& \qquad \text{if } (\mathbf{A}_i, \mathbf{A}'_i) \text{ is in } \mathcal{R} \text{ for all } i \leq n\}
\end{aligned}$$

Note that Sub is a total monotonic function from relations to relations. We can easily see the intuition in the definition of this function: every case corresponds to one rule, even if a set of rules are grouped together, like in (SUB-CHAN) gathering all the different cases for the separate read and write capabilities. Then, if the hypotheses of any rule of Figure 3 are in $\mathcal{R}$ the conclusion is in $\mathsf{Sub}(\mathcal{R})$.

Now that the function Sub is defined, we can use it to define the notion of subtyping on the tree pre-types and consequently obtain the notion of types.

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* $\mathbf{T} <: \mathbf{T}'$ *to mean that* $(\mathbf{T}, \mathbf{T}')$ *is in* $\nu\mathsf{Sub}$*.*

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

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

3.2 Theory of tree types

Now that we have defined a notion of tree types out of recursive pre-types, we want to prove some properties of subtyping over these types. For this, 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 proof method works as follows. To show that some element, say a, is in the greatest fixpoint of any function f it is sufficient to give a set $\mathcal{S}$ such that

- a is in $\mathcal{S}$;
- $\mathcal{S}$ is a postfixpoint of f, that is $\mathcal{S} \subseteq f(\mathcal{S})$.

From this it follows that $\mathcal{S}$ is a subset of the greatest fixpoint of f, which therefore contains the element a.

We will apply this technique to the function Sub, and this case showing that a given $\mathcal{S}$ is a postfixpoint is facilitated by the fact that it is invertible, in the meaning of [GLP03, Pie02]. The *inverse* is the partial function defined as follows. Since there is always at most one conclusion in a rule, we can consider this partial function only on pairs:

$$
\begin{array}{l}
\mathsf{support}_{\mathsf{Sub}}((\mathbf{T}_1, \mathbf{T}_2)) = \\
\quad \emptyset \text{ if } \mathbf{T}_1 = \mathbf{T}_2 = \mathbf{base} \\
\quad \{(\mathbf{A}, \mathbf{B})\} \text{ if } \mathbf{T}_1 = u : \mathbf{A} \text{ and } \mathbf{T}_2 = u : \mathbf{B} \\
\quad \{(\mathbf{C}_i, \mathbf{C}'_i)\} \text{ if } \mathbf{T}_1 = (\widetilde{\mathbf{C}}) \text{ and } \mathbf{T}_2 = (\widetilde{\mathbf{C}'}), \text{ with same arity} \\
\quad \{(\mathbf{T}'_2, \mathbf{T}'_1)\} \text{ if } \mathbf{T}_1 = \mathsf{w}\langle \mathbf{T}'_1 \rangle \text{ and } \mathbf{T}_2 = \mathsf{w}\langle \mathbf{T}'_2 \rangle \\
\quad \{(\mathbf{U}'_1, \mathbf{U}'_2)\} \text{ if } \mathbf{T}_1 = \mathsf{r}\langle \mathbf{U}'_1 \rangle \text{ and } \mathbf{T}_2 = \mathsf{r}\langle \mathbf{U}'_2 \rangle \\
\quad \{(\mathbf{T}'_1, \mathbf{U}'_1), (\mathbf{U}'_1, \mathbf{U}'_2)\} \text{ if } \mathbf{T}_1 = \mathsf{rw}\langle \mathbf{U}'_1, \mathbf{T}'_1 \rangle \text{ and } \mathbf{T}_2 = \mathsf{r}\langle \mathbf{U}'_2 \rangle \\
\quad \{(\mathbf{T}'_2, \mathbf{T}'_1), (\mathbf{T}'_1, \mathbf{U}'_1)\} \text{ if } \mathbf{T}_1 = \mathsf{rw}\langle \mathbf{U}'_1, \mathbf{T}'_1 \rangle \text{ and } \mathbf{T}_2 = \mathsf{w}\langle \mathbf{T}'_2 \rangle \\
\quad \{(\mathbf{T}'_2, \mathbf{T}'_1), (\mathbf{T}'_1, \mathbf{U}'_1), (\mathbf{U}'_1, \mathbf{U}'_2)\} \\
\qquad \text{if } \mathbf{T}_1 = \mathsf{rw}\langle \mathbf{U}'_1, \mathbf{T}'_1 \rangle \text{ and } \mathbf{T}_2 = \mathsf{rw}\langle \mathbf{U}'_2, \mathbf{T}'_2 \rangle \\
\quad \{(\mathbf{A}_1, \mathbf{A}_2), (\mathbf{K}_1, \mathbf{K}_2)\} \text{ if } \mathbf{T}_i = \mathbf{A}_i \circledcirc \mathbf{K}_i \\
\quad \{(\mathbf{A}_{1i}, \mathbf{A}_{2i})_i\} \text{ for all } i \leq n \\
\qquad \text{if } \mathbf{T}_1 = \mathsf{loc}[u_1 : \mathbf{A}_{11}, \ldots, u_{n+p} : \mathbf{A}_{1(n+p)}] \\
\qquad \text{and } \mathbf{T}_2 = \mathsf{loc}[u_1 : \mathbf{A}_{21}, \ldots, u_n : \mathbf{A}_{2n}] \\
\quad \text{undefined otherwise}
\end{array}
$$

Then it is naturally extended into a partial function from relations to relations by:

$$\mathsf{support}_{\mathsf{Sub}}(\mathcal{R}) = \bigcup_{(\mathbf{T}_1, \mathbf{T}_2) \in \mathcal{R}} \mathsf{support}_{\mathsf{Sub}}((\mathbf{T}_1, \mathbf{T}_2))$$

$\mathsf{support}_{\mathsf{Sub}}(\mathcal{R})$ being undefined as soon as $\mathsf{support}_{\mathsf{Sub}}$ is undefined for some tuple in $\mathcal{R}$. Note that this definition implies that $\mathsf{support}_{\mathsf{Sub}}$, as a function from relations to relations, is monotonic on its definition domain.

Intuitively, $\mathsf{support}_{\mathsf{Sub}}$ computes the set of hypotheses needed to reach a given conclusion by Sub. In this sense it can be considered to be an inverse of Sub. Formally the relationship between these two functions is given in the following lemma.

Lemma 3.3. *(1) for a pair t if there exists $\mathcal{R}$ such that $t \in \mathsf{Sub}(\mathcal{R})$ then $\mathsf{support}_{\mathsf{Sub}}(t)$ is defined;*

(2) $\mathsf{support}_{\mathsf{Sub}}(\mathsf{Sub}(\mathcal{R})) \subseteq \mathcal{R}$ for any relation $\mathcal{R}$.

Proof. First we prove (1). Suppose $(\mathbf{T}_1, \mathbf{T}_2)$ is in $\mathsf{Sub}(\mathcal{R})$ for some $\mathcal{R}$. It must be by one of the cases in the definition of Sub, and each case corresponds exactly to one case of the definition of $\mathsf{support}_{\mathsf{Sub}}$.

Now consider (2). To prove that $\mathsf{support}_{\mathsf{Sub}}(\mathsf{Sub}(\mathcal{R})) \subseteq \mathcal{R}$ for any relation $\mathcal{R}$, it is enough again to do an analysis on a pair $(\mathbf{T}_1, \mathbf{T}_2)$ in $\mathsf{support}_{\mathsf{Sub}}(\mathsf{Sub}(\mathcal{R}))$. For instance, let us do one case: if $(\mathbf{T}_1, \mathbf{T}_2)$ is in $\mathsf{support}_{\mathsf{Sub}}(\mathsf{Sub}(\mathcal{R}))$ because there exists in $\mathsf{Sub}(\mathcal{R})$ a pair $(\mathbf{T}_2^0, \mathbf{T}_1^0)$ such that $\mathbf{T}_2^0 = \mathsf{rw}\langle \mathbf{U}_2, \mathbf{T}_2 \rangle$ and $\mathbf{T}_1^0 = \mathsf{w}\langle \mathbf{T}_1 \rangle$ (seventh case of the definition of $\mathsf{support}_{\mathsf{Sub}}$), then we know that $(\mathbf{T}_2^0, \mathbf{T}_1^0)$ can be in $\mathsf{Sub}(\mathcal{R})$ only by the seventh case of its definition which implies that $(\mathbf{T}_1, \mathbf{T}_2)$ and $(\mathbf{T}_2, \mathbf{U}_2)$ must be in $\mathcal{R}$ which concludes this case. □

Of course, since tree types are defined coinductively, equality must be defined as the greatest fixpoint of a function, the following total one:

$$
\begin{aligned}
\mathsf{Eq}(\mathcal{R}) = \{&(\mathbf{base}, \mathbf{base})\} \\
&\cup \{(u : \mathbf{A}, u : \mathbf{B}) \text{ if } (\mathbf{A}, \mathbf{B}) \text{ is in } \mathcal{R}\} \\
&\cup \{((\widetilde{\mathbf{C}}), (\widetilde{\mathbf{C}'})) \text{ if } (\mathbf{C}_i, \mathbf{C}'_i) \text{ is in } \mathcal{R} \text{ for all } i\} \\
&\cup \{(\mathsf{w}\langle \mathbf{T}_1 \rangle, \mathsf{w}\langle \mathbf{T}_2 \rangle) \text{ if } (\mathbf{T}_1, \mathbf{T}_2) \text{ is in } \mathcal{R}\} \\
&\cup \{(\mathsf{r}\langle \mathbf{U}_1 \rangle, \mathsf{r}\langle \mathbf{U}_2 \rangle) \text{ if } (\mathbf{U}_1, \mathbf{U}_2) \text{ is in } \mathcal{R}\} \\
&\cup \{(\mathsf{rw}\langle \mathbf{U}_1, \mathbf{T}_1 \rangle, \mathsf{rw}\langle \mathbf{U}_2, \mathbf{T}_2 \rangle) \text{ if } (\mathbf{T}_1, \mathbf{T}_2) \text{ and } (\mathbf{U}_1, \mathbf{U}_2) \text{ are in } \mathcal{R}\} \\
&\cup \{(\mathbf{A}_1 @ \mathbf{K}_1, \mathbf{A}_2 @ \mathbf{K}_2) \text{ if } (\mathbf{A}_1, \mathbf{A}_2) \text{ and } (\mathbf{K}_1, \mathbf{K}_2) \text{ are in } \mathcal{R}\} \\
&\cup \{(\mathsf{loc}[u_1 : \mathbf{A}_1, \ldots, u_n : \mathbf{A}_n], \mathsf{loc}[u_1 : \mathbf{A}'_1, \ldots, u_n : \mathbf{A}'_n]) \\
&\qquad \text{if } (\mathbf{A}_i, \mathbf{A}'_i) \text{ is in } \mathcal{R} \text{ for all } i \leq n\}
\end{aligned}
$$

So the notion of equality given by the greatest fixpoint of this function uses the main "handle" we have on tree types: it is intuitively checking that the "heads" of the terms are identical and that the "tails" are also equal. From now on, we will write $\mathbf{T}_1 = \mathbf{T}_2$ when $(\mathbf{T}_1, \mathbf{T}_2)$ is in $\nu\mathsf{Eq}$.

With this notion of equality, we show now how Lemma 3.3 can be used to prove a simple and fundamental property, namely the fact that $\nu\mathsf{Sub}$ is a partial order, so that we have reflexivity, antisymmetry and transitivity.

Lemma 3.4 (Reflexivity). *For any tree type* $\mathbf{T}$, $\mathbf{T} <: \mathbf{T}$.

Proof. Let us consider the relation

$$\mathcal{R} = \{(\mathbf{T}, \mathbf{T}) \mid \mathbf{T} \text{ is a type}\} \cup \nu\mathsf{Sub}$$

We prove that $\mathcal{R}$ is a postfixpoint of Sub.

Let us take a pair in $\mathcal{R}$. If this pair is of the form $(\mathbf{T}, \mathbf{T})$, we reason on the form of $\mathbf{T}$.

- $\mathbf{base}$ then $(\mathbf{T}, \mathbf{T})$ is obviously in $\mathsf{Sub}(\mathcal{R})$.
- $\mathsf{r}\langle \mathbf{U}_0 \rangle$ then, since $(\mathbf{U}_0, \mathbf{U}_0)$ is in $\mathcal{R}$, $(\mathbf{T}, \mathbf{T})$ is in $\mathsf{Sub}(\mathcal{R})$.
- $\mathsf{rw}\langle \mathbf{U}_0, \mathbf{T}_0 \rangle$ then, by well-formedness of $\mathbf{T}$, we know that $\mathbf{T}_0 <: \mathbf{U}_0$ so $(\mathbf{T}_0, \mathbf{U}_0)$ is in $\mathcal{R}$. Of course, so are $(\mathbf{T}_0, \mathbf{T}_0)$ and $(\mathbf{U}_0, \mathbf{U}_0)$, which implies that $(\mathbf{T}, \mathbf{T})$ is in $\mathsf{Sub}(\mathcal{R})$.
- The remaining cases are similar.

If the pair is in $\nu\mathsf{Sub}$, we know that it is also in $\mathsf{Sub}(\nu\mathsf{Sub})$ which is included in $\mathsf{Sub}(\mathcal{R})$. □

Lemma 3.5 (Antisymmetry). *Suppose that for some tree types* $\mathbf{T}_1$ *and* $\mathbf{T}_2$, $\mathbf{T}_1 <: \mathbf{T}_2$ *and* $\mathbf{T}_2 <: \mathbf{T}_1$. *Then* $\mathbf{T}_1 = \mathbf{T}_2$.

Proof. Consider the relation $\mathcal{R}$ over types defined by:

$$\mathcal{R} = \{(\mathbf{T}_1, \mathbf{T}_2) \mid \mathbf{T}_1 <: \mathbf{T}_2, \mathbf{T}_2 <: \mathbf{T}_1\}$$

We show that this is a postfixpoint of Eq.

For this let us consider two types $\mathbf{T}_1$ and $\mathbf{T}_2$ such that $(\mathbf{T}_1, \mathbf{T}_2)$ is in $\mathcal{R}$. Then we reason by cases on $\mathbf{T}_1 <: \mathbf{T}_2$. We give here only typical examples:

- $\mathbf{T}_1 = \mathbf{T}_2 = \mathbf{base}$ then $(\mathbf{T}_1, \mathbf{T}_2)$ is obviously in $\mathsf{Eq}(\mathcal{R})$;
- $\mathbf{T}_1 = u : \mathbf{A}_1$ and $\mathbf{T}_2 = u : \mathbf{A}_2$ with $\mathbf{A}_1 <: \mathbf{A}_2$; then $\mathbf{T}_2 <: \mathbf{T}_1$ implies also that $\mathbf{A}_2 <: \mathbf{A}_1$ which means that $(\mathbf{A}_1, \mathbf{A}_2)$ is also in $\mathcal{R}$; this entails that $(\mathbf{T}_1, \mathbf{T}_2)$ is in $\mathsf{Eq}(\{(\mathbf{A}_1, \mathbf{A}_2)\}) \subseteq \mathsf{Eq}(\mathcal{R})$ by monotonicity of Eq;
- $\mathbf{T}_1 = \mathsf{rw}\langle \mathbf{U}'_1, \mathbf{T}'_1 \rangle$ and $\mathbf{T}_2 = \mathsf{r}\langle \mathbf{U}'_2 \rangle$ is impossible because $\mathbf{T}_2 \not<: \mathbf{T}_1$.
- The remaining cases are similar.

This proves that $\mathcal{R}$ is included in $\mathsf{Eq}(\mathcal{R})$, from which the result follows. □

Lemma 3.6 (Transitivity). *Let us suppose that for some tree types* $\mathbf{T}_1$, $\mathbf{T}_2$ *and* $\mathbf{T}_3$, $\mathbf{T}_1 <: \mathbf{T}_2$ *and* $\mathbf{T}_2 <: \mathbf{T}_3$. *Then* $\mathbf{T}_1 <: \mathbf{T}_3$.

Proof. Let us write Tr for the function $\mathsf{Tr}(\mathcal{R}) = \mathcal{R} \cup \mathcal{R} \circ \mathcal{R}$. Then, what we want to prove can be formulated as

$$\mathsf{Tr}(\nu\mathsf{Sub}) \subseteq \nu\mathsf{Sub}$$

for which we can use the coinduction proof principle. It is sufficient to prove that

$$\mathsf{Tr}(\nu\mathsf{Sub}) \subseteq \mathsf{Sub}(\mathsf{Tr}(\nu\mathsf{Sub})) \tag{1}$$

i.e. that $\mathsf{Tr}(\nu\mathsf{Sub})$ is a postfixpoint of Sub.

For this, let us consider a pair $(\mathbf{T}_1, \mathbf{T}_3)$ in $\mathsf{Tr}(\nu\mathsf{Sub})$. By definition of Tr this implies that either $(\mathbf{T}_1, \mathbf{T}_3)$ is in $\nu\mathsf{Sub}$, in which case it is easy to establish that it is also in $\mathsf{Sub}(\mathsf{Tr}(\nu\mathsf{Sub}))$ because $\nu\mathsf{Sub} \subseteq \mathsf{Tr}(\nu\mathsf{Sub})$ implies that $\nu\mathsf{Sub} = \mathsf{Sub}(\nu\mathsf{Sub}) \subseteq \mathsf{Sub}(\mathsf{Tr}(\nu\mathsf{Sub}))$, or else there exists some type $\mathbf{T}_2$ such that $(\mathbf{T}_1, \mathbf{T}_2)$ and $(\mathbf{T}_2, \mathbf{T}_3)$ are in $\nu\mathsf{Sub}$. Therefore $\mathsf{support}_{\mathsf{Sub}}((\mathbf{T}_1, \mathbf{T}_2))$ and $\mathsf{support}_{\mathsf{Sub}}((\mathbf{T}_2, \mathbf{T}_3))$ are defined.

Now we prove that

$$(\mathbf{T}_1, \mathbf{T}_3) \in \mathsf{Sub}(\mathsf{Tr}(\mathsf{support}_{\mathsf{Sub}}((\mathbf{T}_1, \mathbf{T}_2)) \cup \mathsf{support}_{\mathsf{Sub}}((\mathbf{T}_2, \mathbf{T}_3)))) \tag{2}$$

by case analysis on the fact that $(\mathbf{T}_1, \mathbf{T}_2)$ is in $\mathsf{Sub}(\nu\mathsf{Sub})$. In all there are ten possibilities, of which we examine two typical ones.

- $\mathbf{T}_i = \mathsf{rw}\langle \mathbf{U}'_i, \mathbf{T}'_i \rangle$ with $\mathbf{T}'_2 <: \mathbf{T}'_1$, $\mathbf{T}'_1 <: \mathbf{U}'_1$ and $\mathbf{U}'_1 <: \mathbf{U}'_2$. Then $\mathbf{T}_3$ can be any one of the forms $\mathsf{r}\langle \mathbf{U}'_3 \rangle$, $\mathsf{w}\langle \mathbf{T}'_3 \rangle$, or $\mathsf{rw}\langle \mathbf{U}'_3, \mathbf{T}'_3 \rangle$, with the relevant constraints among $\mathbf{T}'_3 <: \mathbf{T}'_2$, $\mathbf{T}'_2 <: \mathbf{U}'_2$ and $\mathbf{U}'_2 <: \mathbf{U}'_3$. In the case where $\mathbf{T}_3 = \mathsf{rw}\langle \mathbf{U}'_3, \mathbf{T}'_3 \rangle$, this implies:

  $$\begin{aligned}&\mathsf{support}_{\mathsf{Sub}}((\mathbf{T}_1, \mathbf{T}_2)) \cup \mathsf{support}_{\mathsf{Sub}}((\mathbf{T}_2, \mathbf{T}_3)) = \\ &\quad \{(\mathbf{T}'_2, \mathbf{T}'_1), (\mathbf{T}'_1, \mathbf{U}'_1), (\mathbf{U}'_1, \mathbf{U}'_2), (\mathbf{T}'_3, \mathbf{T}'_2), (\mathbf{T}'_2, \mathbf{U}'_2), (\mathbf{U}'_2, \mathbf{U}'_3)\}\end{aligned}$$

  so $\mathsf{Tr}(\ldots)$ contains $(\mathbf{T}'_3, \mathbf{T}'_1)$, $(\mathbf{T}'_1, \mathbf{U}'_1)$ and $(\mathbf{U}'_1, \mathbf{U}'_3)$, the three pairs we need so that their Sub contains $(\mathbf{T}_1, \mathbf{T}_3) = (\mathsf{rw}\langle \mathbf{U}'_1, \mathbf{T}'_1 \rangle, \mathsf{rw}\langle \mathbf{U}'_3, \mathbf{T}'_3 \rangle)$.

  On the other hand if $\mathbf{T}_3$ is only $\mathsf{r}\langle \mathbf{U}'_3 \rangle$ or $\mathsf{w}\langle \mathbf{T}'_3 \rangle$, one pair disappears from $\mathsf{support}_{\mathsf{Sub}}(\ldots)$ and $\mathsf{Tr}(\ldots)$. But that pair is not needed to establish that the application of Sub to contains $(\mathbf{T}_1, \mathbf{T}_3)$.

- $\mathbf{T}_1 = \mathsf{loc}[u_1 : \mathbf{A}_1, \ldots, u_{n+p+q} : \mathbf{A}_{n+p+q}]$ and $\mathbf{T}_2 = \mathsf{loc}[u_1 : \mathbf{A}'_1, \ldots, u_{n+p} : \mathbf{A}'_{n+p}]$ with $\mathbf{A}_i <: \mathbf{A}'_i$ for all $i \leq n + p$. In this case $\mathbf{T}_2 <: \mathbf{T}_3$ implies that $\mathbf{T}_3$ is of the form $\mathsf{loc}[u_1 : \mathbf{A}''_1, \ldots, u_n : \mathbf{A}''_n]$ and that $\mathbf{A}'_j <: \mathbf{A}''_j$ for all $j \leq n$. Then $\mathsf{support}_{\mathsf{Sub}}((\mathbf{T}_1, \mathbf{T}_2)) \cup \mathsf{support}_{\mathsf{Sub}}((\mathbf{T}_2, \mathbf{T}_3))$ contains $(\mathbf{A}_i, \mathbf{A}'_i)$ for all $i \leq n + p$ and $(\mathbf{A}'_j, \mathbf{A}''_j)$ for all $j \leq n$ so Tr

of this set contains exactly the pairs we need, namely $(\mathbf{A}_j, \mathbf{A}''_j)$ for all $j \leq n$, to get $(\mathbf{T}_1, \mathbf{T}_3)$ via an application of Sub.

So we have established (2) above.

We now reason as follows. Since $\{(\mathbf{T}_1, \mathbf{T}_2), (\mathbf{T}_2, \mathbf{T}_3)\} \subseteq \nu\mathsf{Sub}$, and the function $\mathsf{support}_{\mathsf{Sub}}$ is monotonic, we know

$$\begin{aligned}\mathsf{support}_{\mathsf{Sub}}(\{(\mathbf{T}_1, \mathbf{T}_2), (\mathbf{T}_2, \mathbf{T}_3)\}) &\subseteq \mathsf{support}_{\mathsf{Sub}}(\nu\mathsf{Sub}) \\ &= \mathsf{support}_{\mathsf{Sub}}(\mathsf{Sub}(\nu\mathsf{Sub}))\end{aligned}$$

But by the second part of Lemma 3.3 we know that $\mathsf{support}_{\mathsf{Sub}}(\mathsf{Sub}(\nu\mathsf{Sub}))$ is a subset of $\nu\mathsf{Sub}$ and so

$$\mathsf{support}_{\mathsf{Sub}}(\{(\mathbf{T}_1, \mathbf{T}_2), (\mathbf{T}_2, \mathbf{T}_3)\}) \subseteq \nu\mathsf{Sub}$$

Therefore (2), together with the monotonicity of Sub and Tr ensures that

$$(\mathbf{T}_1, \mathbf{T}_3) \in \mathsf{Sub}(\mathsf{Tr}(\nu\mathsf{Sub}))$$

from which the required (1) follows. □

We can also define a *meet* relation on our types. We proceed as with the subtyping relation. Meets and joins are defined inductively in DPI using the rules in Figures 4 and 5. Those rules involve statements of the form $\mathsf{T}_1 \sqcap \mathsf{T}_2 = \mathsf{T}_3$ and $\mathsf{T}_1 \sqcup \mathsf{T}_2 = \mathsf{T}_3$. To adapt them to our coinductive setting, we define triples of the form ${}_{\sqcap}(\mathbf{T}_1, \mathbf{T}_2, \mathbf{T}_3)$ and ${}_{\sqcup}(\mathbf{T}_1, \mathbf{T}_2, \mathbf{T}_3)$, where the $\mathbf{T}_i$ are tree types. Then we use the rules from Figures 4 and 5 to define a coinductive total function $\mathsf{MeetJoin}$ over relations of such triples. We give in Figure 6 the definition of the function $\mathsf{MeetJoin}$. All the individual clauses in this definition are inherited from the rules in these figures. The meet and the join operators must be defined at the same time since we have to deal with contravariance in our types.

We will write $\mathbf{T}_1 \sqcap \mathbf{T}_2 = \mathbf{T}_3$ for types $\mathbf{T}_1$, $\mathbf{T}_2$ and $\mathbf{T}_3$ such that ${}_{\sqcap}(\mathbf{T}_1, \mathbf{T}_2, \mathbf{T}_3)$ is in $\nu\mathsf{MeetJoin}$.

We still need to show that the greatest fixpoint of this function actually gives us operators with the same properties as those in the inductive types of DPI. For instance, since meet is defined as a relation over triples of tree types, we need to first prove that this is a partial function.

Lemma 3.7 (Meet is a function). *For any types* $\mathbf{T}_1$, $\mathbf{T}_2$, $\mathbf{T}_3$ *and* $\mathbf{T}_4$, $\mathbf{T}_1 \sqcap \mathbf{T}_2 = \mathbf{T}_3$ *and* $\mathbf{T}_1 \sqcap \mathbf{T}_2 = \mathbf{T}_4$ *implies* $\mathbf{T}_3 = \mathbf{T}_4$.

FIGURE 4 DPI meet rules

(MEET-BASE)

$$\frac{}{\mathbf{base}_1 \sqcap \mathbf{base}_2 = \mathbf{base}_3}\ \mathbf{base}_1 = \mathbf{base}_2 = \mathbf{base}_3$$

(MEET-CAP)

$$\frac{\mathsf{A}_1 \sqcap \mathsf{A}_2 = \mathsf{A}_3}{u : \mathsf{A}_1 \sqcap u : \mathsf{A}_2 = u : \mathsf{A}_3}$$

(MEET-TUPLE)

$$\frac{\mathsf{C}_i \sqcap \mathsf{C}'_i = \mathsf{C}''_i}{(\widetilde{\mathsf{C}}) \sqcap (\widetilde{\mathsf{C}'}) = (\widetilde{\mathsf{C}''})}$$

(MEET-CHAN)

$$\frac{\mathsf{U}_1 \sqcap \mathsf{U}_2 = \mathsf{U}_3 \quad \mathsf{T}_1 \sqcup \mathsf{T}_2 = \mathsf{T}_3}{\mathrm{RW}\langle \mathsf{U}_1, \mathsf{T}_1\rangle \sqcap \mathrm{RW}\langle \mathsf{U}_2, \mathsf{T}_2\rangle = \mathrm{RW}\langle \mathsf{U}_3, \mathsf{T}_3\rangle}\ \mathsf{T}_3 <: \mathsf{U}_3$$

(MEET-HOM)

$$\frac{\mathsf{A}_1 \sqcap \mathsf{A}_2 = \mathsf{A}_3 \quad \mathsf{K}_1 \sqcap \mathsf{K}_2 = \mathsf{K}_3}{\mathsf{A}_1 @ \mathsf{K}_1 \sqcap \mathsf{A}_2 @ \mathsf{K}_2 = \mathsf{A}_3 @ \mathsf{K}_3}$$

(MEET-LOC)

$$\frac{\mathsf{U}_i \sqcap \mathsf{U}'_i = \mathsf{U}''_i}{\begin{array}{l}\mathrm{LOC}[(u_i : \mathsf{U}_i)_i; (v_j : \mathsf{V}_j)_j] \sqcap \mathrm{LOC}[(u_i : \mathsf{U}'_i)_i; (w_k : \mathsf{W}_k)_k] \\ \quad = \mathrm{LOC}[(u_i : \mathsf{U}''_i)_i; (v_j : \mathsf{V}_j)_j; (w_k : \mathsf{W}_k)_k]\end{array}}$$

Proof. Of course, we have to prove this result for both meet and join. Let us consider the relation $\mathcal{R}$ over types defined by:

$$\begin{array}{l}\mathcal{R} = \{(\mathbf{T}_3, \mathbf{T}_4) \mid \\ \quad \exists \mathbf{T}_1, \mathbf{T}_2 \text{ such that } \mathbf{T}_1 \sqcap \mathbf{T}_2 = \mathbf{T}_3, \mathbf{T}_1 \sqcap \mathbf{T}_2 = \mathbf{T}_4 \\ \qquad \text{or such that } \mathbf{T}_1 \sqcup \mathbf{T}_2 = \mathbf{T}_3, \mathbf{T}_1 \sqcup \mathbf{T}_2 = \mathbf{T}_4\}\end{array}$$

and we show that this is a postfixpoint of Eq.

Let us consider $\mathbf{T}_3$ and $\mathbf{T}_4$ such that $\mathbf{T}_3 \mathcal{R} \mathbf{T}_4$, and we write $\mathbf{T}_1$ and $\mathbf{T}_2$ for the corresponding two types. We reason on the possible cases for $\mathbf{T}_1 \sqcap \mathbf{T}_2 = \mathbf{T}_3$ or $\mathbf{T}_1 \sqcup \mathbf{T}_2 = \mathbf{T}_3$. We see here some typical examples.

- $\mathbf{base} \sqcap \mathbf{base} = \mathbf{base}$; then the only possible $\mathbf{T}_4$ is $\mathbf{base}$, so $(\mathbf{T}_3, \mathbf{T}_4)$ is obviously in $\mathsf{Eq}(\mathcal{R})$.
- $u : \mathbf{A}_1 \sqcap u : \mathbf{A}_2 = u : \mathbf{A}_3$ with $\mathbf{A}_1 \sqcap \mathbf{A}_2 = \mathbf{A}_3$; then the only possible $\mathbf{T}_4$ is of the form $u : \mathbf{A}_4$ with $\mathbf{A}_1 \sqcap \mathbf{A}_2 = \mathbf{A}_4$, so $(\mathbf{A}_3, \mathbf{A}_4)$ also is in $\mathcal{R}$. So

$$(\mathbf{T}_3, \mathbf{T}_4) \in \mathsf{Eq}(\{(\mathbf{A}_3, \mathbf{A}_4)\}) \subseteq \mathsf{Eq}(\mathcal{R}) \ .$$

FIGURE 5 DPI join rules

(JOIN-BASE)

$$\frac{}{\mathbf{base}_1 \sqcup \mathbf{base}_2 = \mathbf{base}_3} \; \mathbf{base}_1 = \mathbf{base}_2 = \mathbf{base}_3$$

(JOIN-CAP)

$$\frac{\mathsf{A}_1 \sqcup \mathsf{A}_2 = \mathsf{A}_3}{u : \mathsf{A}_1 \sqcup u : \mathsf{A}_2 = u : \mathsf{A}_3}$$

(JOIN-TUPLE)

$$\frac{\mathsf{C}_i \sqcup \mathsf{C}'_i = \mathsf{C}''_i}{(\widetilde{\mathsf{C}}) \sqcup (\widetilde{\mathsf{C}'}) = (\widetilde{\mathsf{C}''})}$$

(JOIN-CHAN)

$$\frac{\mathsf{U}_1 \sqcup \mathsf{U}_2 = \mathsf{U}_3 \quad \mathsf{T}_1 \sqcap \mathsf{T}_2 = \mathsf{T}_3}{\text{RW}\langle \mathsf{U}_1, \mathsf{T}_1 \rangle \sqcup \text{RW}\langle \mathsf{U}_2, \mathsf{T}_2 \rangle = \text{RW}\langle \mathsf{U}_3, \mathsf{T}_3 \rangle} \; \begin{array}{l} \mathsf{T}_1 <: \mathsf{U}_1 \\ \mathsf{T}_2 <: \mathsf{U}_2 \end{array}$$

(JOIN-HOM)

$$\frac{\mathsf{A}_1 \sqcup \mathsf{A}_2 = \mathsf{A}_3 \quad \mathsf{K}_1 \sqcup \mathsf{K}_2 = \mathsf{K}_3}{\mathsf{A}_1 @ \mathsf{K}_1 \sqcup \mathsf{A}_2 @ \mathsf{K}_2 = \mathsf{A}_3 @ \mathsf{K}_3}$$

(JOIN-LOC)

$$\frac{\mathsf{U}_i \sqcup \mathsf{U}'_i = \mathsf{U}''_i}{\text{LOC}[(u_i : \mathsf{U}_i)_i; (v_j : \mathsf{V}_j)_j] \sqcup \text{LOC}[(u_i : \mathsf{U}'_i)_i; (w_k : \mathsf{W}_k)_k] = \text{LOC}[(u_i : \mathsf{U}''_i)_i]}$$

- $\mathsf{loc}[u_1 : \mathbf{A}_1, \ldots, u_n : \mathbf{A}_n, v_1 : \mathbf{B}_1, \ldots] \sqcup \mathsf{loc}[u_1 : \mathbf{A}'_1, \ldots, u_n : \mathbf{A}'_n, w_1 : \mathbf{B}'_1, \ldots] = \mathsf{loc}[u_{i_1} : \mathbf{A}''_{i_1}, \ldots, u_{i_p} : \mathbf{A}''_{i_p}]$ then we know that $\mathbf{T}_4$ must be of the form $\mathsf{loc}[u_{i_1} : \mathbf{A}'''_{i_1}, \ldots, u_{i_p} : \mathbf{A}'''_{i_p}]$ because the set of indices $\{i_j\}$ is determined by the compatibility of the types $\mathbf{A}_i$ and $\mathbf{A}'_i$; this means that $(\mathbf{A}''_{i_j}, \mathbf{A}'''_{i_j})$ are in $\mathcal{R}$ for every i_j, so

$$(\mathbf{T}_3, \mathbf{T}_4) \in \mathsf{Eq}(\{(\mathbf{A}''_{i_j}, \mathbf{A}'''_{i_j})\}) \subseteq \mathsf{Eq}(\mathcal{R}) \; .$$

So we have then proved that the relation $\nu\mathsf{MeetJoin}$ defines a function from couples of types to types. □

Of course, we want to prove that the meet operator we defined is indeed a meet. This means that we want to prove that for any type $\mathbf{T}$, $\mathbf{T} \sqcap \mathbf{T}$ is indeed $\mathbf{T}$, that the meet of two types is a subtype of each of them, and that any common subtype is also a subtype of the meet. To write these proofs more conveniently, we start by proving that $\sqcap$ is symmetric over its two arguments.

FIGURE 6 MeetJoin definition

$$
\begin{aligned}
&\mathsf{MeetJoin}(\mathcal{R}) = \\
&\quad \{\sqcap(\mathbf{base}, \mathbf{base}, \mathbf{base})\} \\
&\cup \{\sqcap(u:\mathbf{A}_1, u:\mathbf{A}_2, u:\mathbf{A}_3) \text{ if } \sqcap(\mathbf{A}_1,\mathbf{A}_2,\mathbf{A}_3) \text{ is in } \mathcal{R}\} \\
&\cup \{\sqcap((\widetilde{\mathbf{C}}),(\widetilde{\mathbf{C}'}),(\widetilde{\mathbf{C}''})) \text{ if } \sqcap(\mathbf{C}_i,\mathbf{C}'_i,\mathbf{C}''_i) \text{ is in } \mathcal{R} \text{ for all } i\} \\
&\cup \{\sqcap(\mathsf{r}\langle\mathbf{U}_1\rangle,\mathsf{r}\langle\mathbf{U}_2\rangle,\mathsf{r}\langle\mathbf{U}_3\rangle) \text{ if } \sqcap(\mathbf{U}_1,\mathbf{U}_2,\mathbf{U}_3) \text{ is in } \mathcal{R}\} \\
&\cup \{\sqcap(\mathsf{w}\langle\mathbf{T}_1\rangle,\mathsf{w}\langle\mathbf{T}_2\rangle,\mathsf{w}\langle\mathbf{T}_3\rangle) \text{ if } \sqcup(\mathbf{T}_1,\mathbf{T}_2,\mathbf{T}_3) \text{ is in } \mathcal{R}\} \\
&\cup \{\sqcap(\mathsf{r}\langle\mathbf{U}_1\rangle,\mathsf{w}\langle\mathbf{T}_2\rangle,\mathsf{rw}\langle\mathbf{U}_1,\mathbf{T}_2\rangle) \text{ if } \mathbf{T}_2 <: \mathbf{U}_1\} \\
&\cup \{\sqcap(\mathsf{w}\langle\mathbf{T}_1\rangle,\mathsf{r}\langle\mathbf{U}_2\rangle,\mathsf{rw}\langle\mathbf{U}_2,\mathbf{T}_1\rangle) \text{ if } \mathbf{T}_1 <: \mathbf{U}_2\} \\
&\cup \{\sqcap(\mathsf{rw}\langle\mathbf{U}_1,\mathbf{T}_1\rangle,\mathsf{r}\langle\mathbf{U}_2\rangle,\mathsf{rw}\langle\mathbf{U}_3,\mathbf{T}_1\rangle) \text{ if } \sqcap(\mathbf{U}_1,\mathbf{U}_2,\mathbf{U}_3) \text{ is in } \mathcal{R} \text{ and } \mathbf{T}_1 <: \mathbf{U}_3\} \\
&\cup \{\sqcap(\mathsf{rw}\langle\mathbf{U}_1,\mathbf{T}_1\rangle,\mathsf{w}\langle\mathbf{T}_2\rangle,\mathsf{rw}\langle\mathbf{U}_1,\mathbf{T}_3\rangle) \text{ if } \sqcup(\mathbf{T}_1,\mathbf{T}_2,\mathbf{T}_3) \text{ is in } \mathcal{R} \text{ and } \mathbf{T}_3 <: \mathbf{U}_1\} \\
&\cup \{\sqcap(\mathsf{r}\langle\mathbf{U}_1\rangle,\mathsf{rw}\langle\mathbf{U}_2,\mathbf{T}_2\rangle,\mathsf{rw}\langle\mathbf{U}_3,\mathbf{T}_2\rangle) \text{ if } \sqcap(\mathbf{U}_1,\mathbf{U}_2,\mathbf{U}_3) \text{ is in } \mathcal{R} \text{ and } \mathbf{T}_2 <: \mathbf{U}_3\} \\
&\cup \{\sqcap(\mathsf{w}\langle\mathbf{T}_1\rangle,\mathsf{rw}\langle\mathbf{U}_2,\mathbf{T}_2\rangle,\mathsf{rw}\langle\mathbf{U}_2,\mathbf{T}_3\rangle) \text{ if } \sqcup(\mathbf{T}_1,\mathbf{T}_2,\mathbf{T}_3) \text{ is in } \mathcal{R} \text{ and } \mathbf{T}_3 <: \mathbf{U}_2\} \\
&\cup \{\sqcap(\mathsf{rw}\langle\mathbf{U}_1,\mathbf{T}_1\rangle,\mathsf{rw}\langle\mathbf{U}_2,\mathbf{T}_2\rangle,\mathsf{rw}\langle\mathbf{U}_3,\mathbf{T}_3\rangle) \\
&\qquad \text{if } \sqcup(\mathbf{T}_1,\mathbf{T}_2,\mathbf{T}_3), \sqcap(\mathbf{U}_1,\mathbf{U}_2,\mathbf{U}_3) \text{ are in } \mathcal{R} \text{ and } \mathbf{T}_3 <: \mathbf{U}_3\} \\
&\cup \{\sqcap(\mathbf{A}_1@\mathbf{K}_1,\mathbf{A}_2@\mathbf{K}_2,\mathbf{A}_3@\mathbf{K}_3) \text{ if } \sqcap(\mathbf{A}_1,\mathbf{A}_2,\mathbf{A}_3) \text{ and } \sqcap(\mathbf{K}_1,\mathbf{K}_2,\mathbf{K}_3) \text{ are in } \mathcal{R}\} \\
&\cup \{\sqcap(\mathsf{loc}[u_1:\mathbf{A}_1,\ldots,u_n:\mathbf{A}_n,v_1:\mathbf{B}_1,\ldots],\mathsf{loc}[u_1:\mathbf{A}'_1,\ldots,u_n:\mathbf{A}'_n,w_1:\mathbf{B}'_1,\ldots], \\
&\qquad \mathsf{loc}[u_1:\mathbf{A}''_1,\ldots,u_n:\mathbf{A}''_n,v_1:\mathbf{B}_1,\ldots,w_1:\mathbf{B}'_1,\ldots]) \\
&\qquad \text{if } \sqcap(\mathbf{A}_i,\mathbf{A}'_i,\mathbf{A}''_i) \text{ is in } \mathcal{R} \text{ for all } i \le n\} \\
&\cup \{\sqcup(\mathbf{base}, \mathbf{base}, \mathbf{base})\} \\
&\cup \{\sqcup(u:\mathbf{A}_1, u:\mathbf{A}_2, u:\mathbf{A}_3) \text{ if } \sqcup(\mathbf{A}_1,\mathbf{A}_2,\mathbf{A}_3) \text{ is in } \mathcal{R}\} \\
&\cup \{\sqcup((\widetilde{\mathbf{C}}),(\widetilde{\mathbf{C}'}),(\widetilde{\mathbf{C}''})) \text{ if } \sqcup(\mathbf{C}_i,\mathbf{C}'_i,\mathbf{C}''_i) \text{ is in } \mathcal{R} \text{ for all } i\} \\
&\cup \{\sqcup(\mathsf{r}\langle\mathbf{U}_1\rangle,\mathsf{r}\langle\mathbf{U}_2\rangle,\mathsf{r}\langle\mathbf{U}_3\rangle) \text{ if } \sqcup(\mathbf{U}_1,\mathbf{U}_2,\mathbf{U}_3) \text{ is in } \mathcal{R}\} \\
&\cup \{\sqcup(\mathsf{w}\langle\mathbf{T}_1\rangle,\mathsf{w}\langle\mathbf{T}_2\rangle,\mathsf{w}\langle\mathbf{T}_3\rangle) \text{ if } \sqcap(\mathbf{T}_1,\mathbf{T}_2,\mathbf{T}_3) \text{ is in } \mathcal{R}\} \\
&\cup \{\sqcup(\mathsf{rw}\langle\mathbf{U}_1,\mathbf{T}_1\rangle,\mathsf{r}\langle\mathbf{U}_2\rangle,\mathsf{r}\langle\mathbf{U}_3\rangle) \text{ if } \sqcup(\mathbf{U}_1,\mathbf{U}_2,\mathbf{U}_3) \text{ is in } \mathcal{R} \text{ and } \mathbf{T}_1 <: \mathbf{U}_1\} \\
&\cup \{\sqcup(\mathsf{rw}\langle\mathbf{U}_1,\mathbf{T}_1\rangle,\mathsf{w}\langle\mathbf{T}_2\rangle,\mathsf{w}\langle\mathbf{T}_3\rangle) \text{ if } \sqcap(\mathbf{T}_1,\mathbf{T}_2,\mathbf{T}_3) \text{ is in } \mathcal{R} \text{ and } \mathbf{T}_1 <: \mathbf{U}_1\} \\
&\cup \{\sqcup(\mathsf{r}\langle\mathbf{U}_1\rangle,\mathsf{rw}\langle\mathbf{U}_2,\mathbf{T}_2\rangle,\mathsf{r}\langle\mathbf{U}_3\rangle) \text{ if } \sqcup(\mathbf{U}_1,\mathbf{U}_2,\mathbf{U}_3) \text{ is in } \mathcal{R} \text{ and } \mathbf{T}_2 <: \mathbf{U}_2\} \\
&\cup \{\sqcup(\mathsf{w}\langle\mathbf{T}_1\rangle,\mathsf{rw}\langle\mathbf{U}_2,\mathbf{T}_2\rangle,\mathsf{w}\langle\mathbf{T}_3\rangle) \text{ if } \sqcap(\mathbf{T}_1,\mathbf{T}_2,\mathbf{T}_3) \text{ is in } \mathcal{R} \text{ and } \mathbf{T}_2 <: \mathbf{U}_2\} \\
&\cup \{\sqcup(\mathsf{rw}\langle\mathbf{U}_1,\mathbf{T}_1\rangle,\mathsf{rw}\langle\mathbf{U}_2,\mathbf{T}_2\rangle,\mathsf{r}\langle\mathbf{U}_3\rangle) \\
&\qquad \text{if } \mathbf{U}_1 \uparrow \mathbf{U}_2, \sqcup(\mathbf{U}_1,\mathbf{U}_2,\mathbf{U}_3) \text{ is in } \mathcal{R}, \mathbf{T}_1 \not<: \mathbf{T}_2, \mathbf{T}_1 <: \mathbf{U}_1 \text{ and } \mathbf{T}_2 <: \mathbf{U}_2\} \\
&\cup \{\sqcup(\mathsf{rw}\langle\mathbf{U}_1,\mathbf{T}_1\rangle,\mathsf{rw}\langle\mathbf{U}_2,\mathbf{T}_2\rangle,\mathsf{w}\langle\mathbf{T}_3\rangle) \\
&\qquad \text{if } \mathbf{T}_1 \downarrow \mathbf{T}_2, \sqcap(\mathbf{T}_1,\mathbf{T}_2,\mathbf{T}_3) \text{ is in } \mathcal{R}, \mathbf{U}_1 \not<: \mathbf{U}_2, \mathbf{T}_1 <: \mathbf{U}_1 \text{ and } \mathbf{T}_2 <: \mathbf{U}_2\} \\
&\cup \{\sqcup(\mathsf{rw}\langle\mathbf{U}_1,\mathbf{T}_1\rangle,\mathsf{rw}\langle\mathbf{U}_2,\mathbf{T}_2\rangle,\mathsf{rw}\langle\mathbf{U}_3,\mathbf{T}_3\rangle) \\
&\qquad \text{if } \mathbf{T}_1 \downarrow \mathbf{T}_2, \mathbf{U}_1 \uparrow \mathbf{U}_2, \sqcap(\mathbf{T}_1,\mathbf{T}_2,\mathbf{T}_3), \sqcup(\mathbf{U}_1,\mathbf{U}_2,\mathbf{U}_3) \text{ are in } \mathcal{R}, \\
&\qquad\quad \mathbf{T}_1 <: \mathbf{U}_1 \text{ and } \mathbf{T}_2 <: \mathbf{U}_2\} \\
&\cup \{\sqcup(\mathbf{A}_1@\mathbf{K}_1,\mathbf{A}_2@\mathbf{K}_2,\mathbf{A}_3@\mathbf{K}_3) \text{ if } \sqcup(\mathbf{A}_1,\mathbf{A}_2,\mathbf{A}_3) \text{ and } \sqcup(\mathbf{K}_1,\mathbf{K}_2,\mathbf{K}_3) \text{ are in } \mathcal{R}\} \\
&\cup \{\sqcup(\mathsf{loc}[u_1:\mathbf{A}_1,\ldots,u_n:\mathbf{A}_n,v_1:\mathbf{B}_1,\ldots],\mathsf{loc}[u_1:\mathbf{A}'_1,\ldots,u_n:\mathbf{A}'_n,w_1:\mathbf{B}'_1,\ldots], \\
&\qquad \mathsf{loc}[u_{i_1}:\mathbf{A}''_{i_1},\ldots,u_{i_p}:\mathbf{A}''_{i_p}]) \\
&\qquad \text{if } \mathbf{A}_k \uparrow \mathbf{A}'_k \text{ for all } k \text{ in } \{i_j\}, \mathbf{A}_k \not\uparrow \mathbf{A}'_k \text{ for all } k \text{ not in } \{i_j\}, \\
&\qquad \sqcup(\mathbf{A}_{i_j},\mathbf{A}'_{i_j},\mathbf{A}''_{i_j}) \text{ is in } \mathcal{R} \text{ for all } i_j\}
\end{aligned}
$$

Lemma 3.8 (Symmetry on $\sqcap$ arguments). *For any types* $\mathbf{T}_1$, $\mathbf{T}_2$ *and* $\mathbf{T}_3$, $\mathbf{T}_1 \sqcap \mathbf{T}_2 = \mathbf{T}_3$ *if and only if* $\mathbf{T}_2 \sqcap \mathbf{T}_1 = \mathbf{T}_3$.

Proof. Since the definition of MeetJoin is completely symmetric on its first two components, it is enough to consider the relation

$$\mathcal{R} = \{{}_{\sqcap}(\mathbf{T}_2, \mathbf{T}_1, \mathbf{T}_3) \mid \mathbf{T}_1 \sqcap \mathbf{T}_2 = \mathbf{T}_3\} \cup \{{}_{\sqcup}(\mathbf{T}_2, \mathbf{T}_1, \mathbf{T}_3) \mid \mathbf{T}_1 \sqcup \mathbf{T}_2 = \mathbf{T}_3\}$$

which is easily shown to be a postfixpoint of the operator MeetJoin. □

Lemma 3.9 (Reflexivity of $\sqcap$ and $\sqcup$). *For any type* $\mathbf{T}$, *we have* $\mathbf{T} \sqcap \mathbf{T} = \mathbf{T}$ *and* $\mathbf{T} \sqcup \mathbf{T} = \mathbf{T}$.

Proof. We simply consider the relation

$$\mathcal{R} = \{{}_{\sqcap}(\mathbf{T}, \mathbf{T}, \mathbf{T}), {}_{\sqcup}(\mathbf{T}, \mathbf{T}, \mathbf{T}) \mid \mathbf{T} \text{ is any type}\}$$

and prove that it is a postfixpoint of MeetJoin.

For this, let us consider a triple in $\mathcal{R}$. We reason on the form of that triple, namely on the head construct for the type it is based on, and on the operator, $\sqcap$ or $\sqcup$. All the cases are very similar, we give only a few of them.

- ${}_{\sqcap}(\mathbf{base}, \mathbf{base}, \mathbf{base})$. Obviously this triple is also in $\mathsf{MeetJoin}(\mathcal{R})$.
- ${}_{\sqcap}(\mathsf{rw}\langle \mathbf{U}, \mathbf{T}\rangle, \mathsf{rw}\langle \mathbf{U}, \mathbf{T}\rangle, \mathsf{rw}\langle \mathbf{U}, \mathbf{T}\rangle)$. As we know that both ${}_{\sqcap}(\mathbf{U}, \mathbf{U}, \mathbf{U})$ and ${}_{\sqcup}(\mathbf{T}, \mathbf{T}, \mathbf{T})$ are also in $\mathcal{R}$, we can conclude that our triple is in $\mathsf{MeetJoin}(\mathcal{R})$.
- ${}_{\sqcup}(\mathsf{loc}[u_i : \mathbf{A}_i], \mathsf{loc}[u_i : \mathbf{A}_i], \mathsf{loc}[u_i : \mathbf{A}_i])$ is in $\mathsf{MeetJoin}(\mathcal{R})$ because of the triples ${}_{\sqcup}(\mathbf{A}_i, \mathbf{A}_i, \mathbf{A}_i)$ in $\mathcal{R}$.

□

Lemma 3.10 (Meet is a subtype). *For any types* $\mathbf{T}_1$, $\mathbf{T}_2$ *and* $\mathbf{T}_3$ *such that* $\mathbf{T}_1 \sqcap \mathbf{T}_2 = \mathbf{T}_3$, *we have* $\mathbf{T}_3 <: \mathbf{T}_1$ *and* $\mathbf{T}_3 <: \mathbf{T}_2$.

Proof. We need to prove this result and its dual about $\sqcup$ at the same time. For this, let us consider the relation

$$\mathcal{R} = \{(\mathbf{T}^1, \mathbf{T}^2) \mid \exists \mathbf{T}^3 \text{ such that } \mathbf{T}^2 \sqcap \mathbf{T}^3 = \mathbf{T}^1 \text{ or } \mathbf{T}^1 \sqcup \mathbf{T}^3 = \mathbf{T}^2\} \cup \nu\mathsf{Sub}$$

We now prove that this relation is a postfixpoint of Sub.

Let us consider $(\mathbf{T}^1, \mathbf{T}^2)$ in $\mathcal{R}$. If that pair comes from the $\nu\mathsf{Sub}$ part of $\mathcal{R}$, we know that it is in $\mathsf{Sub}(\nu\mathsf{Sub}) \subseteq \mathsf{Sub}(\mathcal{R})$. Otherwise, let us write $\mathbf{T}^3$ the type proving that $(\mathbf{T}^1, \mathbf{T}^2)$ is in $\mathcal{R}$. We reason on the proof of $\mathbf{T}^2 \sqcap \mathbf{T}^3 = \mathbf{T}^1$ or of $\mathbf{T}^1 \sqcup \mathbf{T}^3 = \mathbf{T}^2$. Let us start with $\mathbf{T}^2 \sqcap \mathbf{T}^3 = \mathbf{T}^1$. We give here only some typical examples.

- $\mathbf{T}^1 = \mathbf{T}^2 = \mathbf{T}^3 = \mathbf{base}$, then $(\mathbf{T}^1, \mathbf{T}^2)$ is obviously in $\mathcal{R}$.

- $\mathbf{T}^i = \widetilde{\mathbf{C}^i}$, with $\mathbf{C}^2_j \sqcap \mathbf{C}^3_j = \mathbf{C}^1_j$ for all j. Then, for all j, $\mathbf{C}^1_j \mathcal{R} \mathbf{C}^2_j$, which implies that $(\mathbf{T}^1, \mathbf{T}^2)$ is in $\mathcal{R}$.
- If the triple is $\sqcap(\mathsf{r}\langle \mathbf{U}^2_0\rangle, \mathsf{w}\langle \mathbf{T}^3_0\rangle, \mathsf{rw}\langle \mathbf{U}^2_0, \mathbf{T}^3_0\rangle)$ we know that $\mathbf{T}^3_0 <: \mathbf{U}^2_0$ so $(\mathbf{T}^3_0, \mathbf{U}^2_0)$ is in particular in $\mathcal{R}$. Moreover, by lemma 3.4, we know that $\mathbf{U}^2_0 <: \mathbf{U}^2_0$. These two hypotheses allow us to conclude that $(\mathsf{rw}\langle \mathbf{U}^2_0, \mathbf{T}^3_0\rangle), \mathsf{r}\langle \mathbf{U}^2_0\rangle)$ is in $\mathsf{Sub}(\mathcal{R})$.
- If the triple is $\sqcap(\mathsf{rw}\langle \mathbf{U}^2_0, \mathbf{T}^2_0\rangle, \mathsf{r}\langle \mathbf{U}^3_0\rangle, \mathsf{rw}\langle \mathbf{U}^1_0, \mathbf{T}^2_0\rangle)$ we know that $\mathbf{U}^2_0 \sqcap \mathbf{U}^3_0 = \mathbf{U}^1_0$ so $(\mathbf{U}^1_0, \mathbf{U}^2_0)$ is in $\mathcal{R}$ and that $\mathbf{T}^2_0 <: \mathbf{U}^1_0$ so $(\mathbf{T}^2_0, \mathbf{U}^1_0)$ is in particular in $\mathcal{R}$. Moreover, by lemma 3.4, we know that $\mathbf{T}^2_0 <: \mathbf{T}^2_0$. These three hypotheses allow us to conclude that $(\mathsf{rw}\langle \mathbf{U}^1_0, \mathbf{T}^2_0\rangle, \mathsf{rw}\langle \mathbf{U}^2_0, \mathbf{T}^2_0\rangle)$ is in $\mathsf{Sub}(\mathcal{R})$.
- If the triple is $\sqcup(\mathsf{loc}[u_i : \mathbf{A}^2_i, v_j : \mathbf{B}^2_j], \mathsf{loc}[u_i : \mathbf{A}^3_i, w_k : \mathbf{B}^3_k], \mathsf{loc}[u_i : \mathbf{A}^1_i, v_j : \mathbf{B}^2_j, w_k : \mathbf{B}^3_k])$, we know that $(\mathbf{A}^1_i, \mathbf{A}^2_i)$ are in $\mathcal{R}$ and so are $(\mathbf{B}^2_j, \mathbf{B}^2_j)$ by Lemma 3.4. So $(\mathbf{T}^1, \mathbf{T}^2)$ is in $\mathsf{Sub}(\mathcal{R})$.

The different cases for $\mathbf{T}^1 \sqcup \mathbf{T}^3 = \mathbf{T}^2$ are similar.

So we have proved that $\mathcal{R}$ is a subset of $\nu\mathsf{Sub}$, from which the result follows. □

Lemma 3.11 (Meet is the greatest subtype). *For any types* $\mathbf{T}_1$, $\mathbf{T}_2$, $\mathbf{T}_3$ *and* $\mathbf{T}$ *such that* $\mathbf{T}_1 \sqcap \mathbf{T}_2 = \mathbf{T}_3$, $\mathbf{T} <: \mathbf{T}_1$ *and* $\mathbf{T} <: \mathbf{T}_2$, *we have* $\mathbf{T} <: \mathbf{T}_3$.

Proof. Let

$$\begin{aligned}\mathcal{R} = \{&(\mathbf{T}, \mathbf{T}^3) \mid \exists \mathbf{T}^1, \mathbf{T}^2 \text{ such that } \mathbf{T} <: \mathbf{T}^1, \mathbf{T} <: \mathbf{T}^2, \mathbf{T}^1 \sqcap \mathbf{T}^2 = \mathbf{T}^3\}\\ \cup\{&(\mathbf{T}^3, \mathbf{T}) \mid \exists \mathbf{T}^1, \mathbf{T}^2 \text{ such that } \mathbf{T}^1 <: \mathbf{T}, \mathbf{T}^2 <: \mathbf{T}, \mathbf{T}^1 \sqcup \mathbf{T}^2 = \mathbf{T}^3\}\\ \cup\,&\nu\mathsf{Sub}\end{aligned}$$

We now prove that $\mathcal{R}$ is included in $\nu\mathsf{Sub}$.

Let us consider a pair in $\mathcal{R}$. We have three possible cases. Let us first suppose this pair is of the form $(\mathbf{T}, \mathbf{T}^3)$, with the corresponding types $\mathbf{T}^1$ and $\mathbf{T}^2$. We reason on $\mathbf{T}^1 \sqcap \mathbf{T}^2 = \mathbf{T}^3$. As usual, we give some typical cases.

- $(\widetilde{\mathbf{C}^1}) \sqcap (\widetilde{\mathbf{C}^2}) = (\widetilde{\mathbf{C}^3})$ then $\mathbf{T} <: \mathbf{T}^1$ implies that $\mathbf{T}$ is of the form $(\widetilde{\mathbf{C}})$ and for each i, we have $\mathbf{C}_i <: \mathbf{C}^1_i$, $\mathbf{C}_i <: \mathbf{C}^2_i$ and $\mathbf{C}^1_i \sqcap \mathbf{C}^2_i = \mathbf{C}^3_i$, which means that, for each i, $(\mathbf{C}_i, \mathbf{C}^3_i)$ is in $\mathcal{R}$. Consequently $(\mathbf{T}, \mathbf{T}^3)$ is in $\mathsf{Sub}(\mathcal{R})$.
- $\mathsf{r}\langle \mathbf{U}^1_0\rangle \sqcap \mathsf{w}\langle \mathbf{T}^2_0\rangle = \mathsf{rw}\langle \mathbf{U}^1_0, \mathbf{T}^2_0\rangle$ then $\mathbf{T} <: \mathbf{T}^1$ implies that $\mathbf{T}$ can only be of the form $\mathsf{r}\langle \mathbf{U}_0\rangle$ or $\mathsf{rw}\langle \mathbf{U}_0, \mathbf{T}_0\rangle$ with $\mathbf{U}_0 <: \mathbf{U}^1_0$. Similarly, $\mathbf{T} <: \mathbf{T}^2$ implies that $\mathbf{T}$ can only be of the form $\mathsf{w}\langle \mathbf{T}_0\rangle$ or $\mathsf{rw}\langle \mathbf{U}_0, \mathbf{T}_0\rangle$ with $\mathbf{T}^2_0 <: \mathbf{T}_0$. By combining those two constraints, we know it must of

the form $\mathsf{rw}\langle \mathbf{U}_0, \mathbf{T}_0\rangle$ with $\mathbf{U}_0 <: \mathbf{U}_0^1$ and $\mathbf{T}_0^2 <: \mathbf{T}_0$. By well-formedness of $\mathbf{T}$ we also know that $\mathbf{T}_0 <: \mathbf{U}_0$. Which means that $(\mathbf{T}, \mathbf{T}^3)$ is in $\mathsf{Sub}(\nu\mathsf{Sub}) \subseteq \mathsf{Sub}(\mathcal{R})$.

- $\mathsf{r}\langle \mathbf{U}_0^1\rangle \sqcap \mathsf{rw}\langle \mathbf{U}_0^2, \mathbf{T}_0^2\rangle = \mathsf{rw}\langle \mathbf{U}_0^3, \mathbf{T}_0^2\rangle$, then $\mathbf{T} <: \mathbf{T}^2$ implies that $\mathbf{T}$ is of the form $\mathsf{rw}\langle \mathbf{U}_0, \mathbf{T}_0\rangle$ with $\mathbf{U}_0 <: \mathbf{U}_0^2$ and $\mathbf{T}_0^2 <: \mathbf{T}_0$. We also have that $\mathbf{U}_0 <: \mathbf{U}_0^1$. Of course, we have that $\mathbf{U}_0^1 \sqcap \mathbf{U}_0^2 = \mathbf{U}_0^3$, so $(\mathbf{U}_0, \mathbf{U}_0^3)$ is in $\mathcal{R}$. As so is $(\mathbf{T}_0^2, \mathbf{T}_0)$ and $(\mathbf{T}_0, \mathbf{U}_0)$ by well-formedness of $\mathbf{T}$, $(\mathbf{T}, \mathbf{T}^3)$ is in $\mathsf{Sub}(\mathcal{R})$.

- $\mathsf{loc}[u_i : A1_i, v_j : B1_j] \sqcap \mathsf{loc}[u_i : A2_i, w_k : B2_k] = \mathsf{loc}[u_i : A3_i, v_j : B1_j, w_k : B2_k]$, which implies that $\mathbf{A}_i^1 \sqcap \mathbf{A}_i^2 = \mathbf{A}_i^3$ for all i. The fact $\mathbf{T}$ is a common subtype of $\mathbf{T}^1$ and $\mathbf{T}^2$ implies that it must be of the form $\mathsf{loc}[u_i : AA_i, v_j : B4_j, w_k : B5_k, x_l : B6_l]$ with $\mathbf{A}_i <: \mathbf{A}_i^1$ and $\mathbf{A}_i <: \mathbf{A}_i^2$ for all i, and with $\mathbf{B}_j^4 <: \mathbf{B}_j^1$ and $\mathbf{B}_k^5 <: \mathbf{B}_k^2$. This implies that $(\mathbf{A}_i, \mathbf{A}_i^3)$, $(\mathbf{B}_j^4, \mathbf{B}_j^1)$ and $(\mathbf{B}_k^5, \mathbf{B}_k^2)$ are in $\mathcal{R}$. So $(\mathbf{T}, \mathbf{T}^3)$ is in $\mathsf{Sub}(\mathcal{R})$.

For the second case, let us now suppose that the pair is of the form $(\mathbf{T}^3, \mathbf{T})$, with the corresponding types $\mathbf{T}^1$ and $\mathbf{T}^2$. We reason on $\mathbf{T}^1 \sqcup \mathbf{T}^2 = \mathbf{T}^3$.

- $\mathsf{rw}\langle \mathbf{U}_0^1, \mathbf{T}_0^1\rangle \sqcup \mathsf{rw}\langle \mathbf{U}_0^2, \mathbf{T}_0^2\rangle = \mathsf{r}\langle \mathbf{U}_0^3\rangle$ which implies that $\mathbf{U}_0^1 \sqcup \mathbf{U}_0^2 = \mathbf{U}_0^3$ and $\mathbf{T}_0^1 \mathbf{T} \not\downarrow_0^2$. If $\mathbf{T}$ was of the form $\mathsf{w}\langle \mathbf{T}_0\rangle$ or $\mathsf{rw}\langle \mathbf{U}_0, \mathbf{T}_0\rangle$, $\mathbf{T}^1 <: \mathbf{T}$ and $\mathbf{T}^2 <: \mathbf{T}$ would imply $\mathbf{T}_0 <: \mathbf{T}_0^1$ and $\mathbf{T}_0 <: \mathbf{T}_0^2$, which would contradict the fact that those two types are incompatible. So $\mathbf{T}$ must be of the form $\mathsf{r}\langle \mathbf{U}_0\rangle$, with $\mathbf{U}_0^1 <: \mathbf{U}_0$ and $\mathbf{U}_0^2 <: \mathbf{U}_0$ which means that $(\mathbf{U}_0, \mathbf{U}_0^3)$ is in $\mathcal{R}$ and $(\mathbf{T}, \mathbf{T}^3)$ in $\mathsf{Sub}(\mathcal{R})$.

The last case is that the pair is in $\nu\mathsf{Sub}$, which means that it is obviously in $\mathsf{Sub}(\mathcal{R})$.

So we have proved that $\mathcal{R}$ is a subset of $\nu\mathsf{Sub}$, which finishes our proof. □

The major property of the meet operator is given by the following lemma. This lemma is again proved by using that approach of coinductive proofs. To state it, let us write $\mathbf{T}_1 \downarrow \mathbf{T}_2$ to mean that there is some $\mathbf{T}$ such that $\mathbf{T} <: \mathbf{T}_1$ and $\mathbf{T} <: \mathbf{T}_2$, that is $\mathbf{T}_1$ and $\mathbf{T}_2$ are compatible.

Theorem 3.12 (Partial meets). *The set of tree types, ordered by* $<:$*, has partial meets. That is* $\mathbf{T}_1 \downarrow \mathbf{T}_2$ *implies* $\mathbf{T}_1$ *and* $\mathbf{T}_2$ *have a meet.*

Proof. Let us consider two types $\mathbf{T}_1$ and $\mathbf{T}_2$ that are compatible. Their compatibility implies that the set $\{\mathbf{T} \mid \mathbf{T} <: \mathbf{T}_1, \mathbf{T} <: \mathbf{T}_2\}$ is not empty. So we can consider its set of maximal elements, which are elements $\mathbf{T}$ such that for any element $\mathbf{T}'$, $\mathbf{T} <: \mathbf{T}'$ implies $\mathbf{T} = \mathbf{T}'$. Notice that this equality

is the one we previously defined by Eq. We write $M(\mathbf{T}_1, \mathbf{T}_2)$ for this set of maximal elements, and $m(\mathbf{T}_1, \mathbf{T}_2)$ for its dual, namely the set of minimal elements among the shared supertypes of $\mathbf{T}_1$ and $\mathbf{T}_2$.

Now, let us consider the relation

$$\begin{aligned}\mathcal{R} =&\{{\sqcap}(\mathbf{T}_1, \mathbf{T}_2, \mathbf{T}_3) \mid \mathbf{T}_1 \downarrow \mathbf{T}_2 \text{ and } \mathbf{T}_3 \in M(\mathbf{T}_1, \mathbf{T}_2)\}\\ &\cup\{{\sqcup}(\mathbf{T}_1, \mathbf{T}_2, \mathbf{T}_3) \mid \mathbf{T}_1 \uparrow \mathbf{T}_2 \text{ and } \mathbf{T}_3 \in m(\mathbf{T}_1, \mathbf{T}_2)\}\\ &\cup\nu\mathsf{MeetJoin}\end{aligned}$$

Now, let us prove that this is a postfixpoint of $\mathsf{MeetJoin}$.

Let us consider t in $\mathcal{R}$ and let us prove that t is in $\mathsf{MeetJoin}(\mathcal{R})$. If t is in $\nu\mathsf{MeetJoin}$, this is obvious. If it is of the form ${\sqcap}(\mathbf{T}_1, \mathbf{T}_2, \mathbf{T}_3)$, $\mathbf{T}_3 <: \mathbf{T}_1$ must correspond to one case of the definition of Sub.

- $\mathbf{T}_3 <: \mathbf{T}_1 = \mathbf{base}$; then $\mathbf{T}_2$ must be $\mathbf{base}$ too since $\mathbf{T}_3 <: \mathbf{T}_2$ so they are in $\nu\mathsf{MeetJoin}$ so in $\mathcal{R}$.
- $\mathbf{T}_3 = u : \mathbf{A}_3 <: u : \mathbf{A}_1 = \mathbf{T}_1$; then $\mathbf{T}_2$ must be of the form $u : \mathbf{A}_2$ too. We also know that $\mathbf{A}_3$ is a common subtype of $\mathbf{A}_1$ and $\mathbf{A}_2$. Let us consider $\mathbf{A}'_3$ a common subtype of $\mathbf{A}_1$ and $\mathbf{A}_2$ such that $\mathbf{A}_3 <: \mathbf{A}'_3$. So we have $\mathbf{T}_3 <: u : \mathbf{A}'_3$ with $u : \mathbf{A}'_3$ a common subtype of $\mathbf{T}_1$ and $\mathbf{T}_2$. By maximality of $\mathbf{T}_3$, this implies that $\mathbf{T}_3 = u : \mathbf{A}'_3$. Only one case of the definition of Eq can be applied to get this pair $(\mathbf{T}_3, u : \mathbf{A}'_3)$, so we can deduce that $\mathbf{A}_3 = \mathbf{A}'_3$. So $\mathbf{A}_3$ is maximal in the set of common subtypes of $\mathbf{A}_1$ and $\mathbf{A}_2$. This means that ${\sqcap}(\mathbf{A}_1, \mathbf{A}_2, \mathbf{A}_3)$ must be in $\mathcal{R}$ so t is $\mathsf{MeetJoin}(\mathcal{R})$.
- $\mathbf{T}_3 = \mathsf{rw}\langle\mathbf{U}^1_3, \mathbf{T}^1_3\rangle <: \mathsf{r}\langle\mathbf{U}^1_1\rangle = \mathbf{T}_1$; as a supertype of $\mathbf{T}_3$, $\mathbf{T}_2$ can be of the forms $\mathsf{r}\langle\mathbf{U}^1_2\rangle$, $\mathsf{w}\langle\mathbf{T}^1_2\rangle$ or $\mathsf{rw}\langle\mathbf{U}^1_2, \mathbf{T}^1_2\rangle$.
  - If it is of the form $\mathsf{r}\langle\mathbf{U}^1_2\rangle$ then $\mathsf{r}\langle\mathbf{U}^1_3\rangle$ would be a common subtype of $\mathbf{T}_1$ and $\mathbf{T}_2$ contradicting the maximality of $\mathbf{T}_3$.
  - If it is of the form $\mathsf{w}\langle\mathbf{T}^1_2\rangle$, we know that all the common subtypes of $\mathbf{T}_1$ and $\mathbf{T}_2$ are of the form $\mathsf{rw}\langle\mathbf{U}^2_3, \mathbf{T}^2_3\rangle$ with $\mathbf{U}^2_3 <: \mathbf{U}^1_1$ and $\mathbf{T}^1_2 <: \mathbf{T}^2_3$, so $\mathsf{rw}\langle\mathbf{U}^1_1, \mathbf{T}^1_2\rangle$ is the only maximal common subtype of $\mathbf{T}_1$ and $\mathbf{T}_2$, so it must be the value for $\mathbf{T}_3$, in which case t is directly in $\mathsf{MeetJoin}(\mathcal{R})$.
  - And lastly, if it is of the form $\mathsf{rw}\langle\mathbf{U}^1_2, \mathbf{T}^1_2\rangle$, with $\mathbf{U}^1_3 <: \mathbf{U}^1_2$ and $\mathbf{T}^1_2 <: \mathbf{T}^1_3$. We know that $\mathbf{T}^1_2 = \mathbf{T}^1_3$ since $\mathbf{T}_3$ is maximal. That maximality also implies that $\mathbf{U}^1_3$ is a maximal common subtype of $\mathbf{U}^1_1$ and $\mathbf{U}^1_2$. So $\{{\sqcap}(\mathbf{U}^1_1, \mathbf{U}^1_2, \mathbf{U}^1_3)\}$ must be in $\mathcal{R}$ which proves that t is in $\mathsf{MeetJoin}(\mathcal{R})$.

If t is of the form $\sqcup(\mathbf{T}_1, \mathbf{T}_2, \mathbf{T}_3)$, we reason on the fact that $\mathbf{T}_3$ is a supertype of $\mathbf{T}_1$. Let us just see the most interesting case, $\mathsf{loc}[\ldots]$.

- $\mathbf{T}_1 = \mathsf{loc}[u_1 : \mathbf{A}_1, \ldots, u_n : \mathbf{A}_n, u_{n+1} : \mathbf{A}_{n+1}, \ldots, v_1 : \mathbf{B}_1, \ldots]$ and $\mathbf{T}_3 = \mathsf{loc}[u_1 : \mathbf{A}''_1, \ldots, u_n : \mathbf{A}''_n]$. Then $\mathbf{T}_2$ must be of the form $\mathsf{loc}[u_1 : \mathbf{A}'_1, \ldots, u_n : \mathbf{A}'_n, u_{n+1} : \mathbf{A}'_{n+1}, \ldots, w_1 : \mathbf{B}'_1, \ldots]$. Since $\mathbf{T}_3$ is minimal among the common supertypes of $\mathbf{T}_1$ and $\mathbf{T}_2$, we know that, for every j, $\mathbf{A}_{n+j} \not\uparrow \mathbf{A}'_{n+j}$: otherwise, if $\mathbf{A}_{n+1} \uparrow \mathbf{A}'_{n+1}$, we can define the type $\mathbf{A}''_{n+1}$ as a supertype of $\mathbf{A}_{n+1}$ and $\mathbf{A}'_{n+1}$, and consider $\mathsf{loc}[u_1 : \mathbf{A}''_1, \ldots, u_n : \mathbf{A}''_n, u_{n+1} : \mathbf{A}''_{n+1}]$, contradicting the minimality of $\mathbf{T}_3$. By minimality of $\mathbf{T}_3$, we can also conclude that every $\mathbf{A}''_i$ is minimal in its set of common supertypes. So every $\sqcup(\mathbf{A}_i, \mathbf{A}'_i, \mathbf{A}''_i)$ is $\mathcal{R}$ which implies that t must be in $\mathsf{MeetJoin}(\mathcal{R})$.

This concludes the proof because $\mathcal{R}$ is included in $\nu\mathsf{MeetJoin}$, which means that we have found a meet for every pair of compatible types. □

3.3 Theory of recursive types

Notice that all the properties we mentioned deal with tree types, because all the proofs rely on coinductive techniques. But, in the end, the types we really want to use in our terms are recursive, since we want to be able to denote them with the recursive operator μ. So we need to prove that all the properties we considered on tree types can be lifted up to recursive types.

For this, we define notions of subtyping and meet on recursive types by using a set of rules of the form $\Sigma \vdash \mathsf{T}_1 <: \mathsf{T}_2$ or $\Sigma \vdash \mathsf{T}_1 \sqcap \mathsf{T}_2 = \mathsf{T}_3$. The intuition is that the manipulation of regular trees relies on normal operations with unfolding rules and some "memory", Σ, to record which subterms have already been "seen".

The termination of the proofs we will give on those recursive types will be based on the following notion of subterms.

FIGURE 7 Subtyping rules

(SR-AX)
$$\frac{}{\Sigma, \mathsf{T}_1 <: \mathsf{T}_2 \vdash \mathsf{T}_1 <: \mathsf{T}_2}$$

(SR-BASE)
$$\frac{}{\Sigma \vdash \mathbf{base} <: \mathbf{base}}$$

(SR-CAP)
$$\frac{\Sigma \vdash \mathsf{A} <: \mathsf{B}}{\Sigma \vdash u : \mathsf{A} <: u : \mathsf{B}}$$

(SR-TUPLE)
$$\frac{\Sigma \vdash \mathsf{C}_i <: \mathsf{C}'_i}{\Sigma \vdash (\widetilde{\mathsf{C}}) <: (\widetilde{\mathsf{C}'})}$$

(SR-HOM)
$$\frac{\Sigma \vdash \mathsf{A}_1 <: \mathsf{A}_2 \quad \Sigma \vdash \mathsf{K}_1 <: \mathsf{K}_2}{\Sigma \vdash \mathsf{A}_1 @ \mathsf{K}_1 <: \mathsf{A}_2 @ \mathsf{K}_2}$$

(SR-CHAN)
$$\frac{\Sigma \vdash \mathsf{T}_1 <: \mathsf{T}_2 <: \mathsf{U}_1 <: \mathsf{U}_2}{\begin{array}{l}\Sigma \vdash \mathrm{W}\langle \mathsf{T}_2 \rangle <: \mathrm{W}\langle \mathsf{T}_1 \rangle \\ \Sigma \vdash \mathrm{R}\langle \mathsf{U}_1 \rangle <: \mathrm{R}\langle \mathsf{U}_2 \rangle \\ \Sigma \vdash \mathrm{RW}\langle \mathsf{U}_1, \mathsf{T}_2 \rangle <: \mathrm{R}\langle \mathsf{U}_2 \rangle \\ \Sigma \vdash \mathrm{RW}\langle \mathsf{U}_1, \mathsf{T}_2 \rangle <: \mathrm{W}\langle \mathsf{T}_1 \rangle \\ \Sigma \vdash \mathrm{RW}\langle \mathsf{U}_1, \mathsf{T}_2 \rangle <: \mathrm{RW}\langle \mathsf{U}_2, \mathsf{T}_1 \rangle\end{array}}$$

(SR-LOC)
$$\frac{\Sigma \vdash \mathsf{U}_i <: \mathsf{U}'_i, \;\; 0 \leq i \leq n}{\Sigma \vdash \mathrm{LOC}[u_1 : \mathsf{U}_1, \ldots, u_n : \mathsf{U}_n, \ldots, u_{n+p} : \mathsf{U}_{n+p}] <: \mathrm{LOC}[u_1 : \mathsf{U}'_1, \ldots, u_n : \mathsf{U}'_n]}$$

(SR-LREC)
$$\frac{\Sigma, \mu t_1.\mathsf{T}_1 <: \mathsf{T}_2 \vdash \mathsf{T}_1\{^{\mu t_1.\mathsf{T}_1}/_{t_1}\} <: \mathsf{T}_2}{\Sigma \vdash \mu t_1.\mathsf{T}_1 <: \mathsf{T}_2}$$

(SR-RREC)
$$\frac{\Sigma, \mathsf{T}_1 <: \mu t_2.\mathsf{T}_2 \vdash \mathsf{T}_1 <: \mathsf{T}_2\{^{\mu t_2.\mathsf{T}_2}/_{t_2}\}}{\Sigma \vdash \mathsf{T}_1 <: \mu t_2.\mathsf{T}_2}$$

Definition 3.13 (Subterm). *The set of* subterms *of a recursive type T is defined as the least set satisfying the following equations:*

$$\begin{aligned}
\mathit{SubTerms}(\mathbf{base}) &= \{\mathbf{base}\} \\
\mathit{SubTerms}(\mathrm{R}\langle U_0 \rangle) &= \{\mathrm{R}\langle U_0 \rangle\} \cup \mathit{SubTerms}(U_0) \\
\mathit{SubTerms}(\mathrm{W}\langle T_0 \rangle) &= \{\mathrm{W}\langle T_0 \rangle\} \cup \mathit{SubTerms}(T_0) \\
\mathit{SubTerms}(\mathrm{RW}\langle U_0, T_0 \rangle) &= \{\mathrm{RW}\langle U_0, T_0 \rangle\} \cup \mathit{SubTerms}(U_0) \cup \mathit{SubTerms}(T_0) \\
\mathit{SubTerms}(u : A) &= \{u : A\} \cup \mathit{SubTerms}(A) \\
\mathit{SubTerms}(\mathrm{LOC}[u_i : A_i]) &= \{\mathrm{LOC}[u_i : A_i]\} \cup \bigcup_i \mathit{SubTerms}(A_i) \\
\mathit{SubTerms}(\mu \mathbf{Y}.K) &= \{\mu \mathbf{Y}.K\} \cup \mathit{SubTerms}(K\{^{\mu \mathbf{Y}.K}/_{\mathbf{Y}}\}) \\
\mathit{SubTerms}(A @ K) &= \{A @ K\} \cup \mathit{SubTerms}(A) \cup \mathit{SubTerms}(K) \\
\mathit{SubTerms}((\widetilde{C})) &= \{(\widetilde{C})\} \cup \bigcup_i \mathit{SubTerms}(C_i)
\end{aligned}$$

FIGURE 8 Meet inference rules

(MEET-AX)

$$\frac{}{\Sigma, \mathsf{T}_1 \sqcap \mathsf{T}_2 = \mathsf{T}_3 \vdash \mathsf{T}_1 \sqcap \mathsf{T}_2 = \mathsf{T}_3}$$

(MEET-TUPLE)

$$\frac{\Sigma \vdash \mathsf{C}_i \sqcap \mathsf{C}'_i = \mathsf{C}''_i}{\Sigma \vdash (\widetilde{\mathsf{C}}) \sqcap (\widetilde{\mathsf{C}'}) = (\widetilde{\mathsf{C}''})}$$

(MEET-BASE)

$$\frac{}{\Sigma \vdash \mathbf{base}_1 \sqcap \mathbf{base}_2 = \mathbf{base}_3} \; \mathbf{base}_1 = \mathbf{base}_2 = \mathbf{base}_3$$

(MEET-CAP)

$$\frac{\Sigma \vdash \mathsf{A}_1 \sqcap \mathsf{A}_2 = \mathsf{A}_3}{\Sigma \vdash u : \mathsf{A}_1 \sqcap u : \mathsf{A}_2 = u : \mathsf{A}_3}$$

(MEET-CHAN)

$$\frac{\Sigma \vdash \mathsf{U}_1 \sqcap \mathsf{U}_2 = \mathsf{U}_3 \quad \Sigma \vdash \mathsf{T}_1 \sqcup \mathsf{T}_2 = \mathsf{T}_3}{\Sigma \vdash \mathrm{RW}\langle \mathsf{U}_1, \mathsf{T}_1 \rangle \sqcap \mathrm{RW}\langle \mathsf{U}_2, \mathsf{T}_2 \rangle = \mathrm{RW}\langle \mathsf{U}_3, \mathsf{T}_3 \rangle} \; \mathsf{T}_3 <: \mathsf{U}_3$$

(MEET-HOM)

$$\frac{\Sigma \vdash \mathsf{A}_1 \sqcap \mathsf{A}_2 = \mathsf{A}_3 \quad \Sigma \vdash \mathsf{K}_1 \sqcap \mathsf{K}_2 = \mathsf{K}_3}{\Sigma \vdash \mathsf{A}_1 @ \mathsf{K}_1 \sqcap \mathsf{A}_2 @ \mathsf{K}_2 = \mathsf{A}_3 @ \mathsf{K}_3}$$

(MEET-LOC)

$$\frac{\Sigma \vdash \mathsf{U}_i \sqcap \mathsf{U}'_i = \mathsf{U}''_i}{\begin{array}{l}\Sigma \vdash \mathrm{LOC}[(u_i : \mathsf{U}_i)_i; (v_j : \mathsf{V}_j)_j] \sqcap \mathrm{LOC}[(u_i : \mathsf{U}'_i)_i; (w_k : \mathsf{W}_k)_k] \\ \quad = \mathrm{LOC}[(u_i : \mathsf{U}''_i)_i; (v_j : \mathsf{V}_j)_j; (w_k : \mathsf{W}_k)_k]\end{array}}$$

(MEET-REC-1)

$$\frac{\Sigma, \mu\mathbf{Y}.\mathsf{T}'_1 \sqcap \mathsf{T}_2 = \mathsf{T}_3 \vdash \mathsf{T}'_1\{^{\mu\mathbf{Y}.\mathsf{T}'_1}/_{\mathbf{Y}}\} \sqcap \mathsf{T}_2 = \mathsf{T}_3}{\Sigma \vdash \mu\mathbf{Y}.\mathsf{T}'_1 \sqcap \mathsf{T}_2 = \mathsf{T}_3}$$

(MEET-REC-2)

$$\frac{\Sigma, \mathsf{T}_1 \sqcap \mu\mathbf{Y}.\mathsf{T}'_2 = \mathsf{T}_3 \vdash \mathsf{T}_1 \sqcap \mathsf{T}'_2\{^{\mu\mathbf{Y}.\mathsf{T}'_2}/_{\mathbf{Y}}\} = \mathsf{T}_3}{\Sigma \vdash \mathsf{T}_1 \sqcap \mu\mathbf{Y}.\mathsf{T}'_2 = \mathsf{T}_3}$$

(MEET-REC-3)

$$\frac{\Sigma, \mathsf{T}_1 \sqcap \mathsf{T}_2 = \mu\mathbf{Y}.\mathsf{T}'_3 \vdash \mathsf{T}_1 \sqcap \mathsf{T}_2 = \mathsf{T}'_3\{^{\mu\mathbf{Y}.\mathsf{T}'_3}/_{\mathbf{Y}}\}}{\Sigma \vdash \mathsf{T}_1 \sqcap \mathsf{T}_2 = \mu\mathbf{Y}.\mathsf{T}'_3}$$

Note that the set of subterms of a given term is always finite even if the definition for the recursion operator is a simple unfolding, since every subsequent unfolding after the first one will not add any new term.

So to define subtyping on these terms, we keep all the rules we had in Figure 3, and we add a few rules for unfolding, which is adding terms to the memory, and axioms, when a given statement has already been

FIGURE 9 Join inference rules

(JOIN-AX)

$$\frac{}{\Sigma, \mathsf{T}_1 \sqcup \mathsf{T}_2 = \mathsf{T}_3 \vdash \mathsf{T}_1 \sqcup \mathsf{T}_2 = \mathsf{T}_3}$$

(JOIN-TUPLE)

$$\frac{\Sigma \vdash \mathsf{C}_i \sqcup \mathsf{C}'_i = \mathsf{C}''_i}{\Sigma \vdash (\widetilde{\mathsf{C}}) \sqcup (\widetilde{\mathsf{C}'}) = (\widetilde{\mathsf{C}''})}$$

(JOIN-BASE)

$$\frac{}{\Sigma \vdash \mathbf{base}_1 \sqcup \mathbf{base}_2 = \mathbf{base}_3} \ \mathbf{base}_1 = \mathbf{base}_2 = \mathbf{base}_3$$

(JOIN-CAP)

$$\frac{\Sigma \vdash \mathsf{A}_1 \sqcup \mathsf{A}_2 = \mathsf{A}_3}{\Sigma \vdash u : \mathsf{A}_1 \sqcup u : \mathsf{A}_2 = u : \mathsf{A}_3}$$

(JOIN-CHAN-RW-RW-R)

$$\frac{\Sigma \vdash \mathsf{U}_1 \sqcup \mathsf{U}_2 = \mathsf{U}_3}{\Sigma \vdash \mathrm{RW}\langle \mathsf{U}_1, \mathsf{T}_1 \rangle \sqcup \mathrm{RW}\langle \mathsf{U}_2, \mathsf{T}_2 \rangle = \mathrm{R}\langle \mathsf{U}_3 \rangle} \ \begin{array}{l} \mathsf{T}_1 <: \mathsf{U}_1 \quad \mathsf{T}_2 <: \mathsf{U}_2 \\ \mathsf{U}_1 \uparrow \mathsf{U}_2 \quad \mathsf{T}_1 \not\downarrow \mathsf{T}_2 \end{array}$$

(JOIN-CHAN-RW-RW-W)

$$\frac{\Sigma \vdash \mathsf{T}_1 \sqcap \mathsf{T}_2 = \mathsf{T}_3}{\Sigma \vdash \mathrm{RW}\langle \mathsf{U}_1, \mathsf{T}_1 \rangle \sqcup \mathrm{RW}\langle \mathsf{U}_2, \mathsf{T}_2 \rangle = \mathrm{W}\langle \mathsf{T}_3 \rangle} \ \begin{array}{l} \mathsf{T}_1 <: \mathsf{U}_1 \quad \mathsf{T}_2 <: \mathsf{U}_2 \\ \mathsf{U}_1 \not\uparrow \mathsf{U}_2 \quad \mathsf{T}_1 \downarrow \mathsf{T}_2 \end{array}$$

(JOIN-CHAN-RW-RW-RW)

$$\frac{\Sigma \vdash \mathsf{U}_1 \sqcup \mathsf{U}_2 = \mathsf{U}_3 \quad \Sigma \vdash \mathsf{T}_1 \sqcap \mathsf{T}_2 = \mathsf{T}_3}{\Sigma \vdash \mathrm{RW}\langle \mathsf{U}_1, \mathsf{T}_1 \rangle \sqcup \mathrm{RW}\langle \mathsf{U}_2, \mathsf{T}_2 \rangle = \mathrm{RW}\langle \mathsf{U}_3, \mathsf{T}_3 \rangle} \ \begin{array}{l} \mathsf{T}_1 <: \mathsf{U}_1 \quad \mathsf{T}_2 <: \mathsf{U}_2 \\ \mathsf{U}_1 \uparrow \mathsf{U}_2 \quad \mathsf{T}_1 \downarrow \mathsf{T}_2 \end{array}$$

(JOIN-HOM)

$$\frac{\Sigma \vdash \mathsf{A}_1 \sqcup \mathsf{A}_2 = \mathsf{A}_3 \quad \Sigma \vdash \mathsf{K}_1 \sqcup \mathsf{K}_2 = \mathsf{K}_3}{\Sigma \vdash \mathsf{A}_1 @ \mathsf{K}_1 \sqcup \mathsf{A}_2 @ \mathsf{K}_2 = \mathsf{A}_3 @ \mathsf{K}_3}$$

(JOIN-LOC)

$$\frac{\Sigma \vdash \mathsf{U}_i \sqcup \mathsf{U}'_i = \mathsf{U}''_i}{\begin{array}{l} \Sigma \vdash \mathrm{LOC}[(u_i : \mathsf{U}_i)_i; (u_{n+i} : \mathsf{U}_{n+i})_i; (v_j : \mathsf{V}_j)_j] \\ \quad \sqcup \mathrm{LOC}[(u_i : \mathsf{U}'_i)_i; (u_{n+i} : \mathsf{U}'_{n+i})_i; (w_k : \mathsf{W}_k)_k] = \mathrm{LOC}[(u_i : \mathsf{U}''_i)_i] \end{array}} \ \begin{array}{l} \mathsf{U}_i \uparrow \mathsf{U}'_i \\ \mathsf{U}_{n+i} \not\uparrow \mathsf{U}'_{n+i} \end{array}$$

(JOIN-REC-1)

$$\frac{\Sigma, \mu\mathbf{Y}.\mathsf{T}'_1 \sqcup \mathsf{T}_2 = \mathsf{T}_3 \vdash \mathsf{T}'_1\{^{\mu\mathbf{Y}.\mathsf{T}'_1}/_{\mathbf{Y}}\} \sqcup \mathsf{T}_2 = \mathsf{T}_3}{\Sigma \vdash \mu\mathbf{Y}.\mathsf{T}'_1 \sqcup \mathsf{T}_2 = \mathsf{T}_3}$$

(JOIN-REC-2)

$$\frac{\Sigma, \mathsf{T}_1 \sqcup \mu\mathbf{Y}.\mathsf{T}'_2 = \mathsf{T}_3 \vdash \mathsf{T}_1 \sqcup \mathsf{T}'_2\{^{\mu\mathbf{Y}.\mathsf{T}'_2}/_{\mathbf{Y}}\} = \mathsf{T}_3}{\Sigma \vdash \mathsf{T}_1 \sqcup \mu\mathbf{Y}.\mathsf{T}'_2 = \mathsf{T}_3}$$

(JOIN-REC-3)

$$\frac{\Sigma, \mathsf{T}_1 \sqcup \mathsf{T}_2 = \mu\mathbf{Y}.\mathsf{T}'_3 \vdash \mathsf{T}_1 \sqcup \mathsf{T}_2 = \mathsf{T}'_3\{^{\mu\mathbf{Y}.\mathsf{T}'_3}/_{\mathbf{Y}}\}}{\Sigma \vdash \mathsf{T}_1 \sqcup \mathsf{T}_2 = \mu\mathbf{Y}.\mathsf{T}'_3}$$

seen. The rules we obtain look like the ones in [AC93] in the DPI setting. We do exactly the same thing for the definition of $\sqcap$ and $\sqcup$ to put them also in the purely recursive setting in Figures 8 and 9. In those rules, the different missing cases of the definition of the operators on channel types are obtained as degenerate instances of the given rules.

Of course, now that we have given two sets of rules to define what should be the same relations, we need to formally prove that they coincide on their common domain, namely the recursive types. This is the role of the following two propositions.

Proposition 3.14 (Recursive subtyping is tree subtyping). *For any two types* T_1 *and* T_2, $\mathsf{Tree}(T_1) <: \mathsf{Tree}(T_2)$ *if and only if* $\emptyset \vdash T_1 <: T_2$.

Proposition 3.15 (Recursive type meet is infinite tree meet). *For any three types* T_1, T_2 *and* T_3, $\mathsf{Tree}(T_1) \sqcap \mathsf{Tree}(T_2) = \mathsf{Tree}(T_3)$ *if and only if* $\emptyset \vdash T_1 \sqcap T_2 = T_3$.

The proofs for these two propositions are really similar, as one would expect, so we give only the proof of the second one, slightly trickier.

Proof. Let us consider some types T_1, T_2 and T_3, and some Σ', a set of "hypotheses" of the forms $\mathsf{T}'_1 \sqcap \mathsf{T}'_2 = \mathsf{T}'_3$ and $\mathsf{T}'_1 \sqcup \mathsf{T}'_2 = \mathsf{T}'_3$, with T'_i a subterm of T_i. Let us prove that $\mathsf{Tree}(\mathsf{T}'_1) \sqcap \mathsf{Tree}(\mathsf{T}'_2) = \mathsf{Tree}(\mathsf{T}'_3)$ implies $\Sigma' \vdash \mathsf{T}'_1 \sqcap \mathsf{T}'_2 = \mathsf{T}'_3$ and its dual by reasoning on the forms of T'_i. More precisely, we proceed by induction on the lexicographic order $(|\mathsf{SubTerms}(\mathsf{T}_1)| \cdot |\mathsf{SubTerms}(\mathsf{T}_2)| \cdot |\mathsf{SubTerms}(\mathsf{T}_3)| - |\Sigma'|, |\mathsf{T}'_1| + |\mathsf{T}'_2| + |\mathsf{T}'_3|)$.

- If one of the T'_i is of the form $\mu\mathbf{Y}.\mathsf{T}$, and if $\mathsf{T}'_1 \sqcap \mathsf{T}'_2 = \mathsf{T}'_3$ is in Σ', we apply the axiom rule (MEET-AX). If there is no such statement in Σ', we apply rule (MEET-REC-i), which means that $|\mathsf{SubTerms}(\mathsf{T}_1)| \cdot |\mathsf{SubTerms}(\mathsf{T}_2)| \cdot |\mathsf{SubTerms}(\mathsf{T}_3)| - |\Sigma', \mathsf{T}'_1 \sqcap \mathsf{T}'_2 = \mathsf{T}'_3|$ is smaller, so we can use the induction hypothesis to finish.

- If none of the types T'_i is of the form $\mu\mathbf{Y}.\mathsf{T}$, then, by definition of $\mathsf{Tree}(\mathsf{T}'_i)$, T'_i and $\mathsf{Tree}(\mathsf{T}'_i)$ have the same head construct. Each case of the definition of MeetJoin corresponds to one rule in Figures 8 and 9. We consider only one typical case.

  - $\Sigma' \vdash \mathrm{R}\langle \mathsf{T}''_1 \rangle \sqcap \mathrm{R}\langle \mathsf{T}''_2 \rangle = \mathrm{R}\langle \mathsf{T}''_3 \rangle$. We know that $\mathsf{Tree}(\mathsf{T}''_1) \sqcap \mathsf{Tree}(\mathsf{T}''_2) = \mathrm{R}\langle \mathsf{T}''_3 \rangle$ by definition of MeetJoin. So we can use our induction hypothesis on T''_i to get $\Sigma' \vdash \mathsf{T}''_1 \sqcap \mathsf{T}''_2 = \mathsf{T}''_3$, which entails that $\Sigma' \vdash \mathrm{R}\langle \mathsf{T}''_1 \rangle \sqcap \mathrm{R}\langle \mathsf{T}''_2 \rangle = \mathrm{R}\langle \mathsf{T}''_3 \rangle$ by rule (MEET-CHAN).

Conversely, let us consider some proof of $\emptyset \vdash \mathsf{T}_1 \sqcap \mathsf{T}_2 = \mathsf{T}_3$. We define the relation

$$\begin{aligned}\mathcal{R} = \{&{}_{\sqcap}(\mathsf{Tree}(\mathsf{T}'_1), \mathsf{Tree}(\mathsf{T}'_2), \mathsf{Tree}(\mathsf{T}'_3)) \mid \exists \Sigma' \text{ such that} \\ &\Sigma' \vdash \mathsf{T}'_1 \sqcap \mathsf{T}'_2 = \mathsf{T}'_3 \text{ appears in the proof of } \emptyset \vdash \mathsf{T}_1 \sqcap \mathsf{T}_2 = \mathsf{T}_3\} \\ \cup \{&{}_{\sqcup}(\mathsf{Tree}(\mathsf{T}'_1), \mathsf{Tree}(\mathsf{T}'_2), \mathsf{Tree}(\mathsf{T}'_3)) \mid \exists \Sigma' \text{ such that} \\ &\Sigma' \vdash \mathsf{T}'_1 \sqcup \mathsf{T}'_2 = \mathsf{T}'_3 \text{ appears in the proof of } \emptyset \vdash \mathsf{T}_1 \sqcap \mathsf{T}_2 = \mathsf{T}_3\}\end{aligned}$$

Let us prove that this relation $\mathcal{R}$ is a postfixpoint of $\mathsf{MeetJoin}$. We consider a triple in $\mathcal{R}$ and we reason on the last rule used to reach the corresponding statement in the proof of $\emptyset \vdash \mathsf{T}_1 \sqcap \mathsf{T}_2 = \mathsf{T}_3$.

- (MEET-AX). Then we know that $\mathsf{T}'_1 \sqcap \mathsf{T}'_2 = \mathsf{T}'_3$ can have been introduced in Σ' only by a rule (MEET-REC-i) higher in that branch of the proof. Since our types are contractive, we then know that T'_i must be of the form $\mu\mathbf{Y}_1.\mu\mathbf{Y}_2 \ldots \text{LOC}[\ldots]$. So there must be a (MEET-LOC) corresponding to that LOC[...] in the proof under the different (MEET-REC-j)s. We write $\Sigma'' \vdash \mathsf{T}''_1 \sqcap \mathsf{T}''_2 = \mathsf{T}''_3$ the conclusion statement of that (MEET-LOC). By definition of the function $\mathsf{Tree}(\mathsf{T}''_i) = \mathsf{Tree}(\mathsf{T}'_i)$. This means that we can proceed as in case (MEET-LOC).
- (MEET-LOC). Then we have a proof of $\Sigma' \vdash \mathsf{U}_{1_i}' \sqcap \mathsf{U}_{2_i}' = \mathsf{U}_{3_i}'$ which means that every triple ${}_{\sqcap}(\mathsf{Tree}(\mathsf{U}_{1_i}'), \mathsf{Tree}(\mathsf{U}_{2_i}'), \mathsf{Tree}(\mathsf{U}_{3_i}'))$ are in $\mathcal{R}$ which proves that ${}_{\sqcap}(\mathsf{Tree}(\mathsf{T}'_1), \mathsf{Tree}(\mathsf{T}'_2), \mathsf{Tree}(\mathsf{T}'_3))$ is in $\mathsf{MeetJoin}(\mathcal{R})$.
- (MEET-REC-1), with $\mathsf{T}'_1 = \mu\mathbf{Y}.\mathsf{T}''_1$. By definition of the function $\mathsf{Tree}(\cdot)$, $\mathsf{Tree}(\mathsf{T}'_1) = \mathsf{Tree}(\mathsf{T}''_1\{^{\mu\mathbf{Y}.\mathsf{T}''_1}/_{\mathbf{Y}}\})$. As in the case (MEET-AX) we proceed until we reach a (MEET-LOC) and apply the same argument as for (MEET-LOC).

□

Even if this establishes an equivalence between the coinductive and the inductive versions of the system for $\sqcap$ and $<:$, we feel that manipulation of the coinductive types is easier because it is more intuitive: the intuitions coming from induction on non-recursive types can guide the proof in the coinductive setting.

4 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

$$\Gamma \vdash M$$

Figure 10 Well-formed environments

$$\frac{}{\vdash \mathbf{env}}\ (\text{E-EMPTY})$$

$$\frac{\Gamma \vdash \mathbf{env}}{\Gamma, u : \mathbf{base} \vdash \mathbf{env}}\ \Gamma(u) \downarrow \mathbf{base}\ (\text{E-BASE})$$

$$\frac{\Gamma \vdash \mathbf{env} \quad \Gamma \vdash w : \text{LOC} \quad \Gamma(u@w) = \{\mathsf{A}_i\}}{\Gamma, u@w : \mathsf{A} \vdash \mathbf{env}}\ \{\mathsf{A}_i\} \downarrow \mathsf{A}\ (\text{E-NEW-LCHAN})$$

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

$$\frac{\Gamma \vdash \mathbf{env}}{\Gamma, Z : \text{LOC}[(u_i : \mathsf{A}_i)] \vdash \mathbf{env}}\ Z \notin \Gamma\ (\text{E-REC})$$

$$\frac{\Gamma \vdash \mathbf{env} \quad \Gamma(Z) = \text{LOC}[\ldots, u : \mathsf{A}, \ldots]}{\Gamma, u@Z : \mathsf{A} \vdash \mathbf{env}}\ (\text{E-DEC-AT-REC})$$

Figure 11 Typing values

$$\frac{\Gamma, u : \mathsf{T}, \Gamma' \vdash \mathbf{env}}{\Gamma, u : \mathsf{T}, \Gamma' \vdash u : \mathsf{T}'}\ \mathsf{T} <: \mathsf{T}'\ (\text{V-NAME})$$

$$\frac{\Gamma \vdash u : \mathsf{T} \quad \Gamma \vdash w : \text{LOC}}{\Gamma \vdash_w u : \mathsf{T}}\ (\text{V-LOCATED})$$

$$\frac{\Gamma, u@w : \mathsf{A}, \Gamma' \vdash w : \text{LOC}}{\Gamma, u@w : \mathsf{A}, \Gamma' \vdash_w u : \mathsf{A}'}\ \mathsf{A} <: \mathsf{A}'\ (\text{V-CHANNEL})$$

$$\frac{\Gamma \vdash_w u : \mathsf{T}_1 \quad \Gamma \vdash_w u : \mathsf{T}_2}{\Gamma \vdash_w u : \mathsf{T}_1 \sqcap \mathsf{T}_2}\ (\text{V-MEET})$$

$$\frac{\Gamma \vdash_w u_i : \mathsf{T}_i}{\Gamma \vdash_w (\widetilde{u}) : (\widetilde{\mathsf{T}})}\ (\text{V-TUPLE})$$

$$\frac{\Gamma \vdash w : \text{LOC}}{\Gamma \vdash_w u : \mathbf{base}}\ u \in \mathbf{base}\ (\text{V-BASE})$$

$$\frac{\Gamma \vdash_v u_i : \mathsf{A}_i \quad \Gamma \vdash_w v : \mathsf{K}}{\Gamma \vdash_w (\widetilde{u})@v : (\widetilde{\mathsf{A}})@\mathsf{K}}\ (\text{V-LOCATED-CHANNEL})$$

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

and the rules used are identical to those for Dpi; see Figure 12. These rules are based on the definition of well-formed environments to infer judgements of the form $\Gamma \vdash \mathbf{env}$, which is given in Figure 10. The notion of well-formedness is, again, inherited from Dpi. Environments are lists of elements of the form $u : \mathsf{T}$ with u a name or a variable (includ-

FIGURE 12 Typing Systems

$$\frac{\Gamma, c@k : \mathsf{C} \vdash M}{\Gamma \vdash (\mathsf{new}\, c@k : \mathsf{C})\, M}\ (\text{T-CNEW})$$

$$\frac{\Gamma \vdash \mathbf{env}}{\Gamma \vdash \mathbf{0}}\ (\text{T-NIL}) \qquad \frac{\Gamma \vdash M \quad \Gamma \vdash N}{\Gamma \vdash M \mid N}\ (\text{T-PAR})$$

$$\frac{\Gamma \vdash_k P}{\Gamma \vdash k[\![P]\!]}\ (\text{T-PROC}) \qquad \frac{\Gamma, \{k : \mathsf{K}\} \vdash M}{\Gamma \vdash (\mathsf{new}\, k : \mathsf{K})\, M}\ (\text{T-LNEW})$$

ing a recursion variable) and T its type or of the form $u@w : \mathsf{A}$ with u a name or a variable standing for a channel and A its type. A given u can appear more than once in that list as long as the types at which it is known in a given location are compatible. This is useful for the names received during communications: if you get some name at two different types (through communication on two different channels), you can simply consider the environment in which that name is given those two types. Of course, the typing rules will ensure that this situation will arise only when the channel types are indeed compatible. Note that the formation rules for environments allow the presence of recursion variables with their declaration types, because that information is needed for typing processes, as will be explained shortly. We will also need some subtyping on environments. We say that $\Gamma <: \Gamma'$ when

- $\mathsf{dom}(\Gamma') \subseteq \mathsf{dom}(\Gamma)$;
- for all recursion variable Z defined in Γ', we have $\Gamma(Z) = \Gamma'(Z)$;
- for all name or variable x defined in Γ', we have $\bigsqcap \Gamma(x) <: \bigsqcap \Gamma'(x)$, x being possibly of the form $u@w$.

The other basic typing rules needed for typing systems and processes are for values, see Figure 11. Those are directly taken from DPI, without modifications. Note though that recursion variables can be typed exactly as locations in this setting.

To come back to the system typing, the main rule in Figure 12 is

$$\frac{\Gamma \vdash_k P}{\Gamma \vdash k[\![P]\!]}\ (\text{T-PROC})$$

FIGURE 13 recDPI processes typing system

(T-OUTPUT)
$$\frac{\Gamma \vdash_w u : \mathrm{W}\langle \mathsf{T} \rangle \quad \Gamma \vdash_w V : \mathsf{T} \quad \Gamma \vdash_w P}{\Gamma \vdash_w u\,!\langle V \rangle\, P}$$

(T-INPUT)
$$\frac{\Gamma \vdash_w u : \mathrm{R}\langle \mathsf{T} \rangle \quad \Gamma, \langle X : \mathsf{T} \rangle \,@w \vdash_w P}{\Gamma \vdash_w u\,?(X : \mathsf{T})\, P}$$

(T-GO)
$$\frac{\Gamma \vdash_m P}{\Gamma \vdash_w \mathsf{goto}\, m.P}$$

(T-STOP)
$$\frac{\Gamma \vdash \mathbf{env}}{\Gamma \vdash_w \mathsf{stop}}$$

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

(T-RECVAR)
$$\frac{\Gamma \vdash w : \Gamma(Z)}{\Gamma \vdash_w Z}\ \Gamma(Z) = \mathrm{LOC}[\ldots]$$

(T-MATCH)
$$\frac{\begin{array}{l}\Gamma \vdash_w u : \mathsf{U}, v : \mathsf{V} \quad \Gamma \vdash_w Q \\ \text{If } \Gamma, \langle u : \mathsf{V} \rangle \,@w, \langle v : \mathsf{U} \rangle \,@w \vdash \mathbf{env}, \\ \quad \Gamma, \langle u : \mathsf{V} \rangle \,@w, \langle v : \mathsf{U} \rangle \,@w \vdash_w P\end{array}}{\Gamma \vdash_w \mathsf{if}\ u = v\ \mathsf{then}\ P\ \mathsf{else}\ Q}$$

(T-L-NEW)
$$\frac{\Gamma, \langle k : \mathsf{K} \rangle \vdash_w P}{\Gamma \vdash_w (\mathsf{newloc}\, k : \mathsf{K})\, P}$$

(T-C-NEW)
$$\frac{\Gamma, n @ w : \mathsf{A} \vdash_w P}{\Gamma \vdash_w (\mathsf{newc}\, n : \mathsf{A})\, P}$$

(T-HERE)
$$\frac{\Gamma \vdash_w P[{}^{w}/_{x}]}{\Gamma \vdash_w \mathsf{here}\, [x]\, P}$$

(T-REP)
$$\frac{\Gamma \vdash_w P}{\Gamma \vdash_w * P}$$

(T-PAR)
$$\frac{\Gamma \vdash_w P \quad \Gamma \vdash_w Q}{\Gamma \vdash_w P \mid Q}$$

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 13, and we concentrate here on explaining the three new rules required for recursion

and the here construct. The latter is straightforward:

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

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

$$\Gamma \vdash_k \mathsf{rec}\ (Z : \mathsf{R}).\ P \tag{3}$$

we will need the entries for recursion variables. Recall that here the type R is a location type, such as $\textsc{loc}[u_1 : \mathsf{A}_1, \ldots u_n : \mathsf{A}_n]$, 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 we need to know exactly the type at which the recursion variable is declared when we are typechecking the recursive calls. So, unlike locations, we have in the environment entries of the form $Z : \textsc{loc}[u_i : \mathsf{A}_i]$ and not merely $Z : \textsc{loc}$ since subtyping can not be allowed on the declared type for these variables. Therefore we only allow unique entries for a given recursion variable in a type environment. Then the natural rule for typing-checking a recursive call, that is an occurrence of a recursion variable, is given by:

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

In order to typecheck a recursive definition, such as (3) 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

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

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:

$$\langle\!\langle Z : \textsc{loc}[u_1 : \mathsf{A}_1, \ldots] \rangle\!\rangle = Z : \textsc{loc}[u_i : \mathsf{A}_i], u_1 @ Z : \mathsf{A}_1, u_2 @ Z : \mathsf{A}_2, \ldots$$

Notice that, to type P at Z, we will really have to consider Z as a value from the type point of view, but this will be only an artefact of the way

typing proceeds. Z will never be a value in real terms, this being syntactically prohibited.

Example 4.1. 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:

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

which will start by proving

$$\Gamma, \langle\!\langle Z : \mathsf{S} \rangle\!\rangle \vdash_Z \mathit{test} : \mathrm{R}\langle \mathsf{T}_t \rangle$$

so, by expanding the notation $\langle\!\langle Z : \mathsf{S} \rangle\!\rangle$ with

$$\mathsf{S} = \mu \mathbf{Y}.\mathrm{LOC}[\mathit{test} : \mathrm{R}\langle \mathsf{T}_t \rangle,\ \ \mathit{neigh} : \mathrm{R}\langle \mathbf{Y} \rangle]$$

we get

$$\Gamma, Z : \mathsf{S}, \mathit{test}@Z : \mathrm{R}\langle \mathsf{T}_t \rangle, \mathit{neigh}@Z : \mathrm{R}\langle \mathsf{S} \rangle \vdash_Z \mathit{test} : \mathrm{R}\langle \mathsf{T}_t \rangle$$

where we can see that it is simply an axiom. The other judgement to prove then is

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

which will amount to proving

$$\Gamma, \langle\!\langle Z : \mathsf{S} \rangle\!\rangle \vdash_Z \mathsf{goto\ home}.\mathit{report}\,!\langle x \rangle$$

and

$$\Gamma, \langle\!\langle Z : \mathsf{S} \rangle\!\rangle \vdash_Z \mathit{neigh}\,?(y)\ \mathsf{goto}\ y.Z$$

where this second statement is particularly interesting here. In fact, this turns out as simply:

$$\Gamma, \langle\!\langle Z : \mathsf{S} \rangle\!\rangle, y : \mathsf{S} \vdash_y Z$$

which is obtained directly because y has type S, exactly what is required to "run" Z. ■

The partial view of recursion variables as locations complicates somewhat the formal rules for the construction of valid environments. We do not go into further details here, we refer the reader to [HMR03] for more complete explanations. But we give the formation rules in Figure 10, together with these for value typing, in Figure 11. 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 4.1 (Recursion Variable Substitution). *Suppose that* $\Gamma \vdash_w \mathsf{rec}\ Z : \mathsf{R}.\ P$. *Then* $\Gamma \vdash_w P\{^{\mathsf{rec}\ Z:\mathsf{R}.\ P}/_Z\}$.

Proof. This is done by induction on the proof that P is well-typed. So we generalise the property we prove into: for any location or recursion variable v and for any environment Γ if we have $\Gamma \vdash_v P$ and if for any Γ' and w such that $\Gamma' <: \Gamma$ and $\Gamma' \vdash w : \Gamma'(Z)$ we have $\Gamma' \vdash_w Q$ then $\Gamma \vdash_v P\{^Q/_Z\}$.

- (T-RECVAR) so $P = Z$ and we know $\Gamma \vdash v : \Gamma(Z)$. By hypothesis that implies that $\Gamma \vdash_v Q = P\{^Q/_Z\}$.
- (T-OUTPUT) so $P = u\,!\langle V\rangle\,P'$. This implies that $\Gamma \vdash_v P'$, on which we can apply the induction hypothesis. Therefore we have
$$\Gamma \vdash_v u\,!\langle V\rangle\,(P'\{^Q/_Z\})$$
which is exactly $\Gamma \vdash_v P$.
- (T-INPUT) so $P = u\,?(X : \mathsf{T})\,P'$ and $\Gamma,\langle X : \mathsf{T}\rangle_{@v} \vdash_v P'$. By weakening we know that, for any w such that $\Gamma,\langle X : \mathsf{T}\rangle_{@v} \vdash w : (\Gamma,\langle X : \mathsf{T}\rangle_{@v})(Z)$, $\Gamma,\langle X : \mathsf{T}\rangle_{@v} \vdash_w Q$.
- (T-MATCH) which implies that $P = \mathsf{if}\ u = u'\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2$ and that $\Gamma \vdash_v u : \mathsf{U}, u' : \mathsf{U}'$, $\Gamma \vdash_v P_2$ and, if $\Gamma,\langle u : \mathsf{U}'\rangle_{@v},\langle u' : \mathsf{U}\rangle_{@v} \vdash \mathbf{env}$, $\Gamma,\langle u : \mathsf{U}'\rangle_{@v},\langle u' : \mathsf{U}\rangle_{@v} \vdash_v P_1$. Then, by our induction hypothesis, we know that $\Gamma \vdash_v P_2\{^Q/_Z\}$. And since $\Gamma,\langle u : \mathsf{U}'\rangle_{@v},\langle u' : \mathsf{U}\rangle_{@v} <: \Gamma$ then $\Gamma,\langle u : \mathsf{U}'\rangle_{@v},\langle u' : \mathsf{U}\rangle_{@v} \vdash_v P_1\{^Q/_Z\}$.
- (T-HERE) so $P = \mathsf{here}\,[x]\,P'$ and $\Gamma \vdash_v P'[^v/_x]$. By our induction hypothesis we have $\Gamma \vdash_v P'[^v/_x]\{^Q/_Z\}$ and $P'\{^Q/_Z\}[^v/_x] = P'\{^Q/_Z\}[^v/_x]$ since the two substitutions do not deal with the same objects (recursion variables as terms and location variables). So applying (T-HERE) again gives $\Gamma \vdash_v (\mathsf{here}\,[x]\,P')\{^Q/_Z\}$.
- (T-REC) so $P = \mathsf{rec}\ Z' : \mathsf{R}'.\ P'$ with $\Gamma,\langle\!\langle Z' : \mathsf{R}'\rangle\!\rangle \vdash_{Z'} P'$. Since $\Gamma,\langle\!\langle Z' : \mathsf{R}'\rangle\!\rangle$ is a subtype-environment of Γ we can apply our induction hypothesis on it to get $\Gamma,\langle\!\langle Z' : \mathsf{R}'\rangle\!\rangle \vdash_{Z'} P'\{^Q/_Z\}$ which implies that $\Gamma \vdash_v P\{^Q/_Z\}$.

Now we must prove that what we just proved indeed applies to processes of the form $\mathsf{rec}\ Z : \mathsf{R}.\ P$. We know that $\Gamma \vdash_w \mathsf{rec}\ Z : \mathsf{R}.\ P$. This implies that $\Gamma,\langle\!\langle Z : \mathsf{R}\rangle\!\rangle \vdash_Z P$. By weakening, we obtain that, for any Γ'

such that $\Gamma' <: \Gamma$, $\Gamma', \langle\!\langle Z : \mathsf{R} \rangle\!\rangle \vdash_Z P$. So, for any location v such that $\Gamma' \vdash v : (\Gamma', \langle\!\langle Z : \mathsf{R} \rangle\!\rangle)(Z) = \mathsf{R}$, we have $\Gamma' \vdash_v \mathsf{rec}\ Z : \mathsf{R}.\ P$.

So we can use $\mathsf{rec}\ Z : \mathsf{R}.\ P$ as a "Q" in the previous proof and then conclude. □

This in turn leads to:

Theorem 4.2 (Subject Reduction). *$\Gamma \vdash M$ and $M \xrightarrow{\tau} M'$ implies that $\Gamma \vdash M'$.*

Proof. This proof heavily relies on the preexisting proof of subject reduction in DPI. We simply added two derivation rules (LTS-HERE) and (LTS-REC) so we just have to deal with those two.

- $M = k[\![\mathsf{here}\,[x]\,P]\!]$ and $M' = k[\![P[^k/_x]]\!]$. The result is direct since the only rule to prove that $\Gamma \vdash_k \mathsf{here}\,[x]\,P$ assumes that $\Gamma \vdash_k P[^k/_x]$.
- $M = k[\![\mathsf{rec}\ Z : \mathsf{R}.\ P]\!]$ and $M' = k[\![P\{^{\mathsf{rec}\ Z:\mathsf{R}.\ P}/_Z\}]\!]$. By the previous lemma $\Gamma \vdash_k \mathsf{rec}\ Z : \mathsf{R}.\ P$ implies that $\Gamma \vdash_k P\{^{\mathsf{rec}\ Z:\mathsf{R}.\ P}/_Z\}$. That proves that $\Gamma \vdash M'$.

□

5 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

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

where

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

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.

- $\textsc{unrec}(\mathsf{rec}\ Z : \mathsf{R}.\ P) = (\mathsf{newloc}\, home_Z : \textsc{loc}[ping_Z : \textsc{rw}\langle \mathsf{R} \rangle])$
  $(\textsc{unrec}(Z) \mid$
  $\mathsf{goto}\, home_Z.$
  $*\, ping_Z\,?(l : \mathsf{R})\, \mathsf{goto}\, l.\textsc{unrec}(P))$
- $\textsc{unrec}(Z) = \mathsf{here}\,[x]\, \mathsf{goto}\, home_Z.ping_Z\,!\langle x \rangle$
- $\textsc{unrec}(u\,!\langle V \rangle\ P) = u\,!\langle V \rangle\ \textsc{unrec}(P)$; all the other cases are similar.

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 [FH05], 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 [SW01] 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.

So the translation of a system would start by identifying the set of recursion variables: let us write this set $\{Z_i\}$, and their corresponding processes

$\{P_i\}$ when "$\mathsf{rec}\ Z_i : \mathsf{R}_i.\ P_i$" appear in the system. For any process P_i among those we will note $\tilde{n}_i$ its set of free names. Then the components of the system are simply translated the following way:

- $\text{NC-UNREC}(Z_i) = \mathsf{here}\,[x]\,\mathsf{goto}\,home_{Z_i}.ping_{Z_i}\,!\langle x, \tilde{n}_i\rangle$
- $\text{NC-UNREC}(\mathsf{rec}\ Z_i : \mathsf{R}_i.\ P_i) = \text{NC-UNREC}(Z_i)$
- $\text{NC-UNREC}(u\,!\langle V\rangle\,P) = u\,!\langle V\rangle\,\text{NC-UNREC}(P)$; all the other cases are similar.

A system M is then translated, as a whole, into the following process:

$$\begin{aligned}&(\mathsf{new}\ ping_{Z_1})\,(\mathsf{new}\ home_{Z_1})\,(\mathsf{new}\ ping_{Z_2})\,(\mathsf{new}\ home_{Z_2})\ldots\\ &\quad home_{Z_1}[\![*\ ping_{Z_1}\ ?(l : \mathsf{R}_1, \tilde{n}_1)\,\mathsf{goto}\,l.\text{NC-UNREC}(P_1)]\!]\ |\\ &\quad home_{Z_2}[\![*\ ping_{Z_2}\ ?(l : \mathsf{R}_2, \tilde{n}_2)\,\mathsf{goto}\,l.\text{NC-UNREC}(P_2)]\!]\ |\ldots|\\ &\quad \text{NC-UNREC}(M)\end{aligned}$$

But, of course, such an approach would not be compositional, as the name NC-UNREC$(\cdot)$ suggests.

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 5.1. *$\Gamma \vdash M$ if and only if $\Gamma \vdash \text{UNREC}(M)$*

Proof. We define the function φ over environments:

$$\begin{aligned}&\varphi(\Gamma, \langle\!\langle Z_i : \mathsf{R}_i\rangle\!\rangle, u_{i_j}@Z_i : \mathsf{A}_{i_j}) =\\ &\quad \Gamma, home_{Z_i} : \text{LOC}, ping_{Z_i}@home_{Z_i} : \text{RW}\langle \mathsf{R}_i\rangle, \langle l_i : \mathsf{R}_i\rangle, u_{i_j}@l_i : \mathsf{A}_{i_j}\end{aligned}$$

φ^{-1} is defined as expected.

We now prove the following generalised statement:

- $\Gamma \vdash M$ implies $\varphi(\Gamma) \vdash \text{UNREC}(M)$;
- $\Gamma \vdash_v P$ implies that $\varphi(\Gamma) \vdash_v \text{UNREC}(P)$;
- $\Gamma \vdash_{Z_i} P$ implies that $\varphi(\Gamma) \vdash_{l_i} \text{UNREC}(P)$.

To get there, we first need to prove the equivalent property for value typing:

- $\Gamma \vdash Z_i : \mathsf{K}$ implies that $\varphi(\Gamma) \vdash l_i : \mathsf{K}$;
- $\Gamma \vdash u@Z_i : \mathsf{A}$ implies that $\varphi(\Gamma) \vdash u@l_i : \mathsf{A}$;
- $\Gamma \vdash u : \mathsf{T}$ other than the previous cases implies that $\varphi(\Gamma) \vdash u : \mathsf{T}$.
- Tuples components enter inductively in some of those cases.

We reason on the proof of $\Gamma \vdash \ldots$ for this result.

- (V-MEET): we get the result by induction.
- (V-NAME): if the conclusion is $\Gamma \vdash Z_i : \mathsf{K}$, we know that $\varphi(\Gamma)$ contains $\langle l_i : \mathsf{R}_i \rangle$ with $\Gamma(Z_i) = \mathsf{R}_i$. So $\mathsf{R}_i <: \mathsf{K}$ implies that (V-LOC) gives $\varphi(\Gamma) \vdash l_i : \mathsf{K}$.
- Other cases simply involve induction.

Then we can reason on the proof of $\Gamma \vdash M$, etc.

- (T-PROC): $\Gamma \vdash k[\![P]\!]$ implies that $\Gamma \vdash_k P$ so our induction hypothesis gives $\varphi(\Gamma) \vdash_k \text{UNREC}(P)$ which entails $\varphi(\Gamma) \vdash \text{UNREC}(M)$.
- (T-REC): we first assume that the statement is $\Gamma \vdash_w \mathsf{rec}\ Z : \mathsf{R}.\ P$, with w a real location. We know that $\Gamma, \langle\!\langle Z : \mathsf{R} \rangle\!\rangle \vdash_z P$ and $\Gamma \vdash w : \mathsf{R}$. This means that, by induction, we have: $\varphi(\Gamma, \langle\!\langle Z : \mathsf{R} \rangle\!\rangle) \vdash_l \text{UNREC}(P)$, with $\varphi(\Gamma, \langle\!\langle Z : \mathsf{R} \rangle\!\rangle) = \varphi(\Gamma), \mathit{home}_Z : \text{LOC}, \mathit{ping}_Z @ \mathit{home}_Z : \text{RW}\langle \mathsf{R}_i \rangle, \langle l : \mathsf{R} \rangle$. We can then proceed with:
  - $\varphi(\Gamma), \mathit{home}_Z : \text{LOC}, \mathit{ping}_Z @ \mathit{home}_Z : \text{RW}\langle \mathsf{R} \rangle, \langle l : \mathsf{R} \rangle \vdash_{\mathit{home}_Z} \mathsf{goto}\, l.\text{UNREC}(P)$
  - $\varphi(\Gamma), \mathit{home}_Z : \text{LOC}, \mathit{ping}_Z @ \mathit{home}_Z : \text{RW}\langle \mathsf{R} \rangle \vdash_{\mathit{home}_Z} \mathit{ping}_Z\, ?(l : \mathsf{R})\, \mathsf{goto}\, l.\text{UNREC}(P)$
  - $\varphi(\Gamma), \mathit{home}_Z : \text{LOC}, \mathit{ping}_Z @ \mathit{home}_Z : \text{RW}\langle \mathsf{R} \rangle \vdash_{\mathit{home}_Z} * \mathit{ping}_Z\, ?(l : \mathsf{R})\, \mathsf{goto}\, l.\text{UNREC}(P)$
  - $\varphi(\Gamma), \mathit{home}_Z : \text{LOC}, \mathit{ping}_Z @ \mathit{home}_Z : \text{RW}\langle \mathsf{R} \rangle \vdash_w \mathsf{goto}\, \mathit{home}_Z.\, *\ \mathit{ping}_Z\, ?(l : \mathsf{R})\, \mathsf{goto}\, l.\text{UNREC}(P)$

  On the other side, $\Gamma \vdash w : \mathsf{R}$ implies that $\varphi(\Gamma) \vdash w : \mathsf{R}$.
  - $\varphi(\Gamma), \mathit{home}_Z : \text{LOC}, \mathit{ping}_Z @ \mathit{home}_Z : \text{RW}\langle \mathsf{R} \rangle \vdash_w \text{W}\langle \mathit{ping}_Z \rangle[w]$
  - $\varphi(\Gamma), \mathit{home}_Z : \text{LOC}, \mathit{ping}_Z @ \mathit{home}_Z : \text{RW}\langle \mathsf{R} \rangle \vdash_w \mathsf{goto}\, \mathit{home}_Z.\text{W}\langle \mathit{ping}_Z \rangle[w]$
  - $\varphi(\Gamma), \mathit{home}_Z : \text{LOC}, \mathit{ping}_Z @ \mathit{home}_Z : \text{RW}\langle \mathsf{R} \rangle \vdash_w \mathsf{here}\, [x]\, \mathsf{goto}\, \mathit{home}_Z.\text{W}\langle \mathit{ping}_Z \rangle[x]$

  So we can join the two to get:
  - $\varphi(\Gamma), \mathit{home}_Z : \text{LOC}, \mathit{ping}_Z @ \mathit{home}_Z : \text{RW}\langle \mathsf{R} \rangle \vdash_w \mathsf{here}\, [x]\, \mathsf{goto}\, \mathit{home}_Z.\text{W}\langle \mathit{ping}_Z \rangle[x] \mid \mathsf{goto}\, \mathit{home}_Z.\, *\ \mathit{ping}_Z\, ?(l : \mathsf{R})\, \mathsf{goto}\, l.\text{UNREC}(P)$
  - $\varphi(\Gamma) \vdash_w \text{UNREC}(\mathsf{rec}\ Z : \mathsf{R}.\ P)$.

The reasoning would be identical for $\Gamma \vdash_{Z_j} \mathsf{rec}\ Z_i : \mathsf{R}_i.\ P$ but w would have to be replaced by Z_j when typing in Γ and by l_j when in $\varphi(\Gamma)$.

- (T-RECVAR): $\Gamma \vdash_w Z$ implies that $\Gamma \vdash w : \Gamma(Z)$. Then $\varphi(\Gamma) \vdash w : \Gamma(Z)$, which implies that $\varphi(\Gamma) \vdash_{home_Z} \mathrm{W}\langle ping_Z \rangle[w]$ since Z must be in $\mathsf{dom}(\Gamma)$. So

$$\varphi(\Gamma) \vdash_w \mathsf{here}\,[x]\,\mathsf{goto}\,home_Z.\mathrm{W}\langle ping_Z \rangle[w]$$

  which is $\varphi(\Gamma) \vdash_w \textsc{unrec}(Z)$. Again the reasoning would be identical in a Z_j location.

- (T-INPUT): we have $\Gamma \vdash_{Z_i} \mathrm{R}\langle u \rangle[X : \mathsf{T}]P$, $\Gamma \vdash u @ Z_i : \mathrm{R}\langle \mathsf{T} \rangle$ and $\Gamma, \langle X : \mathsf{T} \rangle @ Z_i \vdash_{Z_i} P$. By induction we get $\varphi(\Gamma) \vdash u @ l_i : \mathrm{R}\langle \mathsf{T} \rangle$ and $\varphi(\Gamma, \langle X : \mathsf{T} \rangle @ Z_i) \vdash_{l_i} \textsc{unrec}(P)$, with $\varphi(\Gamma, \langle X : \mathsf{T} \rangle @ Z_i)$ being the environment $\varphi(\Gamma), \langle X : \mathsf{T} \rangle @ l_i$. So an application of (T-INPUT) gives $\varphi(\Gamma) \vdash \mathrm{R}\langle u \rangle[X : \mathsf{T}]\textsc{unrec}(P)$. The other case, with a real location w would be similar.

- The other cases are similar.

□

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 [HMR03] as a typed version of *reduction barbed congruence*, giving rise to the judgements

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

To emphasise that, in those judgements, the mentioned environment is an observer's knowledge of the system, and therefore that it might not be possible to type the system in that environment, we will write this environment Ω and consider judgements of the form

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

More generally, equivalences over systems are considered within a given knowledge of the observer. So the objects we handle are composed of an environment and a system at the same time. For this we define the notion of configuration which requires subtyping over environments.

Definition 5.2 (Environment subtyping). *We say that an environment Γ is a subtype of another one Γ', written $\Gamma <: \Gamma'$, whenever*

- $\mathsf{dom}(\Gamma') \subseteq \mathsf{dom}(\Gamma)$;

FIGURE 14 Labelled transition semantics. Internal actions.

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

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

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

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

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

(LTS-L-CREATE)
$$\Omega \rhd k[\![(\mathsf{newloc}\, l : \mathsf{L})\, P]\!] \xrightarrow{\tau}_\beta \Omega \rhd (\mathsf{new}\, l : \mathsf{L})\, k[\![P]\!]$$

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

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

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

(LTS-COMM)
$$\frac{\begin{array}{l}\Omega_M \rhd M \xrightarrow{(\tilde{n}:\tilde{\mathsf{T}})k.a!V} \Omega'_M \rhd M' \\ \Omega_N \rhd N \xrightarrow{(\tilde{n}:\tilde{\mathsf{U}})k.a?V} \Omega'_N \rhd N'\end{array}}{\begin{array}{l}\Omega \rhd M \mid N \xrightarrow{\tau} \Omega \rhd (\mathsf{new}\, \widetilde{n} : \widetilde{\mathsf{T}})\, M' \mid N' \\ \Omega \rhd N \mid M \xrightarrow{\tau} \Omega \rhd (\mathsf{new}\, \widetilde{n} : \widetilde{\mathsf{T}})\, N' \mid M'\end{array}} \ \tilde{n} \cap \mathsf{fn}(N) = \emptyset$$

- *for every u but recursion variables in $\mathsf{dom}(\Gamma')$ we have $\Gamma(u) <: \Gamma'(u)$;*
- *for every recursion variable Z in $\mathsf{dom}(\Gamma')$ we have $\Gamma(Z) = \Gamma'(Z)$.* ■

Definition 5.3 (Configurations). *We call* configuration *a tuple of an environment Ω and a system M, written $\Omega \rhd M$, such that there exists an environment Γ, with $\Gamma <: \Omega$ and $\Gamma \vdash M$.* ■

The reader is referred to [HMR03] for the formal details.

Theorem 5.4. *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;
- the actions take the form $\Omega \rhd M \xrightarrow{\mu} \Omega' \rhd M'$; these are based on the labelled transitions system given in Figure 14 and 15.

FIGURE 15 Labelled transition semantics. External actions.

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

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

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

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

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

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

Definition 5.5 (Actions). *For configurations $\mathcal{C}$ of the form $(\Omega \rhd M)$, 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 [HMR03] for further motivation; this paper also contains the result that

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

whenever $\Gamma \vdash M$ and $\Gamma \vdash N$. So we establish Theorem 5.4 by showing

$$\Gamma \vdash M \text{ implies } (\Gamma \rhd M) \approx_{bis} (\Gamma \rhd \text{UNREC}(M)) \tag{4}$$

6 Proof of recursion implementability

Let us hint the problems encountered in trying to prove the equation (4) 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:

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

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:

$$\begin{aligned}\mathsf{IPSearch} \triangleq\ & \mathit{search\_req}\,?(x, \mathit{client}) \\ & \qquad \mathsf{goto}\, k_0.\ (\mathsf{newloc}\ \mathit{base} : \textsc{loc}[\mathit{ping}])\ \mathsf{F} \mid \mathsf{goto}\ \mathit{base}.* \ \mathsf{Inst}\end{aligned}$$

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

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

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:

$$\begin{aligned}&(\mathsf{new}\ \mathit{ping}_1)\ (\mathsf{new}\ \mathit{base}_1)\ (\mathsf{new}\ \mathit{ping}_2)\ (\mathsf{new}\ \mathit{base}_2)\ \ldots \\ &\quad \mathsf{Server}[\![\ldots]\!]\ \ldots \\ &\quad \mid \mathit{base}_1[\![\ldots]\!] \mid k_1^1[\![\ldots \mathsf{F}_1]\!] \mid k_1^2[\![\ldots \mathsf{F}_1]\!] \mid \ldots \\ &\quad \mid \mathit{base}_2[\![\ldots]\!] \mid k_2^1[\![\ldots \mathsf{F}_2]\!]\ \ldots \qquad (5)\end{aligned}$$

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 run-time between our translation and the standard, but non-compositional, one used in the PI-CALCULUS we previously mentioned, see [SW01], which arises because of the replication of $\mathsf{rec}\ Z.\ P$. A translation following the

lines of that in [SW01], would give rise to the following state, corresponding to (5) above:

$$\begin{array}{l}(\mathsf{new}\ \mathit{ping})\,(\mathsf{new}\ \mathit{base})\,\mathsf{Server}[\![*\ \mathit{search\_req}\,?(x, \mathit{client})\ \mathsf{goto}\ k_0.\mathsf{F}(x, \mathit{client})]\!] \\ \quad |\ \mathit{base}[\![*\ \mathit{ping}\,?(k, x, \mathit{client})\ \mathsf{goto}\ k.\mathit{test}\,?(y)\ldots]\!] \\ \quad |\ k_1^1[\![\ldots \mathsf{F}(x_1, \mathit{client}_1)]\!] \,|\, k_1^2[\![\ldots \mathsf{F}(x_1, \mathit{client}_1)]\!] \,|\ldots \\ \quad |\ k_2^1[\![\ldots \mathsf{F}(x_2, \mathit{client}_2)]\!] \ldots\end{array}$$

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

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 (5) depends on the system history. That is why our proof of (4) is based on an extended version of the translation in which we specify whether a given occurrence of $\mathsf{rec}\ Z.\ P$ 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 $\text{UNREC}_{\mathcal{P}}(M)$ for the translation of M parameterised by $\mathcal{P}$, with $\mathcal{P}$ specifying how each $\mathsf{rec}\ Z.\ P$ should be translated in M.

Definition 6.1 (Occurrence). *The occurrence o in a process P or a system M, written $P|_o$ and $M|_o$ is defined inductively by:*

- $P|_\varepsilon = P$ *and* $M|_\varepsilon = M$*;*
- $(P_1 \mid P_2)|_{1o} = P_1|_o$*,* $(P_1 \mid P_2)|_{2o} = P_2|_o$*, and similarly for* M*;*
- $u\,?(X : T)\,P|_{0o} = P|_o$*;*
- $k[\![P]\!]|_{0o} = P|_o$*;*
- *all other cases are similar.*

For any occurrence o in a system M, we call *system*-prefix any prefix o' such that $M|_{o'}$ is a system as opposed to a process.

For a given occurrence o and a given reduction $\Omega \rhd M \xrightarrow{\mu} \Omega' \rhd M'$, we will call *residual* of o the occurrence in M' of the system or process at o in M, if it still exists.

Definition 6.2 (Residual). *We call* residual *of an occurrence o in M after a reduction $\Omega \rhd M \xrightarrow{\mu} \Omega' \rhd M'$ the occurrence defined by the following function:*

- $\mathsf{Res}(\varepsilon, \Omega \rhd M \xrightarrow{\mu} \Omega' \rhd M') = \varepsilon$
- $\mathsf{Res}(1o, \Omega \rhd M \mid N \xrightarrow{\mu} \Omega' \rhd M' \mid N) = \mathsf{Res}(o, \Omega \rhd M \xrightarrow{\mu} \Omega' \rhd M')$
- $\mathsf{Res}(2o, \Omega \rhd M \mid N \xrightarrow{\mu} \Omega' \rhd M' \mid N) = 2o$
- $\mathsf{Res}(0, \Omega \rhd k[\![a\,!\langle V\rangle\, P]\!] \xrightarrow{k.a!V} \Omega' \rhd k[\![O]\!]) = \bot$
- $\mathsf{Res}(00o, \Omega \rhd k[\![a\,!\langle V\rangle\, P]\!] \xrightarrow{k.a!V} \Omega' \rhd k[\![O]\!]) = 0o$
- $\mathsf{Res}(0o, \Omega \rhd (\mathsf{new} n : \mathsf{T})\, M \xrightarrow{(n\tilde{n}:\mathsf{T}\tilde{\mathsf{T}})k.a!V} \Omega' \rhd M')$
  $= \mathsf{Res}(o, \Omega \rhd M \xrightarrow{(\tilde{n}:\tilde{\mathsf{T}})k.a!V} \Omega' \rhd M')$
- $\mathsf{Res}(0o, \Omega \rhd (\mathsf{new} n : \mathsf{T})\, M \xrightarrow{\mu} \Omega' \rhd (\mathsf{new} n : \mathsf{T})\, M')$
  $= 0\mathsf{Res}(o, \Omega \rhd M \xrightarrow{\mu} \Omega' \rhd M')$

and the other cases are similar.

For a given system M, the $\mathcal{P}$s we will consider will be annotated partitions of a part of occurrences in M. We define the "valid" $\mathcal{P}$s as:

- if there exists o_1, o_2 and O such that $o_i \in O \in \mathcal{P}$ then $M|_{o_1} = M|_{o_2} = \mathsf{rec}\ Z : \mathsf{R}.\ P$;
- for any $O \in \mathcal{P}$, we call o the longest common system-prefix of all occurrences in O; then all the free names in P are either free in M or bound at an occurrence that is a prefix of o.

The intuition is that the various occurrences of $\mathsf{rec}\ Z : \mathsf{R}.\ P$ in a given set in $\mathcal{P}$ will be attributed the same "home-base". The occurrences of the $\mathsf{rec}\ Z : \mathsf{R}.\ P$ will be translated by $\textsc{unrec}(\mathsf{rec}\ Z : \mathsf{R}.\ P)$.

To perform that translation, we need to keep track of the "current" occurrence within the system.

- if $o0$ is not in $\mathcal{P}$,

$$\begin{aligned}\textsc{unrec}^o_{\mathcal{P}}(k[\![\ \mathsf{rec}\ Z : \mathsf{R}.\ P]\!]) = \\ &(\mathsf{new}\, home_{Z_{\{o0\}}} : \textsc{loc}[ping_{Z_{\{o0\}}} : \textsc{rw}\langle \mathsf{R}_{\{o0\}}\rangle])\\ &home_{Z_{\{o0\}}}[\![* \ ping_{Z_{\{o0\}}}\ ?(l : \mathsf{R})\, \mathsf{goto}\, l.\textsc{unrec}^{o00}_{\mathcal{P}\cup\{\{o0\}^1\}}(P)]\!]\\ \mid\ &home_{Z_{\{o0\}}}[\![ping_{Z_{\{o0\}}}\ !\langle k\rangle]\!]\end{aligned}$$

- if o is not in $\mathcal{P}$, but the previous case does not apply because $\mathsf{rec}\ Z : \mathsf{R}.\ P$ occurs under a prefix,

$$\begin{aligned}&\textsc{unrec}^o_{\mathcal{P}}(\mathsf{rec}\ Z : \mathsf{R}.\ P) = (\mathsf{newloc}\, home_Z : \textsc{loc}[ping_Z : \textsc{rw}\langle \mathsf{R}\rangle])\\ &\quad(\textsc{unrec}(Z) \mid\\ &\qquad \mathsf{goto}\, home_Z.*\ ping_Z\ ?(l : \mathsf{R})\, \mathsf{goto}\, l.\textsc{unrec}(P))\end{aligned}$$

This will therefore heavily rely on implicit α-conversions.

- if $o0$ is in $\mathcal{P}$, then it must be in some O in $\mathcal{P}$;

$$\begin{aligned}&\textsc{unrec}^o_{\mathcal{P}}(k[\![\mathsf{rec}\ Z : \mathsf{R}.\ P]\!]) = \\ &\quad \mathit{home}_{Z_O}[\![\mathit{ping}_{Z_O}\ ?(l : \mathsf{R})\,\mathsf{goto}\,l.\textsc{unrec}^{o00}_{\mathcal{P}}(P)]\!] \\ &\qquad \mid \mathit{home}_{Z_O}[\![\mathit{ping}_{Z_O}\ !\langle k\rangle]\!]\end{aligned}$$

- if o is in $\mathcal{P}$, then it must be in some O in $\mathcal{P}$, when the previous case cannot apply;

$$\textsc{unrec}^o_{\mathcal{P}}(\mathsf{rec}\ Z : \mathsf{R}.\ P) = \mathsf{here}\,[x]\,\mathsf{goto}\,\mathit{home}_{Z_O}.\mathit{ping}_{Z_O}\ !\langle x\rangle$$

- we write o' for the occurrence of the binder of the occurrence o of Z; if o' is in $\mathcal{P}$, then it must be in some O in $\mathcal{P}$ and Z must be "Z_O";

$$\textsc{unrec}^o_{\mathcal{P}}(Z) = \mathsf{here}\,[x]\,\mathsf{goto}\,\mathit{home}_{Z_O}.\mathit{ping}_{Z_O}\ !\langle x\rangle$$

- we write o' for the occurrence of the binder of the occurrence o of Z; if o' is not in $\mathcal{P}$:

$$\textsc{unrec}^o_{\mathcal{P}}(Z) = \mathsf{here}\,[x]\,\mathsf{goto}\,\mathit{home}_{Z}.\mathit{ping}_{Z}\ !\langle x\rangle$$

- $\textsc{unrec}^o_{\mathcal{P}}(u\,!\langle V\rangle\,P) = u\,!\langle V\rangle\,\textsc{unrec}^{o0}_{\mathcal{P}}(P)$; all the other cases for processes are similar;

- if o is the longest system-prefix of the occurrences in $(O_i) \in \mathcal{P}$, we translate the system this way, with n_i the annotation of O_i in $\mathcal{P}$ and o_i one occurrence in O_i:

$$\begin{aligned}&\textsc{unrec}^o_{\mathcal{P}}((\mathsf{new}\,e : \mathsf{E})\,M) = \\ &\quad (\mathsf{new}\,e : \mathsf{E})\,(\mathsf{new}\,\mathit{home}_{Z_{O_1}} : \textsc{loc}[\mathit{ping}_{Z_{O_1}} : \textsc{rw}\langle \mathsf{R}_{O_1}\rangle]) \\ &\qquad \mathit{home}_{Z_{O_1}}[\![*\ \mathit{ping}_{Z_{O_1}}\ ?(l : \mathsf{R}_{O_1})\,\mathsf{goto}\,l.\textsc{unrec}^{o_i 0}_{\mathcal{P}}(M|_{o_i 0})]\!] \\ &\qquad\quad \vdots \times n_1 \\ &\qquad \mid \mathit{home}_{Z_{O_1}}[\![*\ \mathit{ping}_{Z_{O_1}}\ ?(l : \mathsf{R}_{O_1})\,\mathsf{goto}\,l.\textsc{unrec}^{o_i 0}_{\mathcal{P}}(M|_{o_i 0})]\!] \\ &\qquad \mid \mathit{home}_{Z_{O_2}}[\![\ldots]\!] \\ &\qquad\quad \vdots \\ &\qquad \mid \textsc{unrec}^{o0}_{\mathcal{P}}(M)\end{aligned}$$

  All other cases for system are similar, with the "generation" of all the home-bases that are required at that occurrence before the inductive case.

Notice that, up-to congruence for the order between the different locations home_Z introduced by the last case of the definition, $\textsc{unrec}^o_{\mathcal{P}}(k[\![\mathsf{rec}\ Z : \mathsf{R}.\ P]\!])$ when $o0$ is not in $\mathcal{P}$ is equal to $\textsc{unrec}^o_{\mathcal{P}\cup\{\{o0\}^1\}}(k[\![\mathsf{rec}\ Z : \mathsf{R}.\ P]\!])$.

Of course, we extend the notion of residual of an occurrence to the one of residual of a set $\mathcal{P}$.

We write $\text{UNREC}_{\mathcal{P}}(M)$ for $\text{UNREC}^{\varepsilon}_{\mathcal{P}}(M)$. Note that we do not need a special case for the translation of $k[\![Z]\!]$ since we know that this is an impossible situation.

To deal with the extra steps introduced by the translation, we will resort to a proof technique given in [JR04], 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 14. Thanks to bisimulations up-to-β we can focus only on the communication moves. Then we can consider that the *ping*-communication (which is a τ-move) in the translation corresponds to the recursion unwinding in recDpi.

Lemma 6.3 (unrec() is an bisimulation). *Suppose an environment* Γ *and a system* M*. Then* $\Gamma \vdash M$ *implies* $(\Gamma \rhd M) \approx_{bis} (\Gamma \rhd \text{UNREC}(M))$

Proof. We will prove that

$$\mathcal{R} = \{(\Omega \rhd M, \Omega \rhd \text{UNREC}_{\mathcal{P}}(M)) \mid \mathcal{P} \text{ is valid for } M\}$$

is a bisimulation up-to β.

Consider $(\Omega \rhd M, \Omega \rhd \text{UNREC}_{\mathcal{P}}(M))$ in $\mathcal{R}$. We know that there must exist some $\Gamma <: \Omega$ such that $\Gamma \vdash M$. We write here $\Omega \rhd M \xrightarrow{\mu}$ to express the fact that there exists some configuration $\Omega' \rhd M'$ such that $\Omega \rhd M \xrightarrow{\mu} \Omega' \rhd M'$.

- $\Omega \rhd M \xrightarrow{\mu} \Omega' \rhd M'$. We prove that $\Omega \rhd \text{UNREC}_{\mathcal{P}}(M) \xrightarrow{\mu} \xrightarrow{\tau}{}^{*}_{\beta} \equiv \Omega' \rhd \text{UNREC}_{\mathcal{P}'}(M')$ for some $\mathcal{P}'$, more precisely, if μ is an input or output action, $\mathcal{P}'$ is the residual of $\mathcal{P}$ after that transition. This proof is done by induction on the proof of $\Omega \rhd M \xrightarrow{\mu} \Omega' \rhd M'$. To get into the induction the property we prove is the fact that $\Omega_o \rhd M|_o \xrightarrow{\mu} \Omega'_{o'} \rhd M'|_{o'}$ implies that $\Omega \rhd \text{UNREC}^{o}_{\mathcal{P}}(M|_o) \xrightarrow{\mu} \xrightarrow{\tau}{}^{*}_{\beta} \equiv \Omega \rhd \text{UNREC}^{o'}_{\mathcal{P}'}(M'|_{o'})$.
  - (LTS-GO): $M|_o = k[\![\mathsf{goto}\, l.P]\!]$. This implies that $\text{UNREC}^{o}_{\mathcal{P}}(M|_o)$ is $k[\![\mathsf{goto}\, l.\text{UNREC}^{o00}_{\mathcal{P}}(P)]\!]$ optionnally with some $home_{Z_O}$ generation so that the general form is

    $$(\mathsf{new}\, home_{Z_{O_1}})\,(\mathsf{new}\, home_{Z_{O_2}}) \ldots \\ home_{Z_{O_1}}[\![\ldots]\!] \mid \ldots \mid k[\![\mathsf{goto}\, l.\text{UNREC}^{o00}_{\mathcal{P}}(P)]\!]$$

    which means that $\Omega_o \rhd \text{UNREC}^{o}_{\mathcal{P}}(M|_o)$ can perform the "matching" move by some application of rules (LTS-NEW), (LTS-PAR) and

(LTS-GO). The term it reaches is

$$(\mathsf{new}\, home_{Z_{O_1}})\,(\mathsf{new}\, home_{Z_{O_2}})\ldots \\ home_{Z_{O_1}}[\![\ldots]\!] \mid \ldots \mid l[\![\textsc{unrec}^{o00}_{\mathcal{P}}(P)]\!]$$

which might need some extra β-reductions to become the translation of $M'|_{o'} = l[\![P]\!]$ because there are different possible cases for the form of P. If P is of the form $\mathsf{rec}\ Z : \mathsf{R}.\ P'$:

* if $o'0$, the occurrence for the recursion operator, is in $\mathcal{P}'$ then it must be in some set O' in $\mathcal{P}'$ and $\textsc{unrec}^{o'}_{\mathcal{P}'}(M'|_{o'})$ is

$$(\mathsf{new}\, home_{Z_{O_1}})\ldots home_{Z_{O'}}[\![ping_{Z_{O'}}\,!\langle l\rangle]\!] \\ \mid home_{Z_{O'}}[\![ping_{Z_{O'}}\,?(l : \mathsf{R})\,\mathsf{goto}\, l.\textsc{unrec}^{o'00}_{\mathcal{P}'}(P')]\!]$$

  but, we will take $\mathcal{P}'$ to be the residual of $\mathcal{P}$ after the move so that $o'0$ is in $\mathcal{P}'$ exactly when $o00$ was in a set O in $\mathcal{P}$. This implies that $l[\![\textsc{unrec}^{o00}_{\mathcal{P}}(P)]\!]$ is of the form

$$l[\![\mathsf{here}\,[x]\,\mathsf{goto}\, home_{Z_O}.ping_{Z_O}\,!\langle x\rangle]\!]$$

  which reduces by β-moves to $home_{Z_O}[\![ping_{Z_O}\,!\langle x\rangle]\!]$. We also know by definition of the translation $\textsc{unrec}_{\mathcal{P}}(M)$ that at the longest common system-prefix among occurrences in O is generated the server in the home-base:

$$(\mathsf{new}\, home_{Z_O})\, home_{Z_O}[\![*\ ping_{Z_O}\,?(l : \mathsf{R})\,\mathsf{goto}\, l.\textsc{unrec}^{o00}_{\mathcal{P}}(P)]\!]$$

  so one β-move generates a new instance of the replicated process

$$home_{Z_O}[\![*\ ping_{Z_O}\,?(l : \mathsf{R})\,\mathsf{goto}\, l.\textsc{unrec}^{o00}_{\mathcal{P}}(P)]\!]$$

  which is exactly the system we need. And we can put this new instance by o' by congruence.

* if $o'0$ is not in $\mathcal{P}'$, we know that the translation we will give will be of the form

$$(\mathsf{new}\, home_{Z_{\{o'0\}}} : \textsc{loc}[ping_{Z_{\{o'0\}}} : \textsc{rw}\langle \mathsf{R}_{\{o'0\}}\rangle]) \\ home_{Z_{\{o'0\}}}[\![*\ ping_{Z_{\{o'0\}}}\,?(l : \mathsf{R})\,\mathsf{goto}\, l.\textsc{unrec}^{o'00}_{\mathcal{P}'\cup\{\{o'0\}\}}(P')]\!] \\ \mid home_{Z_{\{o'0\}}}[\![ping_{Z_{\{o'0\}}}\,?(l : \mathsf{R})\,\mathsf{goto}\, l.\textsc{unrec}^{o'00}_{\mathcal{P}'\cup\{\{o'0\}\}}(P')]\!] \\ \mid home_{Z_{\{o'0\}}}[\![ping_{Z_{\{o'0\}}}\,!\langle k\rangle]\!]$$

  but in that case, we will have $o00$ not in $\mathcal{P}$ so $l[\![\textsc{unrec}^{o00}_{\mathcal{P}}(P)]\!]$ will be of the form

$$l[\![(\mathsf{newloc}\, home_Z : \textsc{loc}[ping_Z : \textsc{rw}\langle \mathsf{R}\rangle]) \\ (\textsc{unrec}(Z) \mid \\ \mathsf{goto}\, home_Z.*\ ping_Z\,?(l : \mathsf{R})\,\mathsf{goto}\, l.\textsc{unrec}(P))]\!]$$

so by (LTS-L-CREATE), (LTS-SPLIT), (LTS-HERE), (LTS-GO) and (LTS-ITER) this reduces by β-moves into the translation of $M'|_{o'}$.

Otherwise, if P is not of the form $\mathsf{rec}\ Z : \mathsf{R}.\ P'$, we know that it cannot be of the simple form Z, since Z would in that case be a free recursion variable in the system. So it must be one of the various possible cases for processes. If we take the example of $a\,!\langle V\rangle\,P'$, by simply taking the residual of $\mathcal{P}$ for $\mathcal{P}'$, we get

$$\begin{aligned}\textsc{unrec}^{o'0}_{\mathcal{P}'}(a\,!\langle V\rangle\,P') &= a\,!\langle V\rangle\,\textsc{unrec}^{o'00}_{\mathcal{P}'}(P')\\ &= a\,!\langle V\rangle\,\textsc{unrec}^{o000}_{\mathcal{P}}(P')\\ &= \textsc{unrec}^{o00}_{\mathcal{P}}(a\,!\langle V\rangle\,P') \quad .\end{aligned}$$

– (LTS-SPLIT), (LTS-ITER), (LTS-L-CREATE), (LTS-C-CREATE), (LTS-EQ), (LTS-NEQ), (LTS-OUT) and (LTS-IN): those rules are similar to the previous case.

– (LTS-HERE): this case is similar because the substitution commutes with our translation.

– (LTS-REC): $M|_o = k[\![\mathsf{rec}\ Z : \mathsf{R}.\ P]\!]$ which reduces to $M'|_{o'} = k[\![P\{^{\mathsf{rec}\ (Z:\mathsf{R}).\ P}/_Z\}]\!]$. so the translation will depend on whether $o0$ is in $\mathcal{P}$:

  * if $o0$ is not in $\mathcal{P}$, as we mentionned earlier, $\textsc{unrec}^o_{\mathcal{P}}(M|_o)$ is equal to $\textsc{unrec}^o_{\mathcal{P}\cup\{\{o0\}\}}(M|_o)$, so we can restrict our analysis to the other case;

  * if $o0$ is in O in $\mathcal{P}$, we define $\mathcal{P}'$ as the residual of $\mathcal{P}$, namely with the occurrence $o0$ in $\mathcal{P}$ replaced by the occurrences of $\mathsf{rec}\ (Z : \mathsf{R}).\ P$ in $M'|_{o'}$; $\textsc{unrec}^o_{\mathcal{P}}(M|_o)$ is of the form

$$\begin{aligned}&\mathit{home}_{Z_O}[\![\mathit{ping}_{Z_O}\,?(l : \mathsf{R})\,\mathsf{goto}\,l.\textsc{unrec}^{o00}_{\mathcal{P}}(P)]\!]\\ &\quad |\ \mathit{home}_{Z_O}[\![\mathit{ping}_{Z_O}\,!\langle k\rangle]\!]\end{aligned}$$

  By the rule (LTS-COMM), this can reduce by a τ move into $\mathit{home}_{Z_O}[\![\mathsf{goto}\,k.\textsc{unrec}^{o00}_{\mathcal{P}}(P)]\!]$, so by an extra β-move, we reach $k[\![\textsc{unrec}^{o00}_{\mathcal{P}}(P)]\!]$. And we want to prove that this system can reduce in β-moves into $\textsc{unrec}^{o'}_{\mathcal{P}'}(k[\![P\{^{\mathsf{rec}\ (Z:\mathsf{R}).\ P}/_Z\}]\!])$. Now, remark that

$$\textsc{unrec}^{o''}_{\mathcal{P}''}(Z) = \textsc{unrec}^{o''}_{\mathcal{P}''}(\mathsf{rec}\ Z : \mathsf{R}.\ P)$$

  whenever o'' is not an occurrence of the form $o'''0$ with a system of the form $k[\![\ldots]\!]$ at o''' and when o'' is in $\mathcal{P}''$. Since both conditions are fulfilled in our case when considering the residual

of $\mathcal{P}$ for $\mathcal{P}'$, we get

$$\text{UNREC}^{o'}_{\mathcal{P}'}(k[\![P\{^{\mathsf{rec}\ (Z:\mathsf{R}).\ P}/_Z\}]\!]) = \text{UNREC}^{o'}_{\mathcal{P}'}(k[\![P]\!])$$

As in the case for rule (LTS-GO), showing the adequation between this translation and $k[\![\text{UNREC}^{o00}_{\mathcal{P}}(P)]\!]$ turns out to be a simple case analysis on the form of P.

– (LTS-COMM): $M|_o = M_1 \mid M_2$ and there exists some Ω_1 and Ω_2 such that $\Omega_1 \rhd M|_{o1} \xrightarrow{(\tilde{n}:\widetilde{\mathsf{T}})k.a!V} \Omega'_1 \rhd M'|_{o''1}$ and $\Omega_2 \rhd M|_{o2} \xrightarrow{(\tilde{n}:\widetilde{\mathsf{U}})k.a?V} \Omega'_2 \rhd M'|_{o''2}$. By our induction hypothesis, we can conclude that, writing $\mathcal{P}'$ for the residual of $\mathcal{P}$ after the communication move

$$\Omega_1 \rhd \text{UNREC}^{o1}_{\mathcal{P}}(M|_{o1}) \xrightarrow{(\tilde{n}:\widetilde{\mathsf{T}})k.a!V} \xrightarrow{\tau}{}^*_\beta \equiv \Omega'_1 \rhd \text{UNREC}^{o''1}_{\mathcal{P}'}(M'|_{o''1})$$

and

$$\Omega_2 \rhd \text{UNREC}^{o2}_{\mathcal{P}}(M|_{o2}) \xrightarrow{(\tilde{n}:\widetilde{\mathsf{U}})k.a?V} \xrightarrow{\tau}{}^*_\beta \equiv \Omega'_2 \rhd \text{UNREC}^{o''2}_{\mathcal{P}'}(M'|_{o''2})$$

which implies

$$\begin{aligned}
&\Omega \rhd \text{UNREC}^{o}_{\mathcal{P}}(M|_o) \xrightarrow{\tau} \xrightarrow{\tau}{}^*_\beta \\
&\quad \equiv \Omega \rhd (\mathsf{new}\,\tilde{n} : \widetilde{\mathsf{T}})\, \text{UNREC}^{o''1}_{\mathcal{P}'}(M'|_{o''1}) \mid \text{UNREC}^{o''2}_{\mathcal{P}'}(M'|_{o''2}) \\
&\quad = \Omega \rhd (\mathsf{new}\,\tilde{n} : \widetilde{\mathsf{T}})\, \text{UNREC}^{o''}_{\mathcal{P}'}(M'|_{o''}) \\
&\quad = \Omega \rhd \text{UNREC}^{o'}_{\mathcal{P}'}(M'|_{o'})
\end{aligned}$$

these equalities being true with the omission of the extra $home_{Z_O}$ that might be generated by $\text{UNREC}^{o}_{\mathcal{P}}(M|_o)$ for the sake of simplicity. They would be dealt with properly in the two intermediary steps, keeping the same conclusion.

– (LTS-NEW), (LTS-OPEN), (LTS-WEAK) and (LTS-PAR): we simply apply, for those rules, the induction hypothesis.

• $\Omega \rhd \text{UNREC}^{o}_{\mathcal{P}}(M) \xrightarrow{\mu} \Omega' \rhd N'$. Here are the different possible cases for the axiomatic rules in the proof of this reduction.

– (LTS-ITER) applied on a channel $ping_{Z_O}$: in that case we simply modify the annotation on O in $\mathcal{P}$ from n to $n+1$ to accomodate for that new instance of the replicated process. That move is then match by an absence of move in M, because N' is still a translation of M.

– (LTS-IN) and (LTS-OUT) on a channel $ping_{Z_O}$. Then the reduction we are considering is a communication on that channel. Notice that it is impossible to have only an input or only an input on a channel $ping_{Z_O}$, since all those channels have restricted scopes.

Since we have an output prefix on that channel $ping_{Z_O}$, by definition of the translation it must be due to some $\mathsf{rec}\ Z : \mathsf{R}.\ P$

in M. So we have $\Omega \rhd M \xrightarrow{\tau} \Omega \rhd M'$, that τ corresponding to the recursion unwinding. By a similar proof as in the matching of a move in M by a move in its translation, we then show that $\Omega \rhd N'$ can further reduce into some $\Omega \rhd \textsc{unrec}_{\mathcal{P}'}(M')$ for some $\mathcal{P}'$.

– Otherwise, by definition of the translation, we know that the redex in $\textsc{unrec}_{\mathcal{P}}(M)$ must also exist in M, so $\Omega \rhd M \xrightarrow{\mu} \Omega' \rhd M'$. By a proof similar to the previous case, we can therefore show that $\Omega \rhd \textsc{unrec}_{\mathcal{P}}(M) \xrightarrow{\mu} \xrightarrow{\tau}_{\beta}^{*} \equiv \Omega' \rhd \textsc{unrec}_{\mathcal{P}'}(M')$, since the redex reduced in the μ-move is the same.

□

7 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

[AC93] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, September 1993.

[FH05] Adrian Francalanza and Matthew Hennessy. Location and link failure in a distributed pi-calculus. Computer Science Report 2005:01, University of Sussex, 2005.

[GLP03] Vladimir Gapeyev, Michael Levin, and Benjamin Pierce. Recursive subtyping revealed. *Journal of Functional Programming*, 12(6):511–548, 2003. 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).

[HMR03] Matthew Hennessy, Massimo Merro, and Julian Rathke. Towards a behavioural theory of access and mobility control in distributed systems. *Theoretical Computer Science*, 322:615–669, 2003.

[HR02] Matthew Hennessy and James Riely. Resource access control in systems of mobile agents. *Information and Computation*, 173:82–120, 2002.

[JR04] Alan Jeffrey and Julian Rathke. A theory of bisimulation for a fragment of concurrent ml with local names. *Theoretical Computer Science*, 323:1–48, 2004.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[SW01] Davide Sangiorgi and David Walker. *The π-calculus*. Cambridge University Press, 2001.