Adding Recursion to Dpi

Samuel Hym [1]

PPS, Université Paris 7 & CNRS

Matthew Hennessy [2]

Department of Informatics, University of Sussex

Abstract

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

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

Key words: dpi-calculus, recursion, implementation using replication, recursive and coinductive types

1 Introduction

The pi-calculus, [1], is a well-known formal calculus for describing, and reasoning about, the behaviour of concurrent processes which interact via two-way communication channels. Dpi, [2], 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

[1] Samuel.Hym@pps.jussieu.fr
[2] M.Hennessy@sussex.ac.uk

comes equipped with a sophisticated capability-based type system, and a co-inductive behavioural theory which takes into account the constraints imposed by these types, [3,2]. 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 the process owns the entity. Moreover this type may change over time, reflecting the fact that processes may gradually accumulate capabilities over entities. Types thus indicate the *rights* some process owns over those 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 [1].

However the situation changes when we move to the distributed world of DPI: a replicated process is tied to the location where it is started whereas a recursive process can perform its recursive calls anywhere. 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 extend the already existing type system for DPI and 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 [3,2] for more introductory work on 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 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. We will see in Example 2 how this new construct is useful for the description of recursive processes: without $\mathsf{here}\,[x]$, a generic recursive process would indeed have no way to assert in which location it is triggered. For non-recursive processes, it is always possible to encode this self-localisation by using the name or variable of the last location the process migrated to.

Example 1 (Searching a network)**.** Consider the following recursive process, which searches a network for certain values satisfying some unspecified predicate p:

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

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

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

Fig. 1 Syntax of recDpi

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

P, Q ::=	*Processes*
$u\,!\,\langle V \rangle\, P$	Output
$u\,?\,(X : \mathsf{T})\, P$	Input
$\mathsf{goto}\, v.\ P$	Migration
$\mathsf{if}\ u_1 = u_2\ \mathsf{then}\ P\ \mathsf{else}\ Q$	Matching
$(\mathsf{newc}\, c : \mathsf{C})\, P$	Channel creation
$(\mathsf{newloc}\, k : \mathsf{K})\, P$	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**

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 12. 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 just like the rule (LTS-ITER).

Example 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 it finds a neighbour via the local channel *neigh*. It then sets up a listener on *ans* to await news and, concurrently, it migrates to his neighbour, 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, 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

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

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

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

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. They serve to indicate the capabilities required by a recursive process to run.

Fig. 2 Recursive pre-types

Base Types:	$\mathsf{B} ::= \mathbf{int} \mid \mathbf{bool} \mid \mathbf{unit} \ldots$
Local Channel Types:	$\mathsf{A} ::= \textsc{r}\langle \mathsf{U} \rangle \mid \textsc{w}\langle \mathsf{T} \rangle \mid \textsc{rw}\langle \mathsf{U}, \mathsf{T} \rangle$
Location Types:	$\mathsf{K} ::= \textsc{loc}[u_1 : \mathsf{A}_1, \ldots, u_n : \mathsf{A}_1], n \geq 0 \mid \mu\mathbf{Y}.\mathsf{K} \mid \mathbf{Y}$
Value Types:	$\mathsf{V} ::= \mathsf{B} \mid \mathsf{A} \mid (\tilde{\mathsf{A}})_@\mathsf{K}$
Transmission Types:	$\mathsf{T}, \mathsf{U} ::= (\mathsf{V}_1, \ldots, \mathsf{V}_n), n \geq 0$

3 Co-inductive types for recDpi

There is a well-established capability-based type system for Dpi, [2], which we can adapt to recDpi. This adaptation will be as smooth as possible: we will present co-inductive types and proofs about them in a form really close to what is done in standard Dpi.

3.1 The Types

In this type system local channels have read/write types of the form $\textsc{r}\langle \mathsf{U} \rangle$, $\textsc{w}\langle \mathsf{T} \rangle$, or $\textsc{rw}\langle \mathsf{U}, \mathsf{T} \rangle$ (meaning that values are written at type T and read at type U on a channel of that type), provided the object types U and T "agree", as will be explained later. Locations have record types, of the form

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

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

However it turns out that we need to consider *infinite* location types if we want to allow some recursive behaviours. To see this consider again the searching process Search from Example 1. Any site, such as k, which can support this process needs to have a local channel called *neigh* from which values can be read. These values must be locations, and let us consider their type, that is the object type of *neigh*. These locations must have a local channel called *test*, of an appropriate type, and a local channel called *neigh*; the object type of this local channel must be in turn the same as the type we are trying to describe. Using a recursion operator μ, this type can be described as

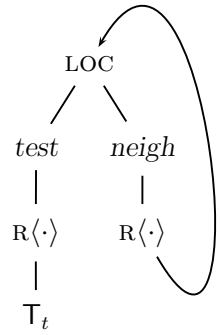
$$\mu\mathbf{Y}.\textsc{loc}[\mathit{test} : \textsc{r}\langle \mathsf{T}_t \rangle, \quad \mathit{neigh} : \textsc{r}\langle \mathbf{Y} \rangle]$$

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

The set of recursive pre-types is given in Figure 2, and is obtained by adding the operator $\mu\mathbf{Y}.\mathsf{K}$ and the variable $\mathbf{Y}$ as constructors to the type forma-

tion rules for DPI. Following [4] we can associate with each recursive pre-type T a co-inductive pre-type denoted $\mathsf{Tree}(\mathsf{T})$, which takes the form of a finite-branching, but possibly infinite, tree whose nodes are labelled by the type constructors. The co-inductive approach simplifies greatly reasoning over types: for instance, the expected equality is indeed the equality of those possibly infinite trees and it is much more difficult to define when unfolding is involved (unfolding the recursion can be insufficient to obtain two identical types). The advantage of using this approach increases further when we will develop the full theory of subtyping.

To give an example of tree pre-type, $\mathsf{Tree}(\mathsf{S})$ is the infinite regular tree we can represent with the following graph:



Definition 3 (Contractive and Tree pre-type)**.** We call a recursive pre-type S *contractive* if for every $\mu\mathbf{Y}.\mathsf{S}'$ it contains, $\mathbf{Y}$ can only appear in S' under an occurrence of LOC. In the paper we will only consider contractive pre-types.

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

- recursive pre-types are unwound $\mathsf{Tree}(\mu\mathbf{Y}.\mathsf{S}') = \mathsf{Tree}(\mathsf{S}'\{\!\{^{\mu\mathbf{Y}.\mathsf{S}'}/_{\mathbf{Y}}\}\!\})$
- $\mathsf{Tree}(\cdot)$ is homomorphic on any other construct; for instance

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

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

Note that $\mathsf{Tree}(\mathsf{S})$ is defined only when the recursive pre-type S is contractive. This condition ensures that types of the form $\mu\mathbf{Y}.\mathbf{Y}$ are avoided: we expect our types to bring some piece of information about how the name or variable it is attached to can be used, and such a type would not bring any information.

To make clearer the distinction between tree pre-types and recursive pre-types, we slightly modify the notation and fonts of types. So the recursive pre-type

$$\mathrm{LOC}[\mathit{test} : \mathrm{R}\langle\mathsf{T}_t\rangle]$$

Fig. 3 DPI subtyping rules

$$\frac{}{\mathbf{base} <: \mathbf{base}}\ (\text{SUB-BASE})$$

$$\frac{\mathsf{C}_i <: \mathsf{C}'_i}{(\widetilde{\mathsf{C}}) <: (\widetilde{\mathsf{C}'})}\ (\text{SUB-TUPLE})$$

$$\frac{\mathsf{A}_1 <: \mathsf{A}_2 \qquad \mathsf{K}_1 <: \mathsf{K}_2}{\mathsf{A}_1 @ \mathsf{K}_1 <: \mathsf{A}_2 @ \mathsf{K}_2}\ (\text{SUB-HOM})$$

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

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

corresponds to the tree pre-type

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

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. If a channel of such type were to be allowed, a system could rightfully contain a process sending an integer on that channel and another process expecting on that same channel some data of type $\mathsf{r}\langle\rangle$: when trying to use that data as a channel, the system would perform a runtime error.

This is then avoided using a notion of subtype, and demanding that, in types of the form $\mathsf{rw}\langle \mathsf{U}, \mathsf{T} \rangle$, T must be a subtype of 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, [4,5]. 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:

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

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

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

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

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.

Most of the proofs will also rely on the fact that Sub is "invertible": we will often consider some pair $(\mathsf{T}_1, \mathsf{T}_2)$ in some $\mathsf{Sub}(\mathcal{R})$ and deduce the possible forms of T_1 and T_2 since only one of the cases of the definition of Sub can apply.

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

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

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 $\mathsf{T}_1 = \mathsf{T}_2$ when $(\mathsf{T}_1, \mathsf{T}_2)$ is in $\nu\mathsf{Eq}$.

With this notion of equality, we show now how 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 5 (Partial order)**.** $<:$ *is partial order on tree types.*

Proof. Let us first prove that $\nu\mathsf{Sub}$ is reflexive, namely that for any tree type T, $\mathsf{T} <: \mathsf{T}$.

For this, let us consider the relation

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

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 (T, T), we reason on the form of T.

- $\mathbf{base}$ then (T, T) is obviously in $\mathsf{Sub}(\mathcal{R})$.

- $\mathsf{r}\langle \mathsf{U}_0 \rangle$ then, since $(\mathsf{U}_0, \mathsf{U}_0)$ is in $\mathcal{R}$, (T, T) is in $\mathsf{Sub}(\mathcal{R})$.
- $\mathsf{rw}\langle \mathsf{U}_0, \mathsf{T}_0 \rangle$ then, by well-formedness of T, we know that $\mathsf{T}_0 <: \mathsf{U}_0$ so $(\mathsf{T}_0, \mathsf{U}_0)$ is in $\mathcal{R}$. Of course, so are $(\mathsf{T}_0, \mathsf{T}_0)$ and $(\mathsf{U}_0, \mathsf{U}_0)$, which implies that (T, 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 $\mathsf{Sub}(\mathcal{R})$, which concludes the proof of reflexivity.

We now prove that $\nu\mathsf{Sub}$ is antisymmetric, namely that for any tree types T_1 and T_2, $\mathsf{T}_1 <: \mathsf{T}_2$ and $\mathsf{T}_2 <: \mathsf{T}_1$ imply $\mathsf{T}_1 = \mathsf{T}_2$.

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

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

We show that this is a postfixpoint of Eq.

For this let us consider two types T_1 and T_2 such that $(\mathsf{T}_1, \mathsf{T}_2)$ is in $\mathcal{R}$. Then we reason by cases on $\mathsf{T}_1 <: \mathsf{T}_2$: $(\mathsf{T}_1, \mathsf{T}_2) \in \mathsf{Sub}(\nu\mathsf{Sub})$ implies that one of the cases of the definition of Sub must apply. We give here only typical examples:

- $\mathsf{T}_1 = \mathsf{T}_2 = \mathbf{base}$ then $(\mathsf{T}_1, \mathsf{T}_2)$ is obviously in $\mathsf{Eq}(\mathcal{R})$;
- $\mathsf{T}_1 = (\tilde{\mathsf{C}}^1)$ and $\mathsf{T}_2 = (\tilde{\mathsf{C}}^2)$ with $\mathsf{C}_i^1 <: \mathsf{C}_i^2$ for all i; then $\mathsf{T}_2 <: \mathsf{T}_1$ implies also that $\mathsf{C}_i^2 <: \mathsf{C}_i^1$ for all i which means that $(\mathsf{C}_i^1, \mathsf{C}_i^2)$ are also in $\mathcal{R}$; this entails that $(\mathsf{T}_1, \mathsf{T}_2)$ is in $\mathsf{Eq}(\{(\mathsf{C}_i^1, \mathsf{C}_i^2)\}) \subseteq \mathsf{Eq}(\mathcal{R})$ by monotonicity of Eq;
- $\mathsf{T}_1 = \mathsf{rw}\langle \mathsf{U}_1', \mathsf{T}_1' \rangle$ and $\mathsf{T}_2 = \mathsf{r}\langle \mathsf{U}_2' \rangle$ is impossible because $\mathsf{T}_2 \not<: \mathsf{T}_1$.
- The remaining cases are similar.

This proves that $\mathcal{R}$ is included in $\mathsf{Eq}(\mathcal{R})$, from which we can conclude that $\nu\mathsf{Sub}$ is antisymmetric.

And finally, we prove that $\nu\mathsf{Sub}$ is transitive, namely that for any tree types T_1, T_2 and T_3, $\mathsf{T}_1 <: \mathsf{T}_2$ and $\mathsf{T}_2 <: \mathsf{T}_3$ imply that $\mathsf{T}_1 <: \mathsf{T}_3$.

For this, 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 $(\mathsf{T}_1, \mathsf{T}_3)$ in $\mathsf{Tr}(\nu\mathsf{Sub})$. By definition of Tr this implies that either $(\mathsf{T}_1, \mathsf{T}_3)$ 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 T_2 such that $(\mathsf{T}_1, \mathsf{T}_2)$ and $(\mathsf{T}_2, \mathsf{T}_3)$ are in $\nu\mathsf{Sub}$. So we proceed by a case analysis on the form of $\mathsf{T}_1 <: \mathsf{T}_2$. We give here only typical examples:

- $\mathsf{T}_1 = (\tilde{\mathsf{C}}^1)$ and $\mathsf{T}_2 = (\tilde{\mathsf{C}}^2)$ with $\mathsf{C}_i^1 <: \mathsf{C}_i^2$: then $\mathsf{T}_2 <: \mathsf{T}_3$ implies that T_3 must also be of the form $(\tilde{\mathsf{C}}^3)$ with $\mathsf{C}_i^2 <: \mathsf{C}_i^3$. So for all i, $(\mathsf{C}_i^1, \mathsf{C}_i^3)$ is in $\mathsf{Tr}(\nu\mathsf{Sub})$ which implies that $(\mathsf{T}_1, \mathsf{T}_3)$ is indeed in $\mathsf{Sub}(\mathsf{Tr}(\nu\mathsf{Sub}))$.
- $\mathsf{T}_1 = \mathsf{rw}\langle \mathsf{U}_1', \mathsf{T}_1' \rangle$ and $\mathsf{T}_2 = \mathsf{rw}\langle \mathsf{U}_2', \mathsf{T}_2' \rangle$ with $\mathsf{T}_2' <: \mathsf{T}_1' <: \mathsf{U}_1' <: \mathsf{U}_2'$. Then $\mathsf{T}_2 <: \mathsf{T}_3$ implies that T_3 must be of one of the following forms: $\mathsf{r}\langle \mathsf{U}_3' \rangle$, $\mathsf{w}\langle \mathsf{T}_3' \rangle$ or $\mathsf{rw}\langle \mathsf{U}_3', \mathsf{T}_3' \rangle$. In any case we have $\mathsf{T}_3' <: \mathsf{T}_2' <: \mathsf{U}_2' <: \mathsf{U}_3'$ or at least the relevant part of that inequation whenever T_3' or U_3' is not defined. Therefore we know that $\mathsf{Tr}(\nu\mathsf{Sub})$ contains the relevant pairs among $(\mathsf{T}_3', \mathsf{T}_1')$, $(\mathsf{T}_1', \mathsf{U}_1')$ and $(\mathsf{U}_1', \mathsf{U}_3')$. So we can conclude that $(\mathsf{T}_1, \mathsf{T}_3)$ is in $\mathsf{Sub}(\mathsf{Tr}(\nu\mathsf{Sub}))$.

□

With the order $<:$ on types come a notion of compatibility of types, and a *meet* relation.

Definition 6 (Compatible types)**.** Suppose two types T_1 and T_2. We say that they are compatible, written $\mathsf{T}_1 \downarrow \mathsf{T}_2$ whenever there exists some type T such that $\mathsf{T} <: \mathsf{T}_1$ and $\mathsf{T} <: \mathsf{T}_2$. We say that they are upward-compatible, written $\mathsf{T}_1 \uparrow \mathsf{T}_2$ whenever there exists some type T such that $\mathsf{T}_1 <: \mathsf{T}$ and $\mathsf{T}_2 <: \mathsf{T}$.

The meet relation and its dual, the join relation, are relations over triples of types. To define *meet*, we proceed as for the subtyping relation, namely by defining a function the greatest fixpoint of which will be our relation. But *meet* and *join* must be defined at the same time since we have to deal with contravariance in our types. So we define a function $\mathsf{MeetJoin}$ over a set of triples either of the form $\sqcap(\mathsf{T}_1, \mathsf{T}_2, \mathsf{T}_3)$ or $\sqcup(\mathsf{T}_1, \mathsf{T}_2, \mathsf{T}_3)$, where the T_i are tree types. We give in Figure 4 the definition of the function $\mathsf{MeetJoin}$. All the individual clauses in this definition are inherited from the standard rules in Dpi: different cases correspond to the join of two read-write types, to take into account incompatibilities of types. Those cases are explicitly disjoint thanks to compatibility conditions, stating for instance that the join of two types $\mathsf{rw}\langle \mathsf{U}_1, \mathsf{T}_1 \rangle$ and $\mathsf{rw}\langle \mathsf{U}_2, \mathsf{T}_2 \rangle$ are of the form $\mathsf{rw}\langle \cdot, \cdot \rangle$ as soon as $\mathsf{U}_1 \uparrow \mathsf{U}_2$ and $\mathsf{T}_1 \downarrow \mathsf{T}_2$. Those conditions would be automatically obtained in the greatest fixpoint of that function. We keep them to emphasise the distinction between the different cases.

We will write $\mathsf{T}_1 \sqcap \mathsf{T}_2 = \mathsf{T}_3$ for types T_1, T_2 and T_3 such that $\sqcap(\mathsf{T}_1, \mathsf{T}_2, \mathsf{T}_3)$ is in $\nu\mathsf{MeetJoin}$.

Fig. 4 MeetJoin definition

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

We still need to show that the greatest fixpoint of this function actually gives us operators with the properties we expect. The first of these properties is the fact that our rules define a partial function, since it is not immediate because meet is defined as a relation over triples of tree types.

Lemma 7 (Meet is a function)**.** *For any types* T_1*,* T_2*,* T_3 *and* T_4*,* $\mathsf{T}_1 \sqcap \mathsf{T}_2 = \mathsf{T}_3$ *and* $\mathsf{T}_1 \sqcap \mathsf{T}_2 = \mathsf{T}_4$ *implies* $\mathsf{T}_3 = \mathsf{T}_4$*.*

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:

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

and we show that this is a postfixpoint of Eq.

Let us consider T_3 and T_4 such that $\mathsf{T}_3 \mathrel{\mathcal{R}} \mathsf{T}_4$, and we write T_1 and T_2 for the corresponding two types. We reason on the possible cases for $\mathsf{T}_1 \sqcap \mathsf{T}_2 = \mathsf{T}_3$ or $\mathsf{T}_1 \sqcup \mathsf{T}_2 = \mathsf{T}_3$. We see here some typical examples.

- $\mathbf{base} \sqcap \mathbf{base} = \mathbf{base}$; then the only possible T_4 is $\mathbf{base}$, so $(\mathsf{T}_3, \mathsf{T}_4)$ is obviously in $\mathsf{Eq}(\mathcal{R})$.
- $(\tilde{\mathsf{C}}^1) \sqcap (\tilde{\mathsf{C}}^2) = (\tilde{\mathsf{C}}^3)$ with $\mathsf{C}^1_i \sqcap \mathsf{C}^2_i = \mathsf{C}^3_i$ for all i; then the only possible T_4 is of the form $(\tilde{\mathsf{C}}^4)$ with $\mathsf{C}^1_i \sqcap \mathsf{C}^2_i = \mathsf{C}^4_i$, so $(\mathsf{C}^3_i, \mathsf{C}^4_i)$ also are in $\mathcal{R}$. So

$$(\mathsf{T}_3, \mathsf{T}_4) \in \mathsf{Eq}(\{(\mathsf{C}^3_i, \mathsf{C}^4_i)\}) \subseteq \mathsf{Eq}(\mathcal{R}) \ .$$

- $\mathsf{loc}[u_1 : \mathsf{A}_1, \ldots, u_n : \mathsf{A}_n, v_1 : \mathsf{B}_1, \ldots] \sqcup \mathsf{loc}[u_1 : \mathsf{A}'_1, \ldots, u_n : \mathsf{A}'_n, w_1 : \mathsf{B}'_1, \ldots] = \mathsf{loc}[u_{i_1} : \mathsf{A}''_{i_1}, \ldots, u_{i_p} : \mathsf{A}''_{i_p}]$ then we know that T_4 must be of the form $\mathsf{loc}[u_{i_1} : \mathsf{A}'''_{i_1}, \ldots, u_{i_p} : \mathsf{A}'''_{i_p}]$ because the set of indices $\{i_j\}$ is determined by the compatibility of the types A_i and A'_i; this means that $(\mathsf{A}''_{i_j}, \mathsf{A}'''_{i_j})$ are in $\mathcal{R}$ for every i_j, so

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

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 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, first notice that $\sqcap$ is symmetric over its two arguments since the definition of the function $\mathsf{MeetJoin}$ is symmetric over the first two components of triples.

Lemma 8 (Meet is a subtype)**.** *For any types* T_1*,* T_2 *and* T_3 *such that* $\mathsf{T}_1 \sqcap \mathsf{T}_2 = \mathsf{T}_3$*, we have* $\mathsf{T}_3 <: \mathsf{T}_1$ *and* $\mathsf{T}_3 <: \mathsf{T}_2$*.*

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

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

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

Let us consider $(\mathsf{T}^1, \mathsf{T}^2)$ 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 T^3 the type proving that $(\mathsf{T}^1, \mathsf{T}^2)$ is in $\mathcal{R}$. We reason on the proof of $\mathsf{T}^2 \sqcap \mathsf{T}^3 = \mathsf{T}^1$ or of $\mathsf{T}^1 \sqcup \mathsf{T}^3 = \mathsf{T}^2$. Let us start with $\mathsf{T}^2 \sqcap \mathsf{T}^3 = \mathsf{T}^1$. We give here only some typical examples.

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

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

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

Lemma 9 (Meet is the greatest subtype)**.** *For any types* T_1, T_2, T_3 *and* T *such that* $\mathsf{T}_1 \sqcap \mathsf{T}_2 = \mathsf{T}_3$, $\mathsf{T} <: \mathsf{T}_1$ *and* $\mathsf{T} <: \mathsf{T}_2$, *we have* $\mathsf{T} <: \mathsf{T}_3$.

Proof. Let

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

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 $(\mathsf{T}, \mathsf{T}^3)$, with the corresponding types T^1 and T^2. We reason on $\mathsf{T}^1 \sqcap \mathsf{T}^2 = \mathsf{T}^3$. As usual, we give some typical cases.

- $(\widetilde{\mathsf{C}^1}) \sqcap (\widetilde{\mathsf{C}^2}) = (\widetilde{\mathsf{C}^3})$ then $\mathsf{T} <: \mathsf{T}^1$ implies that T is of the form $(\widetilde{\mathsf{C}})$ and for each i, we have $\mathsf{C}_i <: \mathsf{C}_i^1$, $\mathsf{C}_i <: \mathsf{C}_i^2$ and $\mathsf{C}_i^1 \sqcap \mathsf{C}_i^2 = \mathsf{C}_i^3$, which means that, for each i, $(\mathsf{C}_i, \mathsf{C}_i^3)$ is in $\mathcal{R}$. Consequently $(\mathsf{T}, \mathsf{T}^3)$ is in $\mathsf{Sub}(\mathcal{R})$.
- $\mathsf{r}\langle \mathsf{U}_0^1 \rangle \sqcap \mathsf{w}\langle \mathsf{T}_0^2 \rangle = \mathsf{rw}\langle \mathsf{U}_0^1, \mathsf{T}_0^2 \rangle$ then $\mathsf{T} <: \mathsf{T}^1$ implies that T can only be of the form $\mathsf{r}\langle \mathsf{U}_0 \rangle$ or $\mathsf{rw}\langle \mathsf{U}_0, \mathsf{T}_0 \rangle$ with $\mathsf{U}_0 <: \mathsf{U}_0^1$. Similarly, $\mathsf{T} <: \mathsf{T}^2$ implies that T can only be of the form $\mathsf{w}\langle \mathsf{T}_0 \rangle$ or $\mathsf{rw}\langle \mathsf{U}_0, \mathsf{T}_0 \rangle$ with $\mathsf{T}_0^2 <: \mathsf{T}_0$. By combining those two constraints, we know it must of the form $\mathsf{rw}\langle \mathsf{U}_0, \mathsf{T}_0 \rangle$ with $\mathsf{U}_0 <: \mathsf{U}_0^1$ and $\mathsf{T}_0^2 <: \mathsf{T}_0$. By well-formedness of T we also know that $\mathsf{T}_0 <: \mathsf{U}_0$. Which means that $(\mathsf{T}, \mathsf{T}^3)$ is in $\mathsf{Sub}(\nu\mathsf{Sub}) \subseteq \mathsf{Sub}(\mathcal{R})$.
- $\mathsf{r}\langle \mathsf{U}_0^1 \rangle \sqcap \mathsf{rw}\langle \mathsf{U}_0^2, \mathsf{T}_0^2 \rangle = \mathsf{rw}\langle \mathsf{U}_0^3, \mathsf{T}_0^2 \rangle$, then $\mathsf{T} <: \mathsf{T}^2$ implies that T is of the form $\mathsf{rw}\langle \mathsf{U}_0, \mathsf{T}_0 \rangle$ with $\mathsf{U}_0 <: \mathsf{U}_0^2$ and $\mathsf{T}_0^2 <: \mathsf{T}_0$. We also have that $\mathsf{U}_0 <: \mathsf{U}_0^1$. Of course, we have that $\mathsf{U}_0^1 \sqcap \mathsf{U}_0^2 = \mathsf{U}_0^3$, so $(\mathsf{U}_0, \mathsf{U}_0^3)$ is in $\mathcal{R}$. As so is $(\mathsf{T}_0^2, \mathsf{T}_0)$ and $(\mathsf{T}_0, \mathsf{U}_0)$ by well-formedness of T, $(\mathsf{T}, \mathsf{T}^3)$ is in $\mathsf{Sub}(\mathcal{R})$.
- $\mathsf{loc}[u_i : \mathsf{A}_i^1, v_j : \mathsf{B}_j^1] \sqcap \mathsf{loc}[u_i : \mathsf{A}_i^2, w_k : \mathsf{B}_k^2] = \mathsf{loc}[u_i : \mathsf{A}_i^3, v_j : \mathsf{B}_j^1, w_k : \mathsf{B}_k^2]$, which implies that $\mathsf{A}_i^1 \sqcap \mathsf{A}_i^2 = \mathsf{A}_i^3$ for all i. The fact T is a common subtype of T^1 and T^2 implies that it must be of the form $\mathsf{loc}[u_i : \mathsf{A}_i, v_j : \mathsf{B}_j^4, w_k : \mathsf{B}_k^5, x_l : \mathsf{B}_l^6]$ with $\mathsf{A}_i <: \mathsf{A}_i^1$ and $\mathsf{A}_i <: \mathsf{A}_i^2$ for all i, and with $\mathsf{B}_j^4 <: \mathsf{B}_j^1$ and $\mathsf{B}_k^5 <: \mathsf{B}_k^2$. This implies that $(\mathsf{A}_i, \mathsf{A}_i^3)$, $(\mathsf{B}_j^4, \mathsf{B}_j^1)$ and $(\mathsf{B}_k^5, \mathsf{B}_k^2)$ are in $\mathcal{R}$. So $(\mathsf{T}, \mathsf{T}^3)$ is in $\mathsf{Sub}(\mathcal{R})$.

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

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

To prove the main result about the meet operator on tree types, namely the existence of partial meets, we need to develop a bit further the theory of tree types with a notion of *subtree* at a *position* in a type.

Definition 10 (Subtree)**.** We define the subtree at position p of a tree type T, written $\mathsf{T}|_p$, as:

- $\mathsf{T}|_\varepsilon = \mathsf{T}$;
- $\mathbf{base}|_p$ is not defined when p is not ε;
- $\mathsf{r}\langle \mathsf{U} \rangle|_{rp} = \mathsf{U}|_p$;

- $\mathsf{w}\langle \mathsf{T} \rangle|_{wp} = \mathsf{T}|_p$;
- $\mathsf{rw}\langle \mathsf{U}, \mathsf{T} \rangle|_{rp} = \mathsf{U}|_p$;
- $\mathsf{rw}\langle \mathsf{U}, \mathsf{T} \rangle|_{wp} = \mathsf{T}|_p$;
- $(\mathsf{V}_1, \ldots, \mathsf{V}_n|_{ip} = \mathsf{V}_i|_p$;
- $\mathsf{loc}[\ldots, u : \mathsf{U}, \ldots]|_{up} = \mathsf{U}|_p$;

Definition 11. We say that a given position p in a tree type is *covariant* whenever there is an even number of "write" branches chosen (the number of w in p) along p. Otherwise we say that it is *contravariant*.

Lemma 12 (Correspondence of position in subtypes)**.** *Suppose that p is a shared position of two tree types T_1 and T_2 (namely $\mathsf{T}_1|_p$ and $\mathsf{T}_2|_p$ are defined). If $\mathsf{T}_1 <: \mathsf{T}_2$ then:*

- $\mathsf{T}_1|_p <: \mathsf{T}_2|_p$ *when p is a covariant position;*
- $\mathsf{T}_2|_p <: \mathsf{T}_1|_p$ *when p is a contravariant position.*

Proof. We prove this result by an induction on the length of p.

When $p = \varepsilon$, the result is direct.

Suppose that p is $p'x$ with $|x| = 1$. By our induction hypothesis, we know that

- $\mathsf{T}_1|_{p'} <: \mathsf{T}_2|_{p'}$ when p' is a covariant position.
- $\mathsf{T}_2|_{p'} <: \mathsf{T}_1|_{p'}$ when p' is a contravariant position.

Let us assume that p' is covariant. We reason on $\mathsf{T}_1|_{p'} <: \mathsf{T}_2|_{p'}$. Among the possible cases, we can have $\mathsf{T}_1 = \mathsf{r}\langle \mathsf{U}_1 \rangle$ and $\mathsf{T}_2 = \mathsf{r}\langle \mathsf{U}_2 \rangle$ which implies that x must be r, so p is covariant and that $\mathsf{U}_1 <: \mathsf{U}_2$. Whenever they are of the form, $\mathsf{T}_1 = \mathsf{rw}\langle \mathsf{U}'_1, \mathsf{T}'_1 \rangle$ and $\mathsf{T}_2 = \mathsf{w}\langle \mathsf{T}'_2 \rangle$ we can induce that x must w since the position p must be a common one. We also obtain that $\mathsf{T}'_2 <: \mathsf{T}'_1$, and that p is contravariant, which is the expected result.

Every other case is similar. □

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.

Theorem 13 (Partial meets)**.** *The set of tree types, ordered by <:, has partial meets. That is $\mathsf{T}_1 \downarrow \mathsf{T}_2$ implies T_1 and T_2 have a meet.*

Proof. This proof is performed in two stages: building a type candidate and checking that this candidate is the meet. The candidate is defined by positions: we write $\mathcal{P}_{\mathsf{T}_1 \sqcap \mathsf{T}_2}$ the set of positions p such that:

- $\mathsf{T}_1|_p$ and $\mathsf{T}_2|_p$ are defined;
- if p' is a covariant prefix of p, $\mathsf{T}_1|_{p'} \downarrow \mathsf{T}_2|_{p'}$;
- if p' is a contravariant prefix of p, $\mathsf{T}_1|_{p'} \uparrow \mathsf{T}_2|_{p'}$.

We write $\mathsf{T}_{\mathsf{T}_1 \sqcap \mathsf{T}_2}$ for the candidate and we define the construction at the position p for $p \in \mathcal{P}_{\mathsf{T}_1 \sqcap \mathsf{T}_2}$ by cases on the form of $\mathsf{T}_1|_p$ and $\mathsf{T}_2|_p$ and the variance of p. We consider here only typical cases:

$\mathsf{r}\langle \mathsf{T}_1^r \rangle$ and $\mathsf{r}\langle \mathsf{T}_2^r \rangle$ The node of $\mathsf{T}_{\mathsf{T}_1 \sqcap \mathsf{T}_2}$ at position p is defined to be $\mathsf{r}\langle \cdot \rangle$. Whether p is covariant or contravariant, pr is a common position of T_1 and T_2 and the condition of compatibility between the types $\mathsf{r}\langle \mathsf{T}_1^r \rangle$ and $\mathsf{r}\langle \mathsf{T}_2^r \rangle$ is kept between the types $\mathsf{r}\langle \mathsf{T}_1^r \rangle$ and $\mathsf{r}\langle \mathsf{T}_2^r \rangle$, so pr must be in $\mathcal{P}_{\mathsf{T}_1 \sqcap \mathsf{T}_2}$.

$\mathsf{rw}\langle \mathsf{T}_1^r, \mathsf{T}_1^w \rangle$ and $\mathsf{r}\langle \mathsf{T}_2^r \rangle$ when p is covariant The candidate must be of the form $\mathsf{rw}\langle \cdot, \mathsf{T}_1^w \rangle$ at position p; the compatibility between T_1 and T_2 ensures that T_1^r and T_2^r must also be compatible which implies that pr is in $\mathcal{P}_{\mathsf{T}_1 \sqcap \mathsf{T}_2}$.

$\mathsf{rw}\langle \mathsf{T}_1^r, \mathsf{T}_1^w \rangle$ and $\mathsf{r}\langle \mathsf{T}_2^r \rangle$ when p is contravariant The candidate must be of the form $\mathsf{r}\langle \cdot \rangle$ at position p and, as in the previous case, pr must be in $\mathcal{P}_{\mathsf{T}_1 \sqcap \mathsf{T}_2}$.

$\mathsf{loc}[\vec{u}^{(1)} : \vec{\mathsf{U}}^{(1)}, \vec{u}^{(2)} : \vec{\mathsf{U}}^{(2)}]$, $\mathsf{loc}[\vec{u}^{(1)} : \vec{\mathsf{U}}^{(3)}, \vec{u}^{(4)} : \vec{\mathsf{U}}^{(4)}]$ and p covariant We define $\mathsf{T}_{\mathsf{T}_1 \sqcap \mathsf{T}_2}|_p$ as $\mathsf{loc}[\vec{u}^{(1)} : \vec{\cdot}, \vec{u}^{(2)} : \vec{\mathsf{U}}^{(2)}, \vec{u}^{(4)} : \vec{\mathsf{U}}^{(4)}]$. Notice that $pu_i^{(1)}$ must be in $\mathcal{P}_{\mathsf{T}_1 \sqcap \mathsf{T}_2}$ for all i.

$\mathsf{loc}[\vec{u}^{(1)} : \vec{\mathsf{U}}^{(1)}, \vec{u}^{(2)} : \vec{\mathsf{U}}^{(2)}]$, $\mathsf{loc}[\vec{u}^{(1)} : \vec{\mathsf{U}}^{(3)}, \vec{u}^{(4)} : \vec{\mathsf{U}}^{(4)}]$, p contravariant We define $\mathsf{T}_{\mathsf{T}_1 \sqcap \mathsf{T}_2}|_p$ as $\mathsf{loc}[\vec{u}^{(1)} : \vec{\cdot}]$. Notice that $pu_i^{(1)}$ must be in $\mathcal{P}_{\mathsf{T}_1 \sqcap \mathsf{T}_2}$ for all i.

$\mathsf{T}_{\mathsf{T}_1 \sqcap \mathsf{T}_2}$ is thus defined at all the positions in $\mathcal{P}_{\mathsf{T}_1 \sqcap \mathsf{T}_2}$. We remark that $\mathcal{P}_{\mathsf{T}_1 \sqcap \mathsf{T}_2}$ is closed by prefix and that all the nodes are given a type construction. Moreover, for each child expected by such a node, we have either directly defined it or checked that its position was in $\mathcal{P}_{\mathsf{T}_1 \sqcap \mathsf{T}_2}$. These different cases therefore define a tree pre-type.

To ensure that this is indeed a type, we must check that every $\mathsf{rw}\langle \mathsf{U}, \mathsf{T} \rangle$ it contains are such that $\mathsf{T} <: \mathsf{U}$. For every $\mathsf{rw}\langle \mathsf{U}, \mathsf{T} \rangle$ appearing in one part of $\mathsf{T}_{\mathsf{T}_1 \sqcap \mathsf{T}_2}$ directly coming from either T_1 or T_2, the result is obvious. Otherwise, we define the relation $\mathcal{R}$ thus:

$$
\begin{aligned}
\mathcal{R} = \; & \{(\mathsf{T}, \mathsf{T}_{\mathsf{T}_1 \sqcap \mathsf{T}_2}|_p), (\mathsf{T}_{\mathsf{T}_1 \sqcap \mathsf{T}_2}|_p \,, \mathsf{T}_1|_p), (\mathsf{T}_{\mathsf{T}_1 \sqcap \mathsf{T}_2}|_p \,, \mathsf{T}_2|_p) \\
& \quad \mid \; p \in \mathcal{P}_{\mathsf{T}_1 \sqcap \mathsf{T}_2} \text{ covariant, } \mathsf{T} \text{ type and subtype of } \mathsf{T}_1|_p \text{ and } \mathsf{T}_2|_p\} \\
\cup \; & \{(\mathsf{T}_{\mathsf{T}_1 \sqcap \mathsf{T}_2}|_p \,, \mathsf{T}), (\mathsf{T}_1|_p \,, \mathsf{T}_{\mathsf{T}_1 \sqcap \mathsf{T}_2}|_p), (\mathsf{T}_2|_p \,, \mathsf{T}_{\mathsf{T}_1 \sqcap \mathsf{T}_2}|_p) \\
& \quad \mid \; p \in \mathcal{P}_{\mathsf{T}_1 \sqcap \mathsf{T}_2} \text{ contravariant, } \mathsf{T} \text{ type, supertype of } \mathsf{T}_1|_p \text{ and } \mathsf{T}_2|_p\} \\
\cup \; & \nu\mathsf{Sub}
\end{aligned}
$$

We prove that $\mathcal{R} \subseteq \nu\mathsf{Sub}$ by showing that $\mathcal{R} \subseteq \mathsf{Sub}(\mathcal{R}^+)$ where $\mathcal{R}^+$ is the transitive closure of $\mathcal{R}$. From this it will be easy to conclude that $\mathcal{R}^+ \subseteq \mathsf{Sub}(\mathcal{R}^+)$ and, consequently, that $\mathcal{R} \subseteq \nu\mathsf{Sub}$.

Let us consider some $(\mathsf{T}, \mathsf{T}_{\mathsf{T}_1 \sqcap \mathsf{T}_2}|_p)$, $(\mathsf{T}_{\mathsf{T}_1 \sqcap \mathsf{T}_2}|_p \,, \mathsf{T}_1|_p)$ and $(\mathsf{T}_{\mathsf{T}_1 \sqcap \mathsf{T}_2}|_p \,, \mathsf{T}_2|_p)$ in $\mathcal{R}$. We reason on the forms of $\mathsf{T}_1|_p$ and $\mathsf{T}_2|_p$, exactly as in the definition of $\mathsf{T}_{\mathsf{T}_1 \sqcap \mathsf{T}_2}|_p$.

Let us study some typical example: $\mathsf{T}_1|_p = \mathsf{rw}\langle \mathsf{T}_1^r, \mathsf{T}_1^w\rangle$ and $\mathsf{T}_2|_p = \mathsf{r}\langle \mathsf{T}_2^r\rangle$ when p is covariant0. $\mathsf{T}_{\mathsf{T}_1\sqcap\mathsf{T}_2}|_p$ must have the form $\mathsf{rw}\langle\cdot, \mathsf{T}_1^w\rangle$ and pr must be in the set $\mathcal{P}_{\mathsf{T}_1\sqcap\mathsf{T}_2}$. Since T is a common subtype of $\mathsf{T}_1|_p$ and $\mathsf{T}_2|_p$, it must be of the form $\mathsf{rw}\langle \mathsf{T}^r, \mathsf{T}^w\rangle$ with $\mathsf{T}^r <: \mathsf{T}_1^r$, $\mathsf{T}^r <: \mathsf{T}_2^r$ and $\mathsf{T}_1^w <: \mathsf{T}^w$. So we can conclude that $(\mathsf{T}^r, \mathsf{T}_{\mathsf{T}_1\sqcap\mathsf{T}_2}|_{pr})$ is in $\mathcal{R}$, from which we deduce that $(\mathsf{T}, \mathsf{T}_{\mathsf{T}_1\sqcap\mathsf{T}_2}|_p)$ is in $\mathsf{Sub}(\mathcal{R}^+)$. To get that $(\mathsf{T}_{\mathsf{T}_1\sqcap\mathsf{T}_2}|_p, \mathsf{T}_1)$ is in $\mathsf{Sub}(\mathcal{R}^+)$, we need to have the three pairs $(\mathsf{T}_1|_{pw}, \mathsf{T}_{\mathsf{T}_1\sqcap\mathsf{T}_2}|_{pw}) = (\mathsf{T}_1^w, \mathsf{T}_1^w)$, $(\mathsf{T}_{\mathsf{T}_1\sqcap\mathsf{T}_2}|_{pw}, \mathsf{T}_{\mathsf{T}_1\sqcap\mathsf{T}_2}|_{pr})$ and $(\mathsf{T}_{\mathsf{T}_1\sqcap\mathsf{T}_2}|_{pr}, \mathsf{T}_1^r)$ in $\mathcal{R}^+$. The first is in $\nu\mathsf{Sub}$. The second comes from T common subtype of T_1 and T_2: $\mathsf{T}_1^w <: \mathsf{T}^w <: \mathsf{T}^r\ \mathcal{R}\ \mathsf{T}_{\mathsf{T}_1\sqcap\mathsf{T}_2}|_{pr})$ so $\mathsf{T}_1^w\ \mathcal{R}^+\ \mathsf{T}_{\mathsf{T}_1\sqcap\mathsf{T}_2}|_{pr})$. Finally, the third comes from the fact that pr must be in $\mathcal{P}_{\mathsf{T}_1\sqcap\mathsf{T}_2}$. The reasoning to prove that $(\mathsf{T}_{\mathsf{T}_1\sqcap\mathsf{T}_2}|_p, \mathsf{T}_2|_p)$ is also in $\mathsf{Sub}(\mathcal{R}^+)$ is similar.

The fact that $\mathcal{R}$ is included in $\nu\mathsf{Sub}$ entails that for every p such that $\mathsf{T}_{\mathsf{T}_1\sqcap\mathsf{T}_2}|_p$ is of the form $\mathsf{rw}\langle\cdot,\cdot\rangle$, we know some subtype, either some T or some $\mathsf{T}_1|_p$, depending on the variance of p.

Finally, we need to show that $\mathsf{T}_{\mathsf{T}_1\sqcap\mathsf{T}_2}$ is indeed the meet of T_1 and T_2. We obtain this result by considering the relation $\mathcal{R}$ defined by:

$$\begin{aligned}\mathcal{R} = &\ \{\sqcap(\mathsf{T}_1|_p, \mathsf{T}_2|_p, \mathsf{T}_{\mathsf{T}_1\sqcap\mathsf{T}_2}|_p) \mid\ p \in \mathcal{P}_{\mathsf{T}_1\sqcap\mathsf{T}_2} \text{ is covariant}\}\\ \cup &\ \{\sqcup(\mathsf{T}_1|_p, \mathsf{T}_2|_p, \mathsf{T}_{\mathsf{T}_1\sqcap\mathsf{T}_2}|_p) \mid\ p \in \mathcal{P}_{\mathsf{T}_1\sqcap\mathsf{T}_2} \text{ is contravariant}\}\end{aligned}$$

and proving that this is a post-fixpoint of $\mathsf{MeetJoin}$. This verification is automatic from the definition of $\mathsf{T}_{\mathsf{T}_1\sqcap\mathsf{T}_2}$. □

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 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.

Fig. 5 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-TUPLE)

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

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

$$\begin{aligned}
\mathsf{SubTerms}(\mathbf{base}) &= \{\mathbf{base}\} \\
\mathsf{SubTerms}(\mathrm{R}\langle \mathsf{U}_0 \rangle) &= \{\mathrm{R}\langle \mathsf{U}_0 \rangle\} \cup \mathsf{SubTerms}(\mathsf{U}_0) \\
\mathsf{SubTerms}(\mathrm{W}\langle \mathsf{T}_0 \rangle) &= \{\mathrm{W}\langle \mathsf{T}_0 \rangle\} \cup \mathsf{SubTerms}(\mathsf{T}_0) \\
\mathsf{SubTerms}(\mathrm{RW}\langle \mathsf{U}_0, \mathsf{T}_0 \rangle) &= \{\mathrm{RW}\langle \mathsf{U}_0, \mathsf{T}_0 \rangle\} \cup \mathsf{SubTerms}(\mathsf{U}_0) \cup \mathsf{SubTerms}(\mathsf{T}_0) \\
\mathsf{SubTerms}(\mathrm{LOC}[u_i : \mathsf{A}_i]) &= \{\mathrm{LOC}[u_i : \mathsf{A}_i]\} \cup \bigcup_i \mathsf{SubTerms}(\mathsf{A}_i) \\
\mathsf{SubTerms}(\mu \mathbf{Y}.\mathsf{K}) &= \{\mu \mathbf{Y}.\mathsf{K}\} \cup \mathsf{SubTerms}(\mathsf{K}\{^{\mu \mathbf{Y}.\mathsf{K}}/_{\mathbf{Y}}\}) \\
\mathsf{SubTerms}(\mathsf{A}@\mathsf{K}) &= \{\mathsf{A}@\mathsf{K}\} \cup \mathsf{SubTerms}(\mathsf{A}) \cup \mathsf{SubTerms}(\mathsf{K}) \\
\mathsf{SubTerms}((\widetilde{\mathsf{C}})) &= \{(\widetilde{\mathsf{C}})\} \cup \bigcup_i \mathsf{SubTerms}(\mathsf{C}_i)
\end{aligned}$$

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 to the set.

Fig. 6 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-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)

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

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 for when a given statement has already been seen. The rules we obtain look like the ones, in [6], 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 6 and 7. 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.

Fig. 7 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-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)

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

(JOIN-HOM)

$$\frac{\begin{array}{l} \Sigma \vdash \mathsf{A}_1 \sqcup \mathsf{A}_2 = \mathsf{A}_3 \\ \Sigma \vdash \mathsf{K}_1 \sqcup \mathsf{K}_2 = \mathsf{K}_3 \end{array}}{\Sigma \vdash \mathsf{A}_1 @ \mathsf{K}_1 \sqcup \mathsf{A}_2 @ \mathsf{K}_2 = \mathsf{A}_3 @ \mathsf{K}_3}$$

(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)

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

Proposition 15 (Recursive subtyping is tree subtyping)**.** *For any two recursive types* T_1 *and* T_2*,* $\mathit{Tree}(\mathsf{T}_1) <: \mathit{Tree}(\mathsf{T}_2)$ *if and only if* $\emptyset \vdash \mathsf{T}_1 <: \mathsf{T}_2$*.*

Proposition 16 (Recursive type meet is infinite tree meet)**.** *For any three recursive types* T_1*,* T_2 *and* T_3*,* $\mathit{Tree}(\mathsf{T}_1) \sqcap \mathit{Tree}(\mathsf{T}_2) = \mathit{Tree}(\mathsf{T}_3)$ *if and only if* $\emptyset \vdash \mathsf{T}_1 \sqcap \mathsf{T}_2 = \mathsf{T}_3$*.*

The proofs for these two propositions are really similar, as one would expect, so we give only the proof of the second one, which is 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. Since recursive types are unfold, the sum of sizes of the types T'_i, written $|\mathsf{T}'_1| + |\mathsf{T}'_2| + |\mathsf{T}'_3|$, is not decreasing. But some subterms of T_i are added to Σ whenever unfolding is performed. So, since the number of subterms of T_i, written $|\mathsf{SubTerms}(\mathsf{T}_i)|$ is constant, the number of triples composed of subterms of the T_i that do not appear in Σ, namely $|\mathsf{SubTerms}(\mathsf{T}_1)| \times |\mathsf{SubTerms}(\mathsf{T}_2)| \times |\mathsf{SubTerms}(\mathsf{T}_3)| - |\Sigma'|$, will decrease. So the lexicographic order

$$(|\mathsf{SubTerms}(\mathsf{T}_1)| \times |\mathsf{SubTerms}(\mathsf{T}_2)| \times |\mathsf{SubTerms}(\mathsf{T}_3)| - |\Sigma'|, |\mathsf{T}'_1| + |\mathsf{T}'_2| + |\mathsf{T}'_3|)$$

will guarantee that the induction terminates.

- 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 $|\mathsf{SubTerms}(\mathsf{T}_1)| \times |\mathsf{SubTerms}(\mathsf{T}_2)| \times |\mathsf{SubTerms}(\mathsf{T}_3)| - |\Sigma', \mathsf{T}'_1 \sqcap \mathsf{T}'_2 = \mathsf{T}'_3|$ 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 6 and 7. 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

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

Let us prove that this relation $\mathcal{R}$ is a postfixpoint of 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 $\emptyset \vdash \mathsf{T}_1 \sqcap \mathsf{T}_2 = \mathsf{T}_3$.

Fig. 8 Well-formed environments

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

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

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

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

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

- (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$[\ldots]$ 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, among other things because we use a notion of equality over tree types instead of an equivalence that a fixpoint operator would imply. This way, 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.

Fig. 9 Typing values

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

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

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

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

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

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

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

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

Systems and processes typechecking will be performed in typing environments. These typing environments are lists of elements of the form $u : \mathsf{T}$ with u a name or a variable (in particular it might be 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 located at w. We put some extra constraints on those lists to ensure the well-formedness of the environments. The rules of formation for environments are given in Figure 8: we will write $\Gamma \vdash \mathbf{env}$ whenever Γ is well-formed.

The main idea of those rules is that an environment is well-formed as soon as the type at which a name or a variable is added to the environment is compatible with all the types already associated with that name or variable, namely the meet of all those types. This is useful for the names received during communications: if you get some name at two different types (through communication on two different channels), the environment obtained should be equivalent to the one in which that name is given those two types, namely we will be able to prove the same statements in the environment containing only $a@k : \mathrm{RW}\langle\rangle$ and in the one containing separately the two capabilities $a@k : \mathrm{R}\langle\rangle, a@k : \mathrm{W}\langle\rangle$.

Another important idea is the way recursion variables are dealt with in that setting: we want that the location type given to a recursion variable be such that the recursive process can safely run in any location of that type. This entails that, during static typechecking, that type will be useful at every occurrence of a recursive call. For this, environments associate a full location type, namely a type of the form $\mathrm{LOC}[a_1 : \mathsf{C}_1, \ldots]$, to recursion variables whereas they

Fig. 10 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)
$$\frac{\Gamma \vdash_w u : \mathsf{U}, v : \mathsf{V} \quad \Gamma \vdash_w Q \quad \text{and, when } \Gamma, \langle u : \mathsf{V} \rangle \,@\, w, \langle v : \mathsf{U} \rangle \,@\, w \vdash \mathbf{env},\ \Gamma, \langle u : \mathsf{V} \rangle \,@\, w, \langle v : \mathsf{U} \rangle \,@\, w \vdash_w P}{\Gamma \vdash_w \mathsf{if}\ u = v\ \mathsf{then}\ P\ \mathsf{else}\ Q}$$

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

Fig. 11 Typing Systems

(T-NIL)
$$\frac{\Gamma \vdash \mathbf{env}}{\Gamma \vdash \mathbf{0}}$$

(T-PAR)
$$\frac{\Gamma \vdash M \quad \Gamma \vdash N}{\Gamma \vdash M \mid N}$$

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

(T-NEW)
$$\frac{\Gamma, \langle n : \mathsf{N} \rangle \vdash M}{\Gamma \vdash (\mathsf{new}\, n : \mathsf{N})\, M}$$

associate the simple type LOC to every "real" location names and variables. This also constrains the subtyping of environments.

Indeed, subtyping on types is naturally extended to environments so that we say $\Gamma <: \Gamma'$ 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 $u@w$.

Note that no subtyping is performed on recursion variable types. Otherwise, it would break the main property of subtyping: when some environment is used, some subtype of this environment might be used instead. Note also that the definition of well-formed environment forces recursion variables to appear at most once. Since subtyping is disallowed, it would be meaningless. This coincides with the fact that recursion variables are introduced into the environment only when type-checking a process of the form $\mathsf{rec}\ Z : \mathsf{R}.\ P$.

Now that we have formally defined well-formed environments, we can successively give the typing rules for values, in Figure 9, processes, in Figure 10, and for whole systems, in Figure 11. Many of those rules are inherited from standard DPI ones, so the reader is referred to [3] for more complete explanations of them. We explain here only the basics of those typing rules and the new rules we add to accommodate for the recursion construct.

At the system level the judgements take the form $\Gamma \vdash M$, stating that all the free names of the system M are defined in Γ and that they are used according to the types they are given in that environment. The main rule in Figure 11 is

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

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. Note that the rule (T-NEW) for the typing of systems of the form $(\mathsf{new}\, n : \mathsf{N})\, M$ can be used for both channels, when n is some $c@k$, and locations, where $n = k$.

At the process level, we have to add some typing rules for the new constructs we introduced in the language, namely recursions, recursion variables and location look-ups. The latter is straightforward:

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

However, for recursive processes like $\mathsf{rec}\ Z : \mathsf{R}.\ P$, the rules are trickier. As stated previously, we want the type R to characterise the requirements on a given location to run the recursive process. Then the natural rule for typing-checking a recursive call, that is an occurrence of a recursion variable, is given by:

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

The recursive calls will then be safe as soon as we can typecheck the process P in a location with exactly the set of resources indicated by R available. Pursuing with the analogy of attributing location types to recursion variables, our idea is to simply use the recursion variable itself as the location in which the process will be typed. Then the rule for typing recursive processes is

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

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 actual terms, this being syntactically prohibited.

Also note that recursion variables are considered exactly as locations as far as value typing is concerned. In particular, notice that value typing rules allow statements of the form $\Gamma \vdash Z : \textsc{loc}$. This is required when typing a process "at Z".

Example 17. Referring back to Example 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

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

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

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

we get

$$\Gamma, Z : \mathsf{S}, \mathit{test}_{@}Z : \textsc{r}\langle \mathsf{T}_t \rangle, \mathit{neigh}_{@}Z : \textsc{r}\langle \mathsf{S} \rangle \vdash_Z \mathit{test} : \textsc{r}\langle \mathsf{T}_t \rangle$$

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

Lemma 18 (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 process Q, 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 $\Gamma \vdash_v P\{^Q/_Z\}$.
- (T-INPUT) so $P = u\,?\,(X : \mathsf{T})\, P'$ and $\Gamma, \langle X : \mathsf{T} \rangle_{@}v \vdash_v P'$. Since $\Gamma, \langle X : \mathsf{T} \rangle_{@}v$ is a subtype of Γ, we know that for any Γ' and any w such that $\Gamma' <: \Gamma, \langle X : \mathsf{T} \rangle_{@}v$ and $\Gamma' \vdash w : \Gamma'(Z)$, we will have $\Gamma' \vdash_w Q$. So we can apply the induction hypothesis to conclude.

- (T-MATCH) which implies that $P = \mathsf{if}\ u = u'\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2$ and that we have the following hypotheses: $\Gamma \vdash_v u : \mathsf{U}, u' : \mathsf{U}'$, $\Gamma \vdash_v P_2$ and, if $\Gamma,\langle u : \mathsf{U}'\rangle \circledast v,\langle u' : \mathsf{U}\rangle \circledast v \vdash \mathbf{env}$, $\Gamma,\langle u : \mathsf{U}'\rangle \circledast v,\langle u' : \mathsf{U}\rangle \circledast v \vdash_v P_1$. Then, by our induction hypothesis, we know that $\Gamma \vdash_v P_2\{^Q/_Z\}$. And since the environment $\Gamma,\langle u : \mathsf{U}'\rangle \circledast v,\langle u' : \mathsf{U}\rangle \circledast v$ is a subtype of Γ then $\Gamma,\langle u : \mathsf{U}'\rangle \circledast v,\langle u' : \mathsf{U}\rangle \circledast v \vdash_v P_1\{^Q/_Z\}$.
- (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 19 (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

As in the PI-CALCULUS, the replicated process $*P$ can be encoded with recursion by $\mathsf{rec}\ Z : \mathsf{R}.\ (Z \mid P)$, for some type R (the type associated with the location where P is located will do). But the converse is trickier to obtain.

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

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

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

where

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

With this example, we can easily see how the translation will mimic the original process step by step: the body of the process is left unmodified, only the recursion parts are changed, by implementing the recursive call with a few reductions. FireOne is the "translation" for the recursive calls, which means going to the home base and firing a new instance. This uses the construct here to express that action neatly: the translation for recursive calls needs to report its current location to indeed trigger the new instance in the "proper" context. When that actual location is obtained, the replicated IterSearch starts off by migrating there. As we already mentioned, it would be possible to encode the construct here by annotating the translation of a process by its location, either the location l for a process $l[\![P]\!]$ or the location used in the previous goto.

This approach underlies our general translation of recursive processes into iterative processes, which we now explain. The main characteristic of the approach we propose here is its *compositionality*: this means that a recursive process can be translated into a replicated process independently of the rest of the "universe", namely the context in which it is placed. This translation will thus be applicable on partial systems as well as full systems. To obtain that compositionality, 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 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.

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

We stress here two facts:

- the translated processes still require recursive types, for instance for the channel ping_Z: the full theory of recursive types developed in the previous sections deals with *recursive behaviours*, even when they are expressed using replication;
- 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 [7], 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 [1] 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:

- $\textsc{nc-unrec}(Z_i) = \mathsf{here}\,[x]\,\mathsf{goto}\ \mathit{home}_{Z_i}.\ \mathit{ping}_{Z_i}\,!\,\langle x, \tilde{n}_i \rangle$
- $\textsc{nc-unrec}(\mathsf{rec}\ Z_i : \mathsf{R}_i.\ P_i) = \textsc{nc-unrec}(Z_i)$
- $\textsc{nc-unrec}(u\,!\,\langle V \rangle\ P) = u\,!\,\langle V \rangle\ \textsc{nc-unrec}(P)$; all the other cases are similar.

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

$$\begin{array}{l}(\mathsf{new}\, home_{Z_1})\,(\mathsf{new}\, ping_{Z_1})\,(\mathsf{new}\, home_{Z_2})\,(\mathsf{new}\, ping_{Z_2})\ldots \\ \quad home_{Z_1}[\![*ping_{Z_1}\,?\,(l:\mathsf{R}_1,\tilde{n}_1)\,\mathsf{goto}\,l.\ \textsc{nc-unrec}(P_1)]\!]\mid \\ \quad home_{Z_2}[\![*ping_{Z_2}\,?\,(l:\mathsf{R}_2,\tilde{n}_2)\,\mathsf{goto}\,l.\ \textsc{nc-unrec}(P_2)]\!]\mid\ldots\mid \\ \quad \textsc{nc-unrec}(M)\end{array}$$

But, of course, such an approach would not be compositional, as the name $\textsc{nc-unrec}(\cdot)$ 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 20. $\Gamma \vdash M$ *if and only if* $\Gamma \vdash \textsc{unrec}(M)$

Proof. We refrain from burdening the reader with the full proof. Indeed it proceeds simply by translating the proof of $\Gamma \vdash M$ into a proof of $\Gamma \vdash \textsc{unrec}(M)$ and back. For this, remark that the translation of the proofs will rely on a translation of the environments appearing in the proofs and that those environments might contain open recursion variables. So we define a function φ that will perform the translation of the environments by introducing a fresh name l_{Z_i} for every recursion variable Z_i. This will accomodate for the fact that the body of recursive process named Z_i is typed "at Z_i", whereas it will be type-checked in the location name bound by a communication on the channel $ping_{Z_i}$.

- $\varphi(\Gamma, u_{i_j} @ Z_i : \mathsf{A}_{i_j}) = \varphi(\Gamma), u_{i_j} @ l_{Z_i} : \mathsf{A}_{i_j}$;
- $\varphi(\Gamma, Z_i : \mathsf{R}_i) = \varphi(\Gamma), home_{Z_i} : \textsc{loc}, ping_{Z_i} @ home_{Z_i} : \textsc{rw}\langle \mathsf{R}_i \rangle, l_{Z_i} : \textsc{loc}$;
- $\varphi(\Gamma, u : \mathsf{A}) = \varphi(\Gamma), u : \mathsf{A}$ whenever none of the previous cases apply.

and we proceed with proving that $\Gamma \vdash M$ implies $\varphi(\Gamma) \vdash \textsc{unrec}(M)$, by a simple induction on the proof, so by proving the corresponding result on value and process typing.

The converse translation can be performed similarly: the only difference is to associate the type R_i corresponding to Z_i in M when translating back l_{Z_i}. □

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

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

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.

Definition 21 (Configurations)**.** We call *configuration* a pair composed of an environment Ω and a closed system M, written $\Omega \rhd M$, such that there exists an environment Γ, with $\Gamma <: \Omega$ and $\Gamma \vdash M$.

Theorem 22. *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 12 and 13.

The rules given in the Figures 12 and 13 are mostly reformulation of the LTS inherited from [3]. In those figures, some transitions are written $\xrightarrow{\tau}_{\beta}$: the "$\beta$" annotation will be explained on page 40 and can be ignored for now. The main rule is (LTS-COMM) which describes how two processes can communicate: this supposes that one of the processes involved is writing while the other is reading on the same channel. Since only those two processes are involved in that communication, the observer is learning nothing when it is performed: that is why the environment Ω is left unmodified while this happens. That is also why the environments in which those two processes perform their actions are different from Ω: they can communicate together even if the overall observer is not able to interact with them on that channel.

Definition 23 (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 [3] for the following result:

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

Fig. 12 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)
$\Omega \rhd k[\![(\mathsf{newloc}\, l : \mathsf{L})\ P]\!] \xrightarrow{\tau}_\beta \Omega \rhd (\mathsf{new}\, \langle l : \mathsf{L}\rangle)\, k[\![P]\!]$

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

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

So we establish Theorem 22 by showing

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

6 Proof of recursion implementability

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

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

used in the system $\mathsf{Server}[\![*\mathsf{PSearch}]\!]$. So this sets up a search server, at Server; but the difference with Search from Example 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.

Fig. 13 Labelled transition semantics. External actions.

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

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

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

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

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

Our translation of this process gives the following DPI code:

$$\begin{aligned} \mathsf{IPSearch} &\triangleq \mathit{search\_req}\,?\,(x, \mathit{client}) \\ &\quad \mathsf{goto}\, k_0.\ (\mathsf{newloc}\, \mathit{base} : \textsc{loc}[\mathit{ping}])\, \mathsf{F} \mid \mathsf{goto}\, \mathit{base}.\ * \mathsf{Inst} \\ \mathsf{Inst} &\triangleq \mathit{ping}\,?\,(k)\, \mathsf{goto}\, k.\ \mathit{test}\,?\,(y) \\ &\quad\ \mathsf{if}\ p(x, y)\ \mathsf{then}\ \mathsf{goto}\, \mathit{client}.\ \mathit{report}\,!\,\langle y \rangle \\ &\quad\ \mathsf{else}\ \mathit{neigh}\,?\,(n_1, n_2)\, \mathsf{goto}\, n_1.\ \mathsf{F} \mid \mathsf{goto}\, n_2.\ \mathsf{F} \\ \mathsf{F} &\triangleq \mathsf{here}\,[l]\, \mathsf{goto}\, \mathit{base}.\ \mathit{ping}\,!\,\langle l \rangle \end{aligned}$$

with Inst an instance of the iterative process, and F the triggering process, written FireOne in 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:

$$\begin{array}{l}(\mathsf{new}\ base_1)\,(\mathsf{new}\ ping_1)\,(\mathsf{new}\ base_2)\,(\mathsf{new}\ ping_2)\ldots\\ \quad \mathsf{Server}[\![\ldots]\!]\ldots\\ \quad \mid base_1[\![\ldots]\!] \mid k_1^1[\![\ldots \mathsf{F}_1]\!] \mid k_1^2[\![\ldots \mathsf{F}_1]\!] \mid \ldots\\ \quad \mid base_2[\![\ldots]\!] \mid k_2^1[\![\ldots \mathsf{F}_2]\!]\ldots\end{array} \tag{3}$$

Of course, this will correspond to the recDpi system:

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

On this example, we can see quite clearly the main difference at runtime between our translation and the standard but non-compositional one outlined above on page 32. A translation following that non-compositional approach would give rise to the following state, corresponding to (3) above:

$$\begin{array}{l}(\mathsf{new}\ base)\,(\mathsf{new}\ ping)\,\mathsf{Server}[\![*search\_req\,?\,(x, client)\ \mathsf{goto}\ k_0.\ \mathsf{F}(x, client)]\!]\\ \quad \mid base[\![*ping\,?\,(k, x, client)\ \mathsf{goto}\ k.\ test\,?\,(y)\ldots]\!]\\ \quad \mid k_1^1[\![\ldots \mathsf{F}(x_1, client_1)]\!] \mid k_1^2[\![\ldots \mathsf{F}(x_1, client_1)]\!] \mid \ldots\\ \quad \mid k_2^1[\![\ldots \mathsf{F}(x_2, client_2)]\!]\ldots\end{array}$$

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

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 would allow an easier proof of equivalence, since there is only one *base* per recursion variable.

To return to the discussion of compositional translation, we have a recDpi process containing a number of recursive constructs but the way they are to be translated to get the Dpi system (3) depends on the system history. That is why our proof of (2) is based on an extended version of the translation in which we specify whether a given occurrence of $\mathsf{rec}\ Z.\ P$ 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. This parameterisation identifies each occurrence of $\mathsf{rec}\ Z.\ P$ by its position in the system or in the process, in a similar way to positions of subtrees in types (Definition 10).

Definition 24 (Occurrence)**.** The occurrence o in a process P or a system M, written $P|_o$ and $M|_o$ is defined inductively by:

- $P|_\varepsilon = P,\ M|_\varepsilon = M$;
- $(P_1 \mid P_2)|_{1p} = P_1|_p,\ (P_1 \mid P_2)|_{2p} = P_2|_p,\ (M_1 \mid M_2)|_{1p} = M_1|_p,\ (M_1 \mid M_2)|_{2p} = M_2|_p$;
- $(\mathsf{if}\ u_1 = u_2\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2)|_{1p} = P|_p,\ (\mathsf{if}\ u_1 = u_2\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2)|_{2p} = P|_p$;
- every other constructor has only one sub-component so: $l[\![P]\!]|_{0p} = P|_p$, $((\mathsf{new}\, a : \mathsf{E})\, M)|_{0p} = M|_p,\ (u\,!\,\langle V\rangle\, P)|_{0p} = P|_p,\ (u\,?\,(X : \mathsf{T})\, P)|_{0p} = P|_p$, etc.

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 occurrences in M' of the system or process at o in M, if it still exists. This is needed to keep track of the recursive processes that are already translated. For instance, in the system $l[\![\mathsf{goto}\, k.\ \mathsf{rec}\ Z.\ P]\!]$, when the prefix goto is reduced, the occurrence of $\mathsf{rec}\ Z.\ P$ moves from the position 00 to the position 0 in $k[\![\mathsf{rec}\ Z.\ P]\!]$. The residual is obviously more complex when recursive processes are unfold: since recursion variables are replaced by a full process, there can be more than one residual for a given occurrence in the initial process. The residual of those occurrences is therefore computed using the position of the recursive calls in the initial process.

Definition 25 (Residual)**.** We call *residual* of an occurrence o in M after a transition $\Omega \rhd M \xrightarrow{\mu} \Omega' \rhd M'$ the set of occurrences defined by the following function:

- $\mathsf{Res}(\varepsilon, \Omega \rhd M \xrightarrow{\mu} \Omega' \rhd M') = \{\varepsilon\}$
- $\mathsf{Res}(1o, \Omega \rhd M \mid N \xrightarrow{\mu} \Omega' \rhd M' \mid N) = \mathsf{Res}(o, \Omega \rhd M \xrightarrow{\mu} \Omega' \rhd M')$
- $\mathsf{Res}(2o, \Omega \rhd M \mid N \xrightarrow{\mu} \Omega' \rhd M' \mid N) = \{2o\}$
- $\mathsf{Res}(0, \Omega \rhd k[\![a\,!\,\langle V\rangle\, P]\!] \xrightarrow{k.a!V} \Omega' \rhd k[\![P]\!]) = \emptyset$
- $\mathsf{Res}(00o, \Omega \rhd k[\![a\,!\,\langle V\rangle\, P]\!] \xrightarrow{k.a!V} \Omega' \rhd k[\![P]\!]) = \{0o\}$
- $\mathsf{Res}(0o, \Omega \rhd (\mathsf{new}\, n : \mathsf{T})\, M \xrightarrow{(n\tilde{n}:\mathsf{T}\tilde{\mathsf{T}})k.a!V} \Omega' \rhd M')$
  $= \mathsf{Res}(o, \Omega \rhd M \xrightarrow{(\tilde{n}:\tilde{\mathsf{T}})k.a!V} \Omega' \rhd M')$
- $\mathsf{Res}(0o, \Omega \rhd (\mathsf{new}\, n : \mathsf{T})\, M \xrightarrow{\mu} \Omega' \rhd (\mathsf{new}\, n : \mathsf{T})\, M')$
  $= \{0o \mid o \in \mathsf{Res}(o, \Omega \rhd M \xrightarrow{\mu} \Omega' \rhd M')\}$
- $\mathsf{Rés}(0, \Omega \rhd l[\![\mathsf{rec}\ (Z : \mathsf{R}).\ P]\!] \xrightarrow{\tau} \Omega \rhd l[\![P[^{\mathsf{rec}\ (Z:\mathsf{R}).\ P}/_Z]]\!]) = \{0o \mid \forall o,\ P|_o = Z\}$
- $\mathsf{Rés}(00o, \Omega \rhd l[\![\mathsf{rec}\ (Z : \mathsf{R}).\ P]\!] \xrightarrow{\tau} \Omega \rhd l[\![P[^{\mathsf{rec}\ (Z:\mathsf{R}).\ P}/_Z]]\!])$
  $= \{0o, 0o'0o \mid \forall o',\ P|_{o'} = Z\}$
- $\mathsf{Rés}(1o, \Omega \rhd M \mid N \xrightarrow{\tau} \Omega \rhd (\mathsf{new}\, \tilde{n} : \tilde{\mathsf{T}})\, M' \mid N') = \{10^m o' \mid \forall o' \in \mathsf{Rés}(o, \Omega_M \rhd M \xrightarrow{\mu} \Omega'_M \rhd M')\}$ where m is the number of names in $\tilde{n}$ and μ is either a reading or a writing

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 will write $O^{(n_O)}$ for one of the sets in that partition, annotated by n_O. We define the "valid" $\mathcal{P}$s as:

- if there exists o_1, o_2 and O such that $o_i \in O^{(n_O)} \in \mathcal{P}$ then $M|_{o_1} = M|_{o_2} = \mathsf{rec}\ Z : \mathsf{R}.\ P$;
- for any $O^{(n_O)} \in \mathcal{P}$, we call o the longest common system-prefix of all occurrences in $O^{(n_O)}$; if we write $\mathsf{rec}\ Z.\ P$ for the recursive process corresponding to O then all the free names in P are either free in the whole system or bound at an occurrence that is a prefix of o; moreover P contains no free variable.

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 $\mathcal{P}$,

$$\begin{aligned}\textsc{unrec}^o_{\mathcal{P}}(k[\![\ \mathsf{rec}\ Z : \mathsf{R}.\ P]\!]) = \\ &(\mathsf{new}\left\langle home_{Z_{\{o0\}}} : \textsc{loc}[ping_{Z_{\{o0\}}} : \textsc{rw}\langle \mathsf{R}_{\{o0\}}\rangle]\right\rangle) \\ &home_{Z_{\{o0\}}}[\![*ping_{Z_{\{o0\}}}\,?\,(l : \mathsf{R})\,\mathsf{goto}\,l.\ \textsc{unrec}^{o00}_{\mathcal{P}\cup\{\{o0\}^0\}}(P)]\!] \\ |\ &home_{Z_{\{o0\}}}[\![ping_{Z_{\{o0\}}}\,?\,(l : \mathsf{R})\,\mathsf{goto}\,l.\ \textsc{unrec}^{o00}_{\mathcal{P}\cup\{\{o0\}^0\}}(P)]\!] \\ |\ &home_{Z_{\{o0\}}}[\![ping_{Z_{\{o0\}}}\,!\,\langle k\rangle]\!]\end{aligned}$$

- 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,

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

  This will therefore heavily rely on implicit α-conversions.
- if $o0$ is in $\mathcal{P}$, then it must be in some O in $\mathcal{P}$;

$$\begin{aligned}\textsc{unrec}^o_{\mathcal{P}}(k[\![\mathsf{rec}\ Z : \mathsf{R}.\ P]\!]) = \\ home_{Z_O}[\![ping_{Z_O}\,?\,(l : \mathsf{R})\,\mathsf{goto}\,l.\ \textsc{unrec}^{o00}_{\mathcal{P}}(P)]\!] \\ |\ home_{Z_O}[\![ping_{Z_O}\,!\,\langle k\rangle]\!]\end{aligned}$$

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

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

- 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:

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

  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 $\textsc{unrec}_{\mathcal{P}}(M)$ for $\textsc{unrec}^{\varepsilon}_{\mathcal{P}}(M)$. 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 [8], namely bisimulation up-to-β. Thanks to this technique, we restrict the standard property of bisimulation to be proved for a relation $\mathcal{R}$ to $\mathcal{C}_1\ \mathcal{R}\ \mathcal{C}_2$ imply that $\mathcal{C}'_1 \xrightarrow{\tau}{}^*_\beta \sim \mathcal{R} \xrightarrow{\tau}{}^*_\beta \sim \mathcal{C}'_2$.

The notion of bisimulation up-to-β 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 12. 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 26 (unrec() is a 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 $\Omega \rhd \text{unrec}_{\mathcal{P}}(M) \xrightarrow{\mu} \xrightarrow{\tau}{}^*_\beta \sim \Omega' \rhd \text{unrec}_{\mathcal{P}'}(M')$ 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 $\Omega \rhd \text{unrec}^o_{\mathcal{P}}(M|_o) \xrightarrow{\mu} \xrightarrow{\tau}{}^*_\beta \sim \Omega \rhd \text{unrec}^{o'}_{\mathcal{P}'}(M'|_{o'})$.
  - (lts-go): $M|_o = k[\![\mathsf{goto}\, l.\ P]\!]$. This implies that $\text{unrec}^o_{\mathcal{P}}(M|_o)$ is the system $k[\![\mathsf{goto}\, l.\ \text{unrec}^{o00}_{\mathcal{P}}(P)]\!]$ optionally 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

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

    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 $\text{UNREC}^{o'}_{\mathcal{P}'}(M'|_{o'})$ is

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

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 $l[\![\text{UNREC}^{o00}_{\mathcal{P}}(P)]\!]$ 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 $\text{UNREC}_{\mathcal{P}}(M)$ that at the longest common system-prefix among occurrences in O is generated the server in the home-base:

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

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

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

which is exactly the system we need. And this new instance can be placed at o' by structural congruence, which is included in $\sim$.

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

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

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

$$l[\![(\mathsf{newloc}\, home_Z : \text{LOC}[ping_Z : \text{RW}\langle \mathsf{R} \rangle]) \\ (\text{UNREC}(Z) \mid \\ \mathsf{goto}\, home_Z.\ *ping_Z ? (l : \mathsf{R})\, \mathsf{goto}\, l.\ \text{UNREC}(P))]\!]$$

so by using (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-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-EQ) and (LTS-NEQ): those two rules are slightly different mostly because some occurrences have no residual by this reduction rules, which means that a home base might become redundant. For instance, consider the system M:

$$k[\![\mathsf{if}\ u = u\ \mathsf{then}\ \mathbf{0}\ \mathsf{else}\ \mathsf{rec}\ Z.\ .P]\!]$$

for which the translation $\textsc{unrec}^{\varepsilon}_{\{\{02\}\}}(M)$ contains a home-base $home_Z$ which is not in $\textsc{unrec}^{\varepsilon}_{\{\}}(M)$.

By taking the residual of $\mathcal{P}$ for $\mathcal{P}'$, the sets of occurrences of $\mathcal{P}$ that have an empty residual in $\mathcal{P}'$ correspond to such "vanished" recursive process. We can easily prove that a system of the form

$$\begin{aligned}&(\mathsf{new}\ home_{Z_{O_1}})\,(\mathsf{new}\ home_{Z_{O_2}})\ldots\\ &\quad home_{Z_{O_1}}[\![*ping_{Z_{O_1}}\ldots]\!]\,|\ldots|\,M\end{aligned}$$

in which $ping_{Z_{O_1}}$ is not free in M is strongly-bisimilar to

$$(\mathsf{new}\ home_{Z_{O_2}})\ldots|\ldots|\,M$$

By a reasoning on the different possible cases for the continuation of the condition similar to the (LTS-GO) case, we therefore obtain that the system reduces after some extra β-moves and strong-bisimulation, into the translation of M'.

- (LTS-REC): $M|_o = k[\![\mathsf{rec}\ Z : \mathsf{R}.\ P]\!]$ which reduces to the system $M'|_{o'} = k[\![P\{^{\mathsf{rec}\ (Z:\mathsf{R}).\ P}/_Z\}]\!]$. 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 $\textsc{unrec}^o_{\mathcal{P}\cup\{\{o0\}^{(0)}\}}(M|_o)$, 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

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

By the rule (LTS-COMM), this can reduce by a τ move into the system $home_{Z_O}[\![\mathsf{goto}\ k.\ \textsc{unrec}^{o00}_{\mathcal{P}}(P)]\!]$, 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

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

As in the case for rule (LTS-EQ), showing the adequation between this translation and $k[\![\textsc{unrec}^{o00}_{\mathcal{P}}(P)]\!]$ turns out to be a simple case analysis on the form of P, after considering whether there exists some occurrence of Z in P or the rest of the system or if this location $home_{Z_O}$ should be "garbage-collected" by strong bisimulation.

· (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

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

and

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

which implies

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

these equalities being true with the omission of the extra $home_{Z_O}$ that might be generated by $\textsc{unrec}^{o}_{\mathcal{P}}(M|_o)$ 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.

- $\Omega \rhd \textsc{unrec}^{o}_{\mathcal{P}}(M) \xrightarrow{\mu} \Omega' \rhd N'$. 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 accommodate 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 output 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 $\Omega \rhd \text{UNREC}_{\mathcal{P}'}(M')$ for some $\mathcal{P}'$.

· Otherwise, by definition of the translation, we know that the redex in $\text{UNREC}_{\mathcal{P}}(M)$ 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 $\Omega \rhd \text{UNREC}_{\mathcal{P}}(M) \xrightarrow{\mu} \xrightarrow{\tau}{}^*_\beta \equiv \Omega' \rhd \text{UNREC}_{\mathcal{P}'}(M')$, since the redex reduced in the μ-move is the same.

□

As announced, this proves that the implementation of recursion with replication generates observationnally equivalent processes, under the assumption that the extra migrations will never fail.

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, but still using recursive types. 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 computa-

tions. We conjecture that in such a setting there is no translation of recursive processes into iterative ones, which preserve their behaviour.

Acknowledgement We would like to thank the anonymous referees for their numerous comments and suggestions regarding the presentation of this work.

References

[1] D. Sangiorgi, D. Walker, The π-calculus, Cambridge University Press, 2001.

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

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

[4] V. Gapeyev, M. Levin, B. Pierce, Recursive subtyping revealed, Journal of Functional Programming 12 (6) (2003) 511–548, preliminary version in *International Conference on Functional Programming (ICFP)*, 2000. Also appears as Chapter 21 of *Types and Programming Languages* by Benjamin C. Pierce (MIT Press, 2002).

[5] B. C. Pierce, Types and Programming Languages, MIT Press, 2002.

[6] R. M. Amadio, L. Cardelli, Subtyping recursive types, ACM Transactions on Programming Languages and Systems 15 (4) (1993) 575–631.

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

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