# Global/Local Subtyping for a Distributed $\pi$-calculus

Peter Sewell
Peter.Sewell@cl.cam.ac.uk

August 22, 1997

**Abstract**

In the design of mobile agent programming languages there is a tension between the implementation cost and the expressiveness of the communication mechanisms provided. This paper gives a static type system for a distributed $\pi$-calculus in which the input and output capabilities of channels may be either global or local. This allows compile-time optimization where possible but retains the expressiveness of channel communication. Subtyping allows all communications to be invoked uniformly. Recursive types and products are included. The distributed $\pi$-calculus used integrates location and migration primitives from the Distributed Join Calculus with asynchronous $\pi$ communication, taking a simple reduction semantics. Some alternative calculi are discussed.

## Contents

**1 Introduction** **2**

**2 A distributed $\pi$-calculus** **5**

2.1 Syntax . . . 5

2.1.1 Types . . . 6

2.1.2 Processes . . . 6

2.2 Reduction semantics . . . 7

2.3 Examples . . . 8

2.4 Discussion . . . 10

2.4.1 Normal forms for implementation . . . 10

2.4.2 Process syntax with implicit location . 11

2.4.3 Coarse/fine grain communication and failure semantics . . . 12

2.4.4 Action calculus semantics . . . 13

**3 Global/local subtyping** **14**

3.1 Recursive types . . . 15

3.2 Kinds, Contexts, Types and Values . . . 15

3.3 Subtyping . . . 17

3.4 Colocality . . . 19

3.5 Processes . . . 21

**4 Soundness** **23**

4.1 Recursive types . . . 23

4.2 Kinds, Contexts, Types and Values . . . 24

4.3 Subtyping . . . 26

4.4 Colocality . . . 27

4.5 Processes . . . 27

4.6 Relocation . . . 28

4.7 Narrowing . . . 28

4.8 Substitution . . . 29

4.9 Subject reduction . . . 30

**5 Conclusion** **32**

**A Soundness proofs** **35**

A.1 Recursive types . . . 35

A.2 Kinds, Contexts, Types and Values . . . 37

A.3 Subtyping . . . 41

A.4 Colocality . . . 42

A.5 Processes . . . 43

A.6 Relocation . . . 44

A.7 Narrowing . . . 46

A.8 Substitution . . . 49

A.9 Subject reduction . . . 55

**References** **55**

# 1 Introduction

A common theme, underlying much programming language and system design, is that of restricting usage of some resource to be *local*. This can support clean design, provide robustness (against accidental errors and malicious attacks) and allow efficient implementation. The development of ubiquitous networking, including technologies such as HTTP and Java, has led to a great deal of interest in distributed programming, particularly in systems in which executing agents (or simply units of executable code) are sent across a network (a survey of some recent work can be found in the volume [VT97]). This brings new kinds of locality to the fore, with resources whose usage is restricted to, for example:

1. Computation within a single executing agent (which may migrate across a network). Examples of such resources include pointers to data held within the agent, or channels used for internal communication.
2. A machine address space. Example resources include access capabilities for low-level input/output primitives.
3. A group of trusted agents. Example resources include cryptographic keys that should be kept within the group.

In this paper we consider how such usage restrictions, particularly the first, can be enforced at compile-time by a static type system. We give a type system in which the input and output capabilities for a communication channel can be either *global*, and therefore usable within any agent, or restricted to be *local*, and therefore usable only within the agent where the channel is declared.

Our primary motivation for the type system is to allow efficient implementation of communication primitives, in programming languages that support migrating agents. The type system allows local communication to be implemented efficiently, while subtyping and subsumption ensure that, from the programmer's point of view, it is not unduly restrictive. The constructs for output and input along a channel are independent of whether its capabilities are global or local, thus facilitating programming (indeed, a narrowing result, Lemma 34.8, holds). At the same time the programmer can distinguish between local and (potentially expensive) global communications via the typing of channel declarations.

Very similar type systems should be applicable to the enforcement of secrecy properties for cryptographic keys or nonces. There is a strong analogy between the agents discussed in this paper and the individuals that engage in security protocols, and between the output (resp. input) of values on a channel and the encryption (resp. decryption) of values with a cryptographic key. Some possibilities are mentioned in the conclusion; for these the strong reasoning principles that can be licenced by such a type system will be central.

The design of programming languages that support migrating agents raises many other interesting issues, including failure modeling, name service semantics, dynamic binding and access control. Where possible, one would like to study these issues in isolation, and without becoming involved in the complexities of a real programming language. This can be done by working with simple *calculi* that have tractable semantic definitions. In this paper we introduce a *distributed $\pi$-calculus* that allows the global/local type system to be presented clearly; the other issues mentioned above appear to be orthogonal to this and are discussed only briefly, if at all. We build on the *$\pi$-calculus* of Milner, Parrow and Walker [MPW92]. This is a calculus of processes which communicate on named channels; its distinguishing feature is an operational semantics which allows the declaration of new channels and treats the sending of channel names themselves, along channels, in a compositional fashion. There may be many writers and readers on a channel; a message output by a writer is nondeterministically received by one of the readers.

The study of simple calculi must be interleaved with experimentation with actual programming languages. The development of the *Pict* language, of Pierce and Turner [PT97], is one such effort. Pict is a concurrent, though not distributed, programming language, closely based on the $\pi$-calculus. It has a rich type system and high-level syntax; the latter is translated (in both the implementation and the semantic definition) into a variant of the asynchronous $\pi$-calculus [Bou92, HT92]. The fact that this is possible shows that the asynchronous $\pi$-calculus is a sufficiently expressive basis for a programming language, although this does not automatically carry over to the distributed case.

The $\pi$-calculus is often described as a calculus of mobile processes. Strictly, however, this refers to the mobility of the scopes of channel declarations (channels are statically scoped, but their scopes may change over time as channel names are sent outside their current scopes). There is no other notion of locality or of identity of processes — in particular the separate identity of processes $P$ and $Q$ in a parallel composition $P \,|\, Q$ is not preserved. This means that to directly model distributed phenomena, such as migration of agents, failure of machines or knowledge of agents, one must add primitives for grouping $\pi$-calculus processes, into units of migration, failure or shared knowledge respectively. This was done by Amadio and Prasad [AP94] in order to model an abstraction of the failure semantics of *Facile* [TLK96], an extension of ML with distribution primitives. More recently, Fournet et al have proposed the *Distributed Join Calculus* and a closely related programming language [FG96, FGL$^+$96, FLMR97]. They argue that communication via $\pi$-calculus style channels is inappropriate for distributed programming and so adopt communication based on *join patterns* to which primitives for *locations*, which are units of migration and failure, are added. Locations are used to model both agents and physical machines.

There is a large design space of calculi with such primitives. For the purposes of this paper, however, many design choices are not critical — we require only a calculus with reasonably simple semantics that allows the type system to be presented clearly. In Section 2 we give such a distributed $\pi$-calculus, with the communication primitives of an asynchronous $\pi$-calculus and location and migration primitives based on those of the Distributed Join Calculus. Its reduction semantics can be given as a mild extension of that of the asynchronous $\pi$-calculus. Section 2 also contains some discussion of the design space, and touches on some of the other issues mentioned above. A full treatment is beyond the scope of this paper, however, so they are not reflected in the calculus.

A number of refined type systems for $\pi$-calculi have been studied, addressing polymorphism [FLMR97, LW95, PS97, Tur96, Vas94], directionality [Ode95, PS96], linearity and receptiveness [Ama97, KPT96, San97], deadlock-freedom [Kob97], object encodings [San96], confluence [Nie96, NS97], type inference [Gay93, VH93] and other phenomena (this is far from exhaustive). Each allows some particular behavioural discipline of processes to be expressed. It may be useful to contrast typing for $\pi$-calculi with the more standard typing for $\lambda$-calculi. A simply-typed $\lambda$-calculus might have types

$$T ::= \mathrm{Int} \;\Big|\; T \times T \;\Big|\; T \to T$$

with $T \to T'$ being the type of functions taking arguments of type $T$ and returning results of type $T'$. The type system will, under assumptions on the types of free variables, define the type(s) of any $\lambda$-term. In contrast, for $\pi$-calculi the fundamental type constructor is not that of functions but of channels carrying values of given types. One might have types

$$T ::= \mathrm{Int} \;\Big|\; T \times T \;\Big|\; \updownarrow T$$

where $\updownarrow T$ is the type of channels carrying values of type $T$. A type system will, under assumptions on the types of free names, define whether a $\pi$-term is a well-formed process or not (in most systems $\pi$-terms do not themselves have interesting types). For example, under the assumption that $w$ and $x$ are the names of channels carrying integers the one-shot buffer $w(y).\overline{x}y$, that reads an integer from $w$ and writes it to $x$, is a well-formed process. This would be written

$$w : \updownarrow \mathrm{Int}, x : \updownarrow \mathrm{Int} \vdash w(y).\overline{x}y : \mathrm{process}$$

Of particular relevance is the Input/Output subtyping of Pierce and Sangiorgi [PS96]. They refine channel types to treat the capabilities for reading and writing on channels separately. Specifically, they annotate channel types with *tags* $t$

$$T ::= \dots \;\Big|\; \updownarrow_t T$$

which are taken from

$$\begin{tikzcd}
- \arrow[dr, dash] & & + \arrow[dl, dash] \\
& \pm &
\end{tikzcd}$$

Channel names of type $\updownarrow_+ T$ (resp. $\updownarrow_- T$) can be used only for output (resp. input); names of type $\updownarrow_\pm T$ can be used for both output and input. This gives reasoning principles, notably showing the correctness of an encoding of the $\lambda$-calculus into the $\pi$-calculus. It also allows some very intuitive refined typing, hence preventing a class of programming errors. For example a library channel $\mathrm{printi}$, used for printing integers, can be typed as $\mathrm{printi} : \updownarrow_+ \mathrm{Int}$. Type checking will detect erroneous attempts to read from this channel.

In this paper we allow input and output capabilities to be not just absent or present but absent, local or global. Channel types will be of the form

$$T ::= \dots \;\Big|\; \updownarrow_{io} T$$

where the semi-tags $i$ and $o$ are taken from $\{-, \mathsf{L}, \mathsf{G}\}$. The intuition is that a $\mathsf{G}$ capability may be used at any location, an $\mathsf{L}$ capability may be used only at the location of the channel concerned and a $-$ capability may not be used at all. We will disallow the tag $--$ (see Section 3.3), leaving those below.

$$\begin{tikzcd}
& \mathsf{L}- \arrow[dl, dash] \arrow[dr, dash] & & -\mathsf{L} \arrow[dl, dash] \arrow[dr, dash] & \\
\mathsf{G}- \arrow[dr, dash] & & \mathsf{LL} \arrow[dl, dash] \arrow[dr, dash] & & -\mathsf{G} \arrow[dl, dash] \\
& \mathsf{GL} \arrow[dr, dash] & & \mathsf{LG} \arrow[dl, dash] & \\
& & \mathsf{GG} & &
\end{tikzcd}$$

Consider a channel $x$ of type $\updownarrow_{io} T$, which is located at a location named $k$ (this will be made more precise later), and the following cases:

$x : \updownarrow_{\mathsf{GG}} T$ Channel $x$ is usable at any location, both for input and for output.

$x : \updownarrow_{\mathsf{LL}} T$ Channel $x$ is usable only within location $k$, but still both for input and output.

$x : \updownarrow_{\mathsf{LG}} T$ Channel $x$ can be used for output anywhere, but for input only within location $k$. Such a channel might be used for sending requests to a server located at $k$.

$x : \updownarrow_{\mathsf{GL}} T$ Channel $x$ can be used for input anywhere, but for output only at location $k$. Such a channel might be used for receiving results from servers, or for 'pushed' data from an information source, particularly if it is necessary to change receivers.

The triples $\mathsf{GG}, \mathsf{G}-, -\mathsf{G}$ and $\mathsf{LL}, \mathsf{L}-, -\mathsf{L}$ correspond exactly to $\pm, -, +$ for global and local communication respectively.

In an implementation, channels with tags $\mathsf{LL}$, $\mathsf{L}-$ and $-\mathsf{L}$ can be implemented with data structures that are local to a single agent, and so always on the same (albeit possibly changing) machine. Their names need not be globally unique, but only unique within their location (note that this implies that equality testing of channel names should not be available) and need not be registered with global name services. Channels with tags GL (resp. LG) are subject to fewer optimizations, but still allow the references to the channel data structures by writers (resp. readers) to be local pointers. The expressiveness of the system should aid programmers by detecting errors at compile time, including communications that inadvertently potentially involve network communication.

The type system is introduced in Section 3. In addition to the channel types above it has base types, products, recursive types (for channels) and a top type. A subtyping order is lifted from the tag ordering above (with $i-$ and $-o$ contravariant and covariant respectively), allowing subcapabilities to be communicated. For example if $x : \updownarrow_{\mathsf{LG}} T$ then $x$ may be transmitted globally, along channels of type $\updownarrow_{\mathsf{GG}} \updownarrow_{-\mathsf{G}} T$, to readers that are guaranteed to use it only at type $\updownarrow_{-\mathsf{G}} T$. The typing rules involve two novel features — the formation of certain types must be forbidden by kinding rules and the capabilities of channels must be compared with capabilities at which they can be used by readers.

The soundness of the type system is proved in Section 4, with some details deferred to Appendix A. The main soundness result is subject reduction (Theorem 1); in addition one can see by examination of the typing rules that no well-typed process can immediately use a channel capability that it does not have (see Proposition 2). As stated above we are primarily interested in using the type system to allow efficient implementation. In principle one could state stronger soundness results in terms of optimized abstract machines. This is left for future work. One might also state stronger results in terms of an annotated labelled transition semantics. It is not clear, however, that for this system the gain would be worth the heavier definitions. For other applications, such as security, where one is primarily interested in the reasoning principles satisfied by well-typed processes, the situation would be different.

Some related work and possible generalisations are discussed in Section 5.

## 2 A distributed $\pi$-calculus

In this section the syntax and operational semantics of our distributed $\pi$-calculus (dpi for short) are given. The operational semantics is a rather mild extension of that for the asynchronous $\pi$-calculus. It is a reduction semantics, defining reductions over process terms (no additional notion of configuration is required) using a structural congruence. It differs from an asynchronous $\pi$ semantics in only two respects — there is a reduction rule for migration and the standard structural congruence and reduction rules are adapted to terms containing location information. Some examples are given in Section 2.3. Section 2.4 contains some discussion — of alternative treatments of location information in §2.4.1 and §2.4.2, of fine grain reductions in §2.4.3 and of an action calculus semantics in §2.4.4.

### 2.1 Syntax

The location and migration primitives of dpi are based on those of the Distributed Join Calculus [FGL+96]. Locations are tree-structured, with a root location $\mathrm{top}$. They may be used to model different entities — in this paper we consider immediate sublocations of $\mathrm{top}$ as modeling virtual machines, with descendants of these modeling executing software agents. Locations are named; new locations can be created and their names can be scope-extruded by communication just as $\pi$-calculus channel names can be. The semantics and type system can therefore be simplified by treating new location and channel declarations similarly, taking a single binder

$$(\mathbf{new}\ x : @_l T)\_$$

which declares $x$ to be a location (resp. channel) if the type $T$ is the type $\mathrm{loc}$ of location names (resp. a channel type). In the first case $x$ is a sublocation of $l$. Channels must also be located — a direct implementation of a $\pi$-calculus channel requires a queue of blocked readers or writers to be maintained. As noted in [AP94], this data structure may be on a different machine to the reader or writer so, to have an accurate failure or performance model, its location must be represented in the calculus. In the second case, therefore, channel $x$ is located at $l$.

### 2.1.1 Types

To the types of channels and locations introduced above we add pairs, as a first step towards more interesting datatypes, type variables, allowing discussion of type inference, recursive channel types, allowing channels to carry names of the same type as themselves, a type to be the top of the subtype order, base types and unit. We take a countably infinite set $\mathrm{TVar}$ of *type variables*, ranged over by $X, Y$, and a set of *base types*, for example $\mathrm{Int}$, ranged over by $B$. The *pre-types*, ranged over by $S, T, U, V$, are given by

$$
\begin{array}{rcll}
T & ::= & B & \text{base type} \\
  &     & 1 & \text{unit} \\
  &     & T \times T & \text{pairs} \\
  &     & \mathrm{loc} & \text{the type of location names} \\
  &     & \top & \text{top} \\
  &     & \updownarrow_{io} T & \text{channel carrying } T\text{, with capability } io \\
  &     & X & \text{type variable} \\
  &     & \mu X\ T & \text{recursive type}
\end{array}
$$

As discussed in the introduction, the tag $io$ ranges over

$$\{\, io \mid i \in \{\mathsf{G}, \mathsf{L}, -\} \wedge o \in \{\mathsf{G}, \mathsf{L}, -\} \wedge \neg(i = - \wedge o = -) \,\}$$

The type variable $X$ in $\mu X\ T$ binds in $T$; we work up to alpha conversion of bound type variables. The set of free type variables of a pre-type $T$ will be written $\mathrm{ftv}(T)$. Only some pre-types will be considered well-formed. The syntax of processes involves types, and hence the reduction semantics does also. Its definition does not depend on them in any interesting way, however, so we defer the type formation rules to Section 3.

### 2.1.2 Processes

We take a countably infinite set $\mathcal{X}$ of *names*, ranged over by $a, j, k, l, x, y, z$ and containing a distinguished name $\mathrm{top}$. We let $m, n, p, q$ range over $\mathbb{N}$. We suppose a set $|B|$ of elements of each base type $B$, and that the sets $|B|$, $\mathcal{X}$ and $\{\langle\rangle\}$ are disjoint from each other and from all products. *Values*, ranged over by $u, v, w$, are

$$
\begin{array}{rclll}
v & ::= & b & b \in |B| & \text{value of base type } B \\
  &     & \langle\rangle & & \text{element of unit type} \\
  &     & \langle v, v\rangle & & \text{pair} \\
  &     & x & & \text{name}
\end{array}
$$

There are two extremal possibilities for adding location information to terms. In one a locator applies to the largest possible unit, with all co-located subterms gathered into a single subterm. This is adopted, for example, in the *Ambient Calculus* of Cardelli and Gordon [CG97]. For dpi, however, communication is possible *across* the location tree structure, so to give a reduction semantics (in which writers and readers at different locations must be brought syntactically adjacent by a structural congruence) the other extreme is adopted, with each elementary subterm explicitly located (intermediate possibilities are discussed in §2.4). Accordingly, *processes*, ranged over by $P, Q, R$, are:

| | | | |
|---|---|---|---|
| $P$ | $::=$ | $@_u \overline{v} w$ | at location $u$, output value $w$ on channel $v$ |
| | | $@_u v(y).P$ | at $u$, input a value from channel $v$ and bind it to $y$ in $P$ |
| | | $@_u \, ! \, v(y).P$ | replicated input |
| | | $@_u \textbf{migrate\_to}\ v\ \textbf{then}\ P$ | migrate location $u$ to become a sublocation of $v$ |
| | | $@_u \textbf{let}\ \langle y:T, y':T' \rangle = w\ \textbf{in}\ P$ | at $u$, bind the halves of the pair $w$ to $y$ and $y'$ in $P$ |
| | | $(\textbf{new}\ y:@_u T)P$ | declare a new channel or location named $y$, of type $T$, located at $u$ and binding in $P$ |
| | | $0$ | the null process |
| | | $P \mid P$ | parallel composition |

The names $y$ and $y'$ above, which must be distinct in the **let** case, bind in the respective subterms $P$ (in particular, in $(\textbf{new}\ y:@_u T)P$ the scope of $y$ does not include $u$); we work up to alpha conversion of bound names. The free names of a value $v$ and process $P$ will be denoted by $\text{fn}(v)$ and $\text{fn}(P)$ respectively. The substitution of a value $v$ for a name $x$ in $P$ will be written $\{v/x\}P$. Output values and input binders of type 1 will often be elided.

The syntax of processes includes some nonsensical terms, which the reduction semantics gives nonsensical reductions to. They will be formally excluded by the typing rules but two points are worth mentioning now. Firstly, in well-typed processes the $u$ and $v$ appearing in the grammar will always be names. They are allowed to be arbitrary values in the syntax so that substitution of values for names is always defined. Secondly, the syntax includes terms which can teleport after a prefix, e.g.

$$@_k x(y).@_l x(z).0 \qquad @_k x(y).@_y x(z).0$$

For conceptual simplicity we would like migration to be the *only* way in which processes may move, and so want locators $@_l\_$ to describe the locations of processes rather than cause them to move. Teleporting terms are excluded by considering the set of free inhabited locations $\text{fl}(P)$ of a process $P$. This is the set of the free location names that are inhabited in $P$ by outputs, inputs, migrates, pair splits, channels or locations. It is the subset of $\text{fn}(P)$ defined below. The type system will require, in every prefix located at $l$ with continuation $P$, that $\text{fl}(P) \subseteq \{l\}$.

$$\begin{array}{rcl}
\text{fl}(@_u \overline{v} w) & \overset{def}{=} & \text{fn}(u) \\
\text{fl}(@_u v(y).P) & \overset{def}{=} & \text{fn}(u) \cup (\text{fl}(P) - \{y\}) \\
\text{fl}(@_u \, ! \, v(y).P) & \overset{def}{=} & \text{fn}(u) \cup (\text{fl}(P) - \{y\}) \\
\text{fl}(@_u \textbf{migrate\_to}\ v\ \textbf{then}\ P) & \overset{def}{=} & \text{fn}(u) \cup \text{fl}(P) \\
\text{fl}(@_u \textbf{let}\ \langle y:T, y':T' \rangle = w\ \textbf{in}\ P) & \overset{def}{=} & \text{fn}(u) \cup (\text{fl}(P) - \{y, y'\}) \\
\text{fl}((\textbf{new}\ y:@_u T)P) & \overset{def}{=} & \text{fn}(u) \cup (\text{fl}(P) - \{y\}) \\
\text{fl}(0) & \overset{def}{=} & \{\} \\
\text{fl}(P \mid Q) & \overset{def}{=} & \text{fl}(P) \cup \text{fl}(Q)
\end{array}$$

## 2.2 Reduction semantics

Structural congruence $\equiv$ is the least congruence relation over processes satisfying the following.

$$\begin{array}{rcll}
P \mid 0 & \equiv & P & \quad (1) \\
P \mid Q & \equiv & Q \mid P & \quad (2) \\
P \mid (Q \mid R) & \equiv & (P \mid Q) \mid R & \quad (3) \\
(\textbf{new}\ x:@_u S)(\textbf{new}\ y:@_v T)P & \equiv & (\textbf{new}\ y:@_v T)(\textbf{new}\ x:@_u S)P \qquad x \notin \text{fn}(v), y \wedge y \notin \text{fn}(u) & \quad (4) \\
P \mid (\textbf{new}\ x:@_v T)Q & \equiv & (\textbf{new}\ x:@_v T)(P \mid Q) \qquad x \notin \text{fn}(P) & \quad (5)
\end{array}$$

The first three equations are standard, allowing parallel compositions to be treated as multisets. Equation 5 allows scope extrusion, both of channel names and of location names. Equation 4 allows new-binders to be permuted; the side condition ensures that the location tree structure, and the locations of channels, are preserved.

The reduction relation $\longrightarrow$ over processes is the least relation satisfying the following.

$$\begin{array}{rcll}
@_k\overline{x}v \mid @_l x(y).P & \longrightarrow & \{v/y\}P & (1)\\
@_k\overline{x}v \mid @_l\,!\,x(y).P & \longrightarrow & \{v/y\}P \mid @_l\,!\,x(y).P & (2)\\
@_l\mathbf{let}\ \langle y_1 : T_1, y_2 : T_2\rangle = \langle v_1, v_2\rangle\ \mathbf{in}\ P & \longrightarrow & \{v_1/y_1\}\{v_2/y_2\}P & (3)\\
(\mathbf{new}\ l : @_j T)(\mathbf{new}\ \Delta)(Q \mid @_l\mathbf{migrate\_to}\ k\ \mathbf{then}\ P) & \longrightarrow & (\mathbf{new}\ l : @_k T)(\mathbf{new}\ \Delta)(Q \mid P) & (4)\\
\text{if } \{k, l\} \cap \mathrm{dom}(\Delta) = \{\} \wedge k \neq l & & &
\end{array}$$

$$\frac{P \longrightarrow Q}{P \mid R \longrightarrow Q \mid R} \quad (5)$$

$$\frac{P \longrightarrow Q}{(\mathbf{new}\ x : @_l T)P \longrightarrow (\mathbf{new}\ x : @_l T)Q} \quad (6)$$

$$\frac{P \equiv P' \quad P' \longrightarrow Q' \quad Q' \equiv Q}{P \longrightarrow Q} \quad (7)$$

where we define $(\mathbf{new}\ \Delta)P$, for lists of the grammar $\Delta ::= \bullet \;\big|\; \Delta, x : @_l T$, by

$$\begin{array}{rcl}
(\mathbf{new}\ \bullet)P & \stackrel{def}{=} & P\\
(\mathbf{new}\ \Delta, x : @_l T)P & \stackrel{def}{=} & (\mathbf{new}\ \Delta)(\mathbf{new}\ x : @_l T)P
\end{array}$$

The first two reduction rules are the standard communication rules for an asynchronous $\pi$-calculus; note that the communications can take place irrespective of the locations of the writer, reader and channel. The third is an unproblematic pair splitting reduction. The fourth is the only substantially new reduction rule. It allows location $l$ to migrate from being a sublocation of $j$ to become a sublocation of $k$. After the migration the continuation $P$ is released. The additional context $(\mathbf{new}\ \Delta)(Q \mid \_)$, which is preserved by the reduction, is required as the scope of $l$ may contain other location and channel declarations, and processes, that mention $l$. In particular, note that $Q$ may contain other subterms $@_l \ldots$ that remain located at $l$ as it migrates. Note also that the side condition means that the rule is not applicable if $k$ is a sublocation of $l$. Such migrations, which would introduce a cycle into the location tree, are blocked, although later migrations may unblock them. The last three rules are standard.

The sublocation tree of a migrating location is unchanged, and so migrates with it. The unit of migration is thus a subtree of locations with all their processes and channels. The largest unit that is guaranteed to stay together (and so always be on the same machine), however, is not a subtree but just the processes and channels at a single location — its sublocations may migrate away. The tree structure is therefore essentially orthogonal to global/local typing.

## 2.3 Examples

We give some simple example processes that will be well-typed in the empty context. Examples 1 and 4 correspond roughly to examples 2 and 4 of [FGL+96]. The syntax of processes contains redundant location information, for example in $@_l x(y).@_l x(y).P$ the second location must be $l$ and so the second $@_l\_$ could in principle be omitted. Some less verbose possibilities are discussed in Section 2.4.

1. A simple server process and client in different locations:

$$\begin{array}{l}
(\mathbf{new}\ \mathrm{printServer} : @_{\mathrm{top}}\mathrm{loc})(\mathbf{new}\ \mathrm{client} : @_{\mathrm{top}}\mathrm{loc})\\
\quad (\mathbf{new}\ \mathrm{print} : @_{\mathrm{printServer}} \updownarrow_{\mathsf{LG}} \mathrm{Int})\\
\quad\quad @_{\mathrm{printServer}}\, !\, \mathrm{print}(x). \cdots\\
\quad\quad |\\
\quad\quad @_{\mathrm{client}}\overline{\mathrm{print}}7
\end{array}$$

2. Two copies of a replicated server, together with a client:

$$\begin{array}{l}
(\mathbf{new}\ \mathrm{printServer1} : @_{\mathrm{top}}\mathrm{loc})(\mathbf{new}\ \mathrm{printServer2} : @_{\mathrm{top}}\mathrm{loc})(\mathbf{new}\ \mathrm{client} : @_{\mathrm{top}}\mathrm{loc})\\
\quad (\mathbf{new}\ \mathrm{print} : @_{\mathrm{top}} \updownarrow_{\mathsf{GG}} \mathrm{Int})\\
\quad\quad @_{\mathrm{printServer1}}\, !\, \mathrm{print}(x). \cdots\\
\quad\quad |\ @_{\mathrm{printServer2}}\, !\, \mathrm{print}(x). \cdots\\
\quad\quad |\ @_{\mathrm{client}}\overline{\mathrm{print}}7
\end{array}$$

The reduction semantics allows either server to receive the request, nondeterministically. A server may exist in multiple locations for reliability or performance reasons, or transiently during replacement of an old server. Whether an implementation would have satisfactory behaviour may depend on the fairness properties it guarantees. The location of channel $\mathrm{print}$ is arbitrary, although obviously it may affect performance.

3. A server that returns the result of some computation (in this trivial example it simply pairs the argument with itself):

$$\begin{array}{l}
(\mathbf{new}\ \mathrm{pairServer} : @_{\mathrm{top}}\mathrm{loc})(\mathbf{new}\ \mathrm{client} : @_{\mathrm{top}}\mathrm{loc})\\
\quad (\mathbf{new}\ \mathrm{pair} : @_{\mathrm{pairServer}} \updownarrow_{\mathsf{LG}}(\mathrm{Int} \times \updownarrow_{-\mathsf{G}}(\mathrm{Int} \times \mathrm{Int})))\\
\quad\quad @_{\mathrm{pairServer}}\, !\, \mathrm{pair}(y).@_{\mathrm{pairServer}}\mathbf{let}\ \langle n : \mathrm{Int}, c : \updownarrow_{-\mathsf{G}}(\mathrm{Int} \times \mathrm{Int})\rangle = y\ \mathbf{in}\ @_{\mathrm{pairServer}}\overline{c}\langle n, n\rangle\\
\quad\quad |\\
\quad\quad (\mathbf{new}\ c : @_{\mathrm{client}} \updownarrow_{\mathsf{LG}}(\mathrm{Int} \times \mathrm{Int}))@_{\mathrm{client}}\overline{\mathrm{pair}}\langle 7, c\rangle\ |\ @_{\mathrm{client}}c(x). \cdots
\end{array}$$

As in the asynchronous $\pi$-calculus the result must be returned on a new channel $c$. A realistic programming language would provide higher level syntax (as in Pict) and an optimized implementation of this. Note the use of subsumption for typing the output $@_{\mathrm{client}}\overline{\mathrm{pair}}\langle 7, c\rangle$.

4. A server that, on demand, migrates a copy of an applet to a client:

$$\begin{array}{l}
(\mathbf{new}\ \mathrm{appletServer} : @_{\mathrm{top}}\mathrm{loc})(\mathbf{new}\ \mathrm{client} : @_{\mathrm{top}}\mathrm{loc})\\
\quad (\mathbf{new}\ \mathrm{getApplet} : @_{\mathrm{appletServer}} \updownarrow_{\mathsf{LG}} \mathrm{loc})\\
\quad\quad @_{\mathrm{appletServer}}\, !\, \mathrm{getApplet}(l).(\mathbf{new}\ \mathrm{applet} : @_{\mathrm{appletServer}}\mathrm{loc})@_{\mathrm{applet}}\mathbf{migrate\_to}\ l\ \mathbf{then}\ \cdots\\
\quad\quad |\\
\quad\quad @_{\mathrm{client}}\overline{\mathrm{getApplet}}\ \mathrm{client}
\end{array}$$

(As it stands this does not set up channels for the applet and client to communicate.)

5. A rudimentary tracker, that receives location names on a channel $\mathrm{move}$ (perhaps provided by an active badge system controller [HH94]) and migrates to them:

$$\begin{array}{l}
(\mathbf{new}\ \mathrm{l}_1 : @_{\mathrm{top}}\mathrm{loc}) \cdots (\mathbf{new}\ \mathrm{l}_3 : @_{\mathrm{top}}\mathrm{loc})\\
(\mathbf{new}\ \mathrm{controller} : @_{\mathrm{top}}\mathrm{loc})\\
\quad (\mathbf{new}\ \mathrm{move} : @_{\mathrm{controller}} \updownarrow_{\mathsf{GL}} \mathrm{loc})\\
\\
\quad\quad @_{\mathrm{controller}}\overline{\mathrm{move}}\,\mathrm{l}_1\ |\cdots|\ @_{\mathrm{controller}}\overline{\mathrm{move}}\,\mathrm{l}_3\\
\quad\quad |\\
\quad\quad (\mathbf{new}\ \mathrm{follower} : @_{\mathrm{controller}}\mathrm{loc})\\
\quad\quad\quad @_{\mathrm{follower}}\, !\, \mathrm{move}(l).@_{\mathrm{follower}}\mathbf{migrate\_to}\ l\ \mathbf{then}\ \cdots\\
\quad\quad\quad |\ @_{\mathrm{follower}} \cdots
\end{array}$$

The channel $\mathrm{move}$ could be located at the follower, with type $\updownarrow_{\mathsf{LG}}\mathrm{loc}$ and presumably with altered performance. A more realistic tracker would have additional communications so that the moves could be sequentialised.

## 2.4 Discussion

### 2.4.1 Normal forms for implementation

The syntax of §2.1.2 admits a clean reduction semantics but is not directly suited for implementation. It contains a lot of redundant location information and allows co-located processes, that should be gathered together at compile time, to be syntactically separated. We give a less redundant syntax as a step towards an implementation representation. Two isomorphic grammars are below; on the left is a sub-grammar of processes, on the right is a grammar with redundant locators removed.

$$
\begin{array}{ll@{\qquad\qquad}ll}
P_l ::= & 0 & Q_l ::= & 0 \\
& P_l \mid P_l & & Q_l \mid Q_l \\
& @_l\overline{x}v & & \overline{x}v \\
& @_l x(y).P_l & & x(y).Q_l \\
& @_l\, !\, x(y).P_l & & !\, x(y).Q_l \\
& @_l \textbf{migrate\_to}\ k\ \textbf{then}\ P_l & & \textbf{migrate\_to}\ k\ \textbf{then}\ Q_l \\
& @_l \textbf{let}\ \langle y:S, y':S'\rangle = w\ \textbf{in}\ P_l & & \textbf{let}\ \langle y:S, y':S'\rangle = w\ \textbf{in}\ Q_l \\
& & & \\
& (\textbf{new}\ \Delta_l) & & (\textbf{new}\ \Theta_l) \\
& \quad (\textbf{new}\ l_2 : @_{k_2}L_2)(\textbf{new}\ \Delta_{l_2}) & & \quad (\textbf{new}\ l_2 : @_{k_2}L_2)(\textbf{new}\ \Theta_{l_2}) \\
& \quad \ldots & & \quad \ldots \\
& \quad (\textbf{new}\ l_n : @_{k_n}L_n)(\textbf{new}\ \Delta_{l_n}) & & \quad (\textbf{new}\ l_n : @_{k_n}L_n)(\textbf{new}\ \Theta_{l_n}) \\
& \quad\quad P_l \mid (P_{l_2} \mid (\ldots \mid P_{l_n})) & & \quad\quad Q_l \mid Q_{l_2} \mid \ldots \mid Q_{l_n}
\end{array}
$$

where $n \geq 1$, for $j \in 2..n$, $k_j$ is taken from $l, l_2, \ldots, l_{j-1}$ and $L_j \Downarrow \mathrm{loc}$, and

$$
\begin{array}{lcl@{\qquad\qquad}lcl}
\Delta_l & ::= & \bullet & \Theta_l & ::= & \bullet \\
& & \Delta_l, x : @_l T \ \text{ where } \ \neg(T \Downarrow \mathrm{loc}) & & & \Theta_l, x : T \ \text{ where } \ \neg(T \Downarrow \mathrm{loc})
\end{array}
$$

(We anticipate definitions of $\Downarrow$ and $\vdash$ from Section 3). Obviously the left form will not be closed under reduction; we conjecture, however, that there are good abstract machines which manipulate the terms of the right grammar. The abstract machine step for the last clause will involve the simultaneous creation of the subtree of locations $l_2, \ldots, l_n$, together with their channels and processes. Well-typed processes can be statically put into the left form, as follows.

**Proposition 1** *If $\Gamma \vdash P : \mathrm{process}$ and $\mathrm{fl}(P) = \{l_1, \ldots, l_p\}$ for $p \geq 1$ then there exists a term $\hat{P}$ of the form below such that $P \equiv \hat{P}$.*

$$
\begin{array}{l}
(\textbf{new}\ \Delta_{l_1}) \\
\quad \ldots \\
(\textbf{new}\ \Delta_{l_p}) \\
\quad (\textbf{new}\ l_{p+1} : @_{k_{p+1}}L_{p+1})(\textbf{new}\ \Delta_{l_{p+1}}) \\
\quad\quad \ldots \\
\quad (\textbf{new}\ l_{p+q} : @_{k_{p+q}}L_{p+q})(\textbf{new}\ \Delta_{l_{p+q}}) \\
\quad\quad P_{l_1} \mid (P_{l_2} \mid (\ldots \mid P_{l_{p+q}}))
\end{array}
$$

*where $q \geq 0$ and, for $j \in (p+1)..(p+q)$, $k_j$ is taken from $l_1, \ldots, l_{j-1}$ and $L_j \Downarrow \mathrm{loc}$.*

PROOF Induction on typing derivations. □

### 2.4.2 Process syntax with implicit location

The design of a syntax that is suitable for use by programmers is a separate question. The redundant location information in that of §2.1.2 is unacceptable, but the syntax of §2.4.1 is rather inflexible. Instead, one can allow locators to occur anywhere:

$$\begin{array}{rcl} P & ::= & 0 \\ & & P \mid P \\ & & \overline{x}w \\ & & x(y).P \\ & & !\,x(y).P \\ & & \textbf{migrate\_to}\ l\ \textbf{then}\ P \\ & & \textbf{let}\ \langle y:T, y':T'\rangle = w\ \textbf{in}\ P \\ & & (\textbf{new}\ y:T)P \\ & & (\textbf{new}\ y:@_l T)P \\ & & @_l P \end{array}$$

The reduction semantics and type system for this can be defined either directly or via a translation into the syntax of §2.1.2. A direct reduction semantics involves structural congruence and reduction rules such as the following

$$\begin{array}{rcll} @_l(P \mid Q) & \equiv & (@_l P) \mid (@_l Q) & \\ @_l(\textbf{new}\ x:T)P & \equiv & (\textbf{new}\ x:@_l T)(@_l P) & l \neq x \\ @_l(\textbf{new}\ x:@_k T)P & \equiv & (\textbf{new}\ x:@_k T)(@_l P) & l \neq x \\ @_k @_l P & \equiv & @_l P & \\ @_k 0 & \equiv & 0 & \end{array}$$

$$@_k \overline{x}v \mid @_l x(y).P \longrightarrow @_l\{v/y\}P$$

In the reduction rule the continuation $P$ may not be explicitly located, so the right hand side must repeat the locator $@_l\_$. If $P$ *is* explicitly located, however, this will introduce vacuous locators, requiring the fourth structural congruence equation to remove. Some care must be taken to ensure that the fourth and fifth equations preserve typing; we omit all the details.

A direct type system can be given either by using an additional judgement $\Gamma \vdash P : \mathrm{process}_l$, indexed by a location name $l$ at which the process context of $P$ will locate its unlocated subterms, or by explicitly accumulating constraints on the location of unlocated subterms.

A conceivable alternative would be to identify the constructs for declaring a location and for locating processes at it, essentially restricting $(\textbf{new}\ l:@_k\mathrm{loc})\_$ and $@_l\_$ to always occur together as

$$(\textbf{new}\ l:@_k\mathrm{loc})@_l\_$$

This is the approach taken in [FGL+96]. It is not compatible with a reduction semantics expressed purely with reductions between processes, as location names may be scope-extruded. For a programming language the simplicity might be desirable, but there is a loss of expressiveness — one cannot declare two locations that refer to one another, e.g. as in

$$@_j x(y).(\textbf{new}\ k:@_j\mathrm{loc})(\textbf{new}\ l:@_j\mathrm{loc})\ (@_k\overline{y}l \mid @_l\overline{y}k \mid @_j y(w).0)$$

Such cross-referencing would have to be established dynamically, by communications on channels declared outside $l$.

### 2.4.3 Coarse/fine grain communication and failure semantics

The reduction semantics is *coarse grain*, in that channel communications (rules 1 and 2) happen in single reduction steps. In a distributed implementation they may require several network communications. If writer, channel and reader are on machines $j$, $k$ and $l$ respectively then there might be three, sending a value from $j$ to $k$, a request from $l$ to $k$ and the value from $k$ to $l$. In order to give an accurate failure model this must be reflected in a finer-grain semantics, following [AP94]. Such fine grain reduction rules must refer to the locations of channels and so must involve additional context. Rule 1 could be replaced by the following:

$$\begin{array}{lcll}
(\mathbf{new}\, x : @_k T)(Q \mid @_j \overline{x}v) & \longrightarrow & (\mathbf{new}\, x : @_k T)(Q \mid @_k \overline{x}v) & j \neq k \\
(\mathbf{new}\, x : @_k T)(Q \mid @_l x(y).P) & \longrightarrow & (\mathbf{new}\, x : @_k T)(Q \mid @_k x(y).P) & l \neq k \\
(\mathbf{new}\, x : @_k T)(Q \mid @_k \overline{x}v \mid @_k x(y).P) & \longrightarrow & (\mathbf{new}\, x : @_k T)(Q \mid \{v/y\}P) &
\end{array}$$

The side conditions on the first two rules prevent the introduction of spurious divergencies. Contrasting with rule 1, communications can only occur on new-bound channels, not on free channels. This is perhaps closer to the desired behaviour of a distributed programming language, so is not problematic. The treatment of requests introduces a delicate relationship between the calculus and an implementation — in the second and third rule one would not wish to transmit the whole of $P$ across the network. Consider the example below.

$$\begin{array}{lcl}
(\mathbf{new}\, x : @_k T)(@_k \overline{x}z \mid @_l x(y).@_l \overline{y}) & \longrightarrow & (\mathbf{new}\, x : @_k T)(@_k \overline{x}z \mid @_k x(y).@_l \overline{y}) \\
& \longrightarrow & (\mathbf{new}\, x : @_k T)@_l \overline{z}
\end{array}$$

The subterm $@_k x(y).@_l \overline{y}$ of the intermediate state should correspond, in an implementation, to a request at location $k$, for a value from channel $x$, linked to a blocked reader at location $l$, containing the continuation $@_l \overline{y}$ abstracted on $y$. The body of the continuation thus remains at $l$ throughout. The global/local type system can be adapted to a fine-grain semantics at the cost of an additional typing rule for such requests.

Instead of adding term contexts to the reduction rules one can give reduction rules over terms equipped with location-contexts that record the locations of free names. We sketch such a system, making use of typing contexts $\Gamma$ that record both the types and the locations of names. These will be defined precisely in Section 3. The analogues of the three fine grain rules above are:

$$\begin{array}{lcll}
\Gamma, @_j \overline{x}v & \longrightarrow & \Gamma, @_k \overline{x}v & \Gamma \vdash x@k \wedge j \neq k \\
\Gamma, @_l x(y).P & \longrightarrow & \Gamma, @_k x(y).P & \Gamma \vdash x@k \wedge l \neq k \\
\Gamma, @_k \overline{x}v \mid @_k x(y).P & \longrightarrow & \Gamma, \{v/y\}P & \Gamma \vdash x@k
\end{array}$$

together with rules for structural congruence (both for processes and for legitimate permutations of typing contexts) and reduction under parallel composition and new-binders:

$$\frac{\Gamma'_1, P'_1 \equiv \Gamma_1, P_1 \quad \Gamma_1, P_1 \longrightarrow \Gamma_2, P_2 \quad \Gamma_2, P_2 \equiv \Gamma'_2, P'_2}{\Gamma'_1, P'_1 \longrightarrow \Gamma'_2, P'_2}$$

$$\frac{\Gamma, P \longrightarrow \Gamma', P'}{\Gamma, P \mid Q \longrightarrow \Gamma', P' \mid Q} \qquad \frac{(\Gamma, x : @_l T), P \longrightarrow (\Gamma', x : @_{l'} T), P'}{\Gamma, (\mathbf{new}\, x : @_l T)P \longrightarrow \Gamma', (\mathbf{new}\, x : @_{l'} T)P'}$$

The migration rule can also be cast into this style:

$$(\Gamma, k : @_a S, \Delta, l : @_j T, \Theta),\ @_l \mathbf{migrate\_to}\ k\ \mathbf{then}\ P \longrightarrow (\Gamma, k : @_a S, \Delta, l : @_k T, \Theta),\ P$$

This said, a distributed implementation of any programming language with migration and location-transparent communication will require a location directory service, for finding the physical machines of locations. In the large-scale case one must consider failures of the machines and communication links used for implementing this service, which will give a rather more intricate failure model at the language level.

### 2.4.4 Action calculus semantics

There is a rather large space of possible calculi with reduction semantics. One way of understanding it, particularly for comparing different calculi, is to put them into a common framework, such as the *Action Calculi* of Milner [Mil96]. This provides a well-understood structural congruence, with a clear graphical intuition, that has been helpful in the design of dpi. As an illustration, we give an action calculus mDPIC corresponding to the monadic part of dpi, and compare it with the monadic part of AC($\nu$, **out**, **box**, **rep**) from [Mil96, §5.4] (restricting the ouput arity of **box** to be zero).

We take the arity monoid $(\mathbb{N}, +, 0)$, the names of arity 1 to be $\mathcal{X}$ and controls:

| | | | | |
|---|---|---|---|---|
| mDPIC | $\mathbf{new}_T : 1 \to 1$ | $\mathbf{out} : 3 \to 0$ | $\dfrac{a : 1 \to 0}{\begin{array}{c}\mathbf{in}(a) : 2 \to 0 \\ \mathbf{rep}(a) : 2 \to 0\end{array}}$ | $\dfrac{a : 0 \to 0}{\mathbf{mig}(a) : 2 \to 0}$ |
| mAC($\nu$, **out**, **box**, **rep**) | $\boldsymbol{\nu} : 0 \to 1$ | $\mathbf{out} : 2 \to 0$ | $\dfrac{a : 1 \to 0}{\begin{array}{c}\mathbf{box}(a) : 1 \to 0 \\ \mathbf{rep}(a) : 1 \to 0\end{array}}$ | |

The mDPIC arities of **new**, **out**, **in** and **rep** are obtained by adding one to the source of their corresponding arities; the name binding the new port on a control gives the location of that control. The reaction rules are:

| | | | |
|---|---|---|---|
| mDPIC | $\langle kxz\rangle \cdot \mathbf{out} \otimes \langle lx\rangle \cdot \mathbf{in}(a)$ | $\longrightarrow$ | $\langle z\rangle \cdot a$ |
| | $\langle kxz\rangle \cdot \mathbf{out} \otimes \langle lx\rangle \cdot \mathbf{rep}(a)$ | $\longrightarrow$ | $\langle z\rangle \cdot a \otimes \langle lx\rangle \cdot \mathbf{rep}(a)$ |
| | $\langle j\rangle \cdot \mathbf{new}_T \cdot (l) C_\Delta[b \otimes \langle lk\rangle \cdot \mathbf{mig}(a)]$ | $\longrightarrow$ | $\langle k\rangle \cdot \mathbf{new}_T \cdot (l) C_\Delta[b \otimes a]$ |
| mAC($\nu$, **out**, **box**, **rep**) | $\langle xz\rangle \cdot \mathbf{out} \otimes \langle x\rangle \cdot \mathbf{box}(a)$ | $\longrightarrow$ | $\langle z\rangle \cdot a$ |
| | $\langle xz\rangle \cdot \mathbf{out} \otimes \langle x\rangle \cdot \mathbf{rep}(a)$ | $\longrightarrow$ | $\langle z\rangle \cdot a \otimes \langle x\rangle \cdot \mathbf{rep}(a)$ |

where in the third rule $\{k, l\} \cap \mathrm{dom}(\Delta) = \{\} \wedge k \neq l$ and

$$\begin{array}{rcl} C_\bullet & \stackrel{\mathit{def}}{=} & \_ \\ C_{\Delta, x : @_l T} & \stackrel{\mathit{def}}{=} & C_\Delta[\langle l\rangle \cdot \mathbf{new}_T \cdot (x)\_] \end{array}$$

The first two mDPIC reaction rules are obtained by adding (but ignoring) location information. In both action calculi reaction under controls is not admitted. Note that it is essential that mDPIC is not a reflexive action calculus, as otherwise the migration rule could introduce cycles into the location tree structure.

The function $[\![\_]\!]$ below maps processes in the monadic part of dpi to mDPIC actions of arity $0 \to 0$.

$$\begin{array}{rcl} [\![0]\!] & \stackrel{\mathit{def}}{=} & \mathbf{id}_0 \\ [\![P \mid Q]\!] & \stackrel{\mathit{def}}{=} & [\![P]\!] \otimes [\![Q]\!] \\ [\![@_l \overline{x} z]\!] & \stackrel{\mathit{def}}{=} & \langle lxz\rangle \cdot \mathbf{out} \\ [\![@_l x(y).P]\!] & \stackrel{\mathit{def}}{=} & \langle lx\rangle \cdot \mathbf{in}((y)[\![P]\!]) \\ [\![@_l \,!\, x(y).P]\!] & \stackrel{\mathit{def}}{=} & \langle lx\rangle \cdot \mathbf{rep}((y)[\![P]\!]) \\ [\![@_l \mathbf{migrate\_to}\ k\ \mathbf{then}\ P]\!] & \stackrel{\mathit{def}}{=} & \langle lk\rangle \cdot \mathbf{mig}([\![P]\!]) \\ [\![(\mathbf{new}\ x : @_l T) P]\!] & \stackrel{\mathit{def}}{=} & \langle l\rangle \cdot \mathbf{new}_T \cdot (x)[\![P]\!] \end{array}$$

We conjecture that this is a bijection, up to structural congruence, that preserves one-step reaction.

Such a result would not, of course, make the calculus of §2.1.2 obsolete. As with any framework, there is a cost (notational, if nothing else) in working with an embedding. Moreover some variant calculi, such as the second system of §2.4.3, do not easily fit into the framework.

## 3 Global/local subtyping

This section gives the global/local type system. The soundness results are given in Section 4, together with the intermediate properties they depend on. The type system defines a judgement $\Gamma \vdash P : \mathrm{process}$ which should be read as 'under assumptions $\Gamma$ the process $P$ is well-formed'. As usual these contexts $\Gamma$ contain assumptions on the types of names that may occur free in $P$. They must also contain assumptions on the locations of such names (and on the kinds of type variables). *Pre-contexts* are therefore lists:

$$\begin{array}{lll} \Gamma & ::= & \bullet & \text{the empty context} \\ & & \Gamma, x : @_l T & \Gamma \text{ extended with name } x\text{, located at } l\text{, of type } T \\ & & \Gamma, X : K & \Gamma \text{ extended with type variable } X \text{ of kind } K \end{array}$$

We now illustrate the three main phenomena that the type system must cope with. Firstly, a channel name must only be used (for input or output) if it has the appropriate capability, i.e. L or G for usages at its location; G for usages at other locations. For example, with respect to the context

$$\Gamma \overset{def}{=} k : @_{\mathrm{top}}\mathrm{loc},\ l : @_{\mathrm{top}}\mathrm{loc},\ w : @_l \updownarrow_{-\mathsf{G}} 1,\ z : @_l \updownarrow_{-\mathsf{L}} 1$$

we should have

$$\begin{array}{ll} \Gamma \vdash @_l\overline{w} : \mathrm{process} & \Gamma \vdash @_l\overline{z} : \mathrm{process} \\ \Gamma \vdash @_k\overline{w} : \mathrm{process} & \Gamma \not\vdash @_k\overline{z} : \mathrm{process} \end{array}$$

Secondly, local capabilities must not be sent outside their locations. Consider the context

$$\begin{array}{llll} \Gamma & \overset{def}{=} & k : @_{\mathrm{top}}\mathrm{loc}, & \text{top level location} \\ & & l : @_{\mathrm{top}}\mathrm{loc}, & \text{top level location} \\ & & z : @_l \updownarrow_{\mathsf{LL}} 1, & \text{local channel carrying 1, at } l \\ & & x : @_l \updownarrow_{\mathsf{GG}} \updownarrow_{\mathsf{LL}} 1 & \text{global channel carrying names of local channels carrying 1, at } l \end{array}$$

and the process $P \overset{def}{=} @_l\overline{x}z \mid @_k x(y).@_k\overline{y}$. At first sight one might expect $\Gamma \vdash P : \mathrm{process}$, but the reduction

$$@_l\overline{x}z \mid @_k x(y).@_k\overline{y} \longrightarrow @_k\overline{z}$$

can send both L capabilities of $z$ out of $l$ — it is clear that $\Gamma \vdash @_k\overline{z} : \mathrm{process}$ should not hold, and hence that $\Gamma \vdash P : \mathrm{process}$ should not. It is prevented by restricting type formation, ruling out channel types, such as $\updownarrow_{\mathsf{GG}} \updownarrow_{\mathsf{LL}} 1$, that can be used to communicate local capabilities globally. This is done in §3.2.

Thirdly, there must be a restriction on the mention of names outside their locations. This is a little delicate, as one cannot simply forbid all such mentions of the names of channels that are declared with some local capability. Consider the two contexts

$$\begin{array}{lll} \Gamma_1 & \overset{def}{=} & k : @_{\mathrm{top}}\mathrm{loc},\ l : @_{\mathrm{top}}\mathrm{loc},\ x : @_k \updownarrow_{\mathsf{LL}} \updownarrow_{\mathsf{LL}} 1,\ z : @_l \updownarrow_{\mathsf{LL}} 1 \\ \Gamma_2 & \overset{def}{=} & k : @_{\mathrm{top}}\mathrm{loc},\ l : @_{\mathrm{top}}\mathrm{loc},\ x : @_k \updownarrow_{\mathsf{LL}} \updownarrow_{-\mathsf{G}} 1,\ z : @_l \updownarrow_{\mathsf{LG}} 1 \end{array}$$

We should clearly have $\Gamma_1 \not\vdash @_k\overline{z} : \mathrm{process}$ and $\Gamma_2 \vdash @_k\overline{z} : \mathrm{process}$. Now consider the process $Q \overset{def}{=} @_k\overline{x}z \mid @_k x(y).@_k\overline{y}$, which has the reduction

$$@_k\overline{x}z \mid @_k x(y).@_k\overline{y} \longrightarrow @_k\overline{z}$$

With respect to $\Gamma_1$, channel $z$ is declared to be at $l$ and have a local output capability. This is used by the receiver on $x$ outside $l$, at $k$, so we should have $\Gamma_1 \not\vdash @_k\overline{x}z : \mathrm{process}$. On the other hand, with respect to $\Gamma_2$, channel $z$ is declared to have a local input capability and a global output capability. The type of $x$ ensures that the receiver can only use the output capability, so we should allow $\Gamma_2 \vdash @_k\overline{x}z : \mathrm{process}$. This would still apply if $x$ was of type $\updownarrow_{\mathsf{LL}} \updownarrow_{-\mathsf{L}} 1$ — the process $@_k\overline{x}z$ should be well typed iff the capabilities of $z$ and the capabilities at which it can be used by receivers (determined by the type of $x$) do not share a local capability (either for input or for output). This is made precise in §3.4.

The subtype order is discussed in §3.3; the typing rules for processes are given in §3.5.

## 3.1 Recursive types

Some routine preliminary definitions for dealing with recursive types are required. We wish to allow recursion only at channel types, disallowing pre-types such as $\mu X\ X$ and $\mu X\ X \times X$ but allowing $\mu X\ \updownarrow_{\mathsf{GG}} X$ and $\mu X\ (\updownarrow_{\mathsf{GG}} X) \times (\updownarrow_{\mathsf{GG}} X)$. This is done by requiring, in the type formation rule for $\mu X\ T$, that $X$ is *guarded in* $T$.

$$
\frac{}{X \text{ guarded in } B} \qquad
\frac{}{X \text{ guarded in } 1} \qquad
\frac{\begin{array}{c} X \text{ guarded in } T \\ X \text{ guarded in } T' \end{array}}{X \text{ guarded in } T \times T'} \qquad
\frac{}{X \text{ guarded in } \mathrm{loc}}
$$

$$
\frac{}{X \text{ guarded in } \top} \qquad
\frac{}{X \text{ guarded in } \updownarrow_{io} T} \qquad
\frac{X \neq Y}{X \text{ guarded in } Y} \qquad
\frac{X \text{ guarded in } T}{X \text{ guarded in } \mu Y\ T}
$$

This ensures that all well-formed types are *guarded*, defined as follows.

$$
\frac{}{B \text{ guarded}} \qquad
\frac{}{1 \text{ guarded}} \qquad
\frac{\begin{array}{c} T \text{ guarded} \\ T' \text{ guarded} \end{array}}{T \times T' \text{ guarded}} \qquad
\frac{}{\mathrm{loc} \text{ guarded}}
$$

$$
\frac{}{\top \text{ guarded}} \qquad
\frac{T \text{ guarded}}{\updownarrow_{io} T \text{ guarded}} \qquad
\frac{}{X \text{ guarded}} \qquad
\frac{T \text{ guarded} \quad X \text{ guarded in } T}{\mu X\ T \text{ guarded}}
$$

At various points recursive types must be *unfolded*. We define a relation $\_ \Downarrow \_$ over pre-types by

$$
B \Downarrow B \qquad 1 \Downarrow 1 \qquad T \times T' \Downarrow T \times T' \qquad \mathrm{loc} \Downarrow \mathrm{loc}
$$

$$
\top \Downarrow \top \qquad \updownarrow_{io} T \Downarrow \updownarrow_{io} T \qquad X \Downarrow X \qquad \frac{T \Downarrow S \quad S \neq X}{\mu X\ T \Downarrow \{\mu X\ T / X\} S}
$$

## 3.2 Kinds, Contexts, Types and Values

In this subsection we define four mutually recursive judgements:

| | |
|---|---|
| $\vdash \Gamma\ \mathit{ok}$ | context $\Gamma$ is well-formed |
| $\Gamma \vdash T : K$ | type $T$ has kind $K$ |
| $\Gamma \vdash v : T$ | value $v$ has type $T$ |
| $\Gamma \vdash x@l$ | name $x$ is located at $l$ |

The *kinds*, ranged over by $K$, are $\mathrm{Type}_{\gamma\varepsilon}$ where $\gamma$ and $\varepsilon$ range over the 2-point lattices $\mathsf{G} \leqslant -$ and

$\mathsf{E} \leqslant -$ respectively. They are ordered by the product order

$$\begin{tikzcd}
& \mathrm{Type}_{--} \arrow[dl, dash] \arrow[dr, dash] & \\
\mathrm{Type}_{\mathsf{G}-} \arrow[dr, dash] & & \mathrm{Type}_{-\mathsf{E}} \arrow[dl, dash] \\
& \mathrm{Type}_{\mathsf{GE}} &
\end{tikzcd}$$

The intuition is that types that have a kind $\mathrm{Type}_{\mathsf{G}\varepsilon}$ are *global*, with values of such types being freely communicable between locations. Types that have a kind $\mathrm{Type}_{\gamma\mathsf{E}}$ are *extensible*; new names at these types may be created by new-binders. We write $\vee$ and $\wedge$ for the least upper bounds and greatest lower bounds in these lattices. The formation rules for *contexts* are:

$$\frac{}{\vdash \bullet\ ok} \qquad \frac{\vdash \Gamma\ ok \quad X \notin \mathrm{dom}(\Gamma)}{\vdash \Gamma, X : K\ ok} \qquad \frac{\begin{array}{l}\Gamma \vdash T : K \\ \Gamma \vdash l : \mathrm{loc} \vee l = \mathrm{top} \\ x \notin \mathrm{dom}(\Gamma) \cup \{\mathrm{top}\}\end{array}}{\vdash \Gamma, x : @_l T\ ok}$$

Contexts thus contain location and type assumptions on free names, and kind assumptions on type variables. The rules ensure that locations are tree structured, with root $\mathrm{top}$. The kinding rules for *types* are:

$$\frac{\vdash \Gamma\ ok}{\Gamma \vdash B : \mathrm{Type}_{\mathsf{G}-}} \qquad \frac{\vdash \Gamma\ ok}{\Gamma \vdash 1 : \mathrm{Type}_{\mathsf{G}-}} \qquad \frac{\begin{array}{c}\Gamma \vdash T : \mathrm{Type}_{\gamma\varepsilon} \\ \Gamma \vdash T' : \mathrm{Type}_{\gamma'\varepsilon'}\end{array}}{\Gamma \vdash T \times T' : \mathrm{Type}_{(\gamma \vee \gamma')-}}$$

$$\frac{\vdash \Gamma\ ok}{\Gamma \vdash \mathrm{loc} : \mathrm{Type}_{\mathsf{GE}}} \qquad \frac{\vdash \Gamma\ ok}{\Gamma \vdash \top : \mathrm{Type}_{\mathsf{GE}}}$$

$$\frac{\begin{array}{c}\Gamma \vdash T : \mathrm{Type}_{\mathsf{G}\varepsilon} \\ io \in \{\mathsf{GG}, \mathsf{G}-, -\mathsf{G}\}\end{array}}{\Gamma \vdash \updownarrow_{io} T : \mathrm{Type}_{\mathsf{GE}}} \qquad \frac{\begin{array}{c}\Gamma \vdash T : \mathrm{Type}_{\mathsf{G}\varepsilon} \\ io \in \{\mathsf{GL}, \mathsf{LG}\}\end{array}}{\Gamma \vdash \updownarrow_{io} T : \mathrm{Type}_{-\mathsf{E}}} \qquad \frac{\begin{array}{c}\Gamma \vdash T : \mathrm{Type}_{\gamma\varepsilon} \\ io \in \{\mathsf{LL}, -\mathsf{L}, \mathsf{L}-\}\end{array}}{\Gamma \vdash \updownarrow_{io} T : \mathrm{Type}_{-\mathsf{E}}}$$

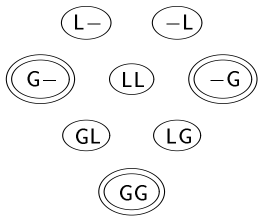
$$\frac{\vdash \Gamma, X : K, \Delta\ ok}{\Gamma, X : K, \Delta \vdash X : K} \qquad \frac{\begin{array}{c}\Gamma, X : K \vdash T : K \\ X \text{ guarded in } T\end{array}}{\Gamma \vdash \mu X\, T : K} \qquad \frac{\begin{array}{c}\Gamma \vdash T : K \\ K \leqslant K'\end{array}}{\Gamma \vdash T : K'}$$

The rules for channel types prevent the formation of types that could be used to carry local capabilities between locations. For example, we have:

$$\bullet \vdash \updownarrow_{\mathsf{LL}} \updownarrow_{\mathsf{LL}} 1 : \mathrm{Type}_{-\mathsf{E}} \qquad \bullet \not\vdash \updownarrow_{\mathsf{GG}} \updownarrow_{\mathsf{LL}} 1 : \mathrm{Type}_{--}$$

$$\bullet \vdash \updownarrow_{\mathsf{LL}} \updownarrow_{\mathsf{GG}} 1 : \mathrm{Type}_{-\mathsf{E}} \qquad \bullet \vdash \updownarrow_{\mathsf{GG}} \updownarrow_{\mathsf{GG}} 1 : \mathrm{Type}_{\mathsf{GE}}$$

and $\bullet \vdash \updownarrow_{io} \updownarrow_{i'o'} 1 : \mathrm{Type}_{-\mathsf{E}}$ iff $io \in \{\mathsf{GG}, \mathsf{G}-, -\mathsf{G}, \mathsf{GL}, \mathsf{LG}\} \Rightarrow i'o' \in \{\mathsf{GG}, \mathsf{G}-, -\mathsf{G}\}$, i.e. if $io$ is at all global then $i'o'$ must be not at all local. Products are global only if both their components are global. Base types, unit and $\top$ are global, as is $\mathrm{loc}$, so location names may be communicated freely. For illustration, the types (in boxes) and global types (in double boxes) of the form $\updownarrow_{io} 1$ are shown below, and those of the form $\updownarrow_{io} \updownarrow_{i'o'} 1$ are shown in Figure 1.



The only extensible types are channel types, $\mathrm{loc}$, $\top$, and type variables of kinds $\mathrm{Type}_{\gamma\mathsf{E}}$. The typing rules for *values* are straightforward. Recursive types are unfolded, if necessary, in the rule for names; the typing rules for processes then do not need to mention unfolding.

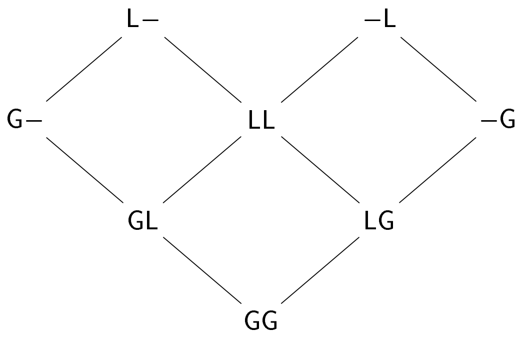
$$\frac{\vdash \Gamma, x : @_l S, \Delta \ ok \quad S \Downarrow T}{\Gamma, x : @_l S, \Delta \vdash x : T} \qquad \frac{\vdash \Gamma \ ok \quad b \in |B|}{\Gamma \vdash b : B} \qquad \frac{\vdash \Gamma \ ok}{\Gamma \vdash \langle\rangle : 1} \qquad \frac{\Gamma \vdash v : T \quad \Gamma \vdash v' : T'}{\Gamma \vdash \langle v, v' \rangle : T \times T'}$$

Finally, there is a single rule for the location of names:

$$\frac{\vdash \Gamma, x : @_l T, \Delta \ ok}{\Gamma, x : @_l T, \Delta \vdash x@l}$$

## 3.3 Subtyping

The ordering on tags



induces a subtype order on types — if $io \leqslant i'o'$ then a channel of type $\updownarrow_{io} T$ may be used as if it were a channel of type $\updownarrow_{i'o'} T$, which has weaker capabilities. As in [PS96], the variance of tags is

$$\begin{aligned} \mathrm{covariant}(io) &\overset{def}{\Leftrightarrow} o = - \\ \mathrm{contravariant}(io) &\overset{def}{\Leftrightarrow} i = - \\ \mathrm{nonvariant}(io) &\overset{def}{\Leftrightarrow} i \neq - \wedge o \neq - \end{aligned}$$

Clearly if $io \leqslant i'o'$ then

$$\begin{aligned} \mathrm{covariant}(io) &\Rightarrow \mathrm{covariant}(i'o') \\ \mathrm{contravariant}(io) &\Rightarrow \mathrm{contravariant}(i'o') \\ \mathrm{covariant}(i'o') &\Rightarrow \mathrm{covariant}(io) \vee \mathrm{nonvariant}(io) \\ \mathrm{contravariant}(i'o') &\Rightarrow \mathrm{contravariant}(io) \vee \mathrm{nonvariant}(io) \\ \mathrm{nonvariant}(i'o') &\Rightarrow \mathrm{nonvariant}(io) \end{aligned}$$

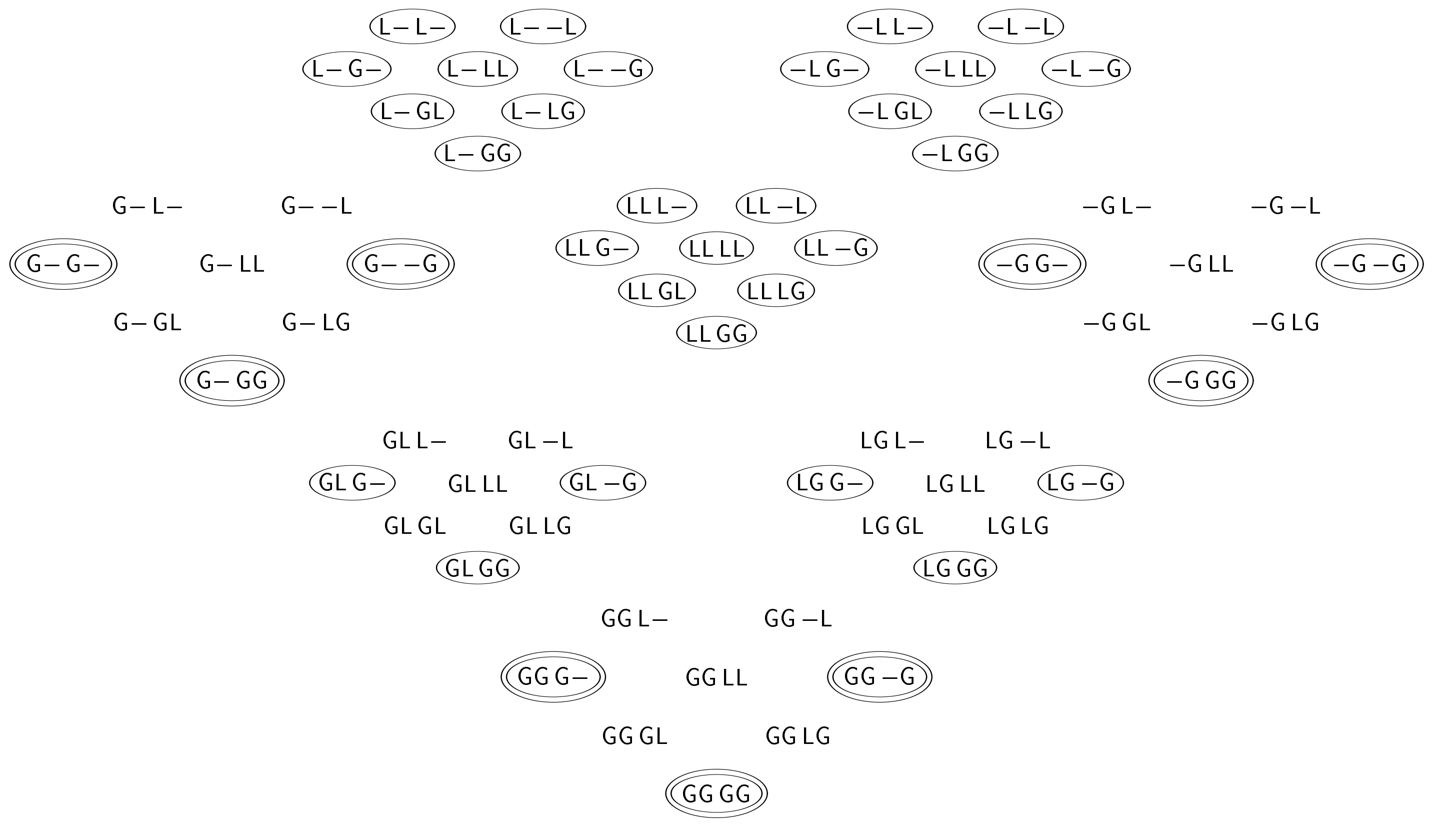


Figure 1: The well-formed types. The vertex $io\ i'o'$ is boxed if $\bullet \vdash \updownarrow_{io} \updownarrow_{i'o'} 1 : \mathrm{Type}_{-\mathsf{E}}$, and is in a double box if $\bullet \vdash \updownarrow_{io} \updownarrow_{i'o'} 1 : \mathrm{Type}_{\mathsf{GE}}$.

The subtype order is defined coinductively (without recursive types the two unfolding rules below could be omitted and a least fixed point could be used). If $\leqslant$ is a binary relation over the guarded pre-types then $\dot{\leqslant}$ is the least binary relation over the guarded pre-types such that

$$\frac{}{\mathrm{loc}\mathrel{\dot{\leqslant}}\mathrm{loc}} \qquad \frac{}{S\mathrel{\dot{\leqslant}}\top} \qquad \frac{\begin{array}{l} io \leqslant i'o' \\ \mathrm{covariant}(i'o') \Rightarrow S \leqslant T \\ \mathrm{contravariant}(i'o') \Rightarrow T \leqslant S \\ \mathrm{nonvariant}(i'o') \Rightarrow S \leqslant T \leqslant S \end{array}}{\updownarrow_{io} S \mathrel{\dot{\leqslant}} \updownarrow_{i'o'} T}$$

$$\frac{}{B\mathrel{\dot{\leqslant}}B} \qquad \frac{}{1\mathrel{\dot{\leqslant}}1} \qquad \frac{S_1 \leqslant T_1 \quad S_2 \leqslant T_2}{S_1 \times S_2 \mathrel{\dot{\leqslant}} T_1 \times T_2}$$

$$\frac{}{X\mathrel{\dot{\leqslant}}X} \qquad \frac{\mu X\ S \Downarrow S' \quad S' \mathrel{\dot{\leqslant}} T}{\mu X\ S \mathrel{\dot{\leqslant}} T} \qquad \frac{S \mathrel{\dot{\leqslant}} T' \quad \mu X\ T \Downarrow T'}{S \mathrel{\dot{\leqslant}} \mu X\ T}$$

The subtype order $\leqslant$ is taken to be the greatest fixed point of this operator. It is a preorder; in addition one can show that the quotient of $\leqslant$ by a recursion-unfolding equivalence is a partial order (see Proposition 22). If the tag $--$ was included this would no longer hold. Names of types $\updownarrow_{--} T$ would be communicable but not usable, so we would have $\updownarrow_{--} S \leqslant \updownarrow_{--} T$ for all $S$ and $T$. It seems cleaner to take a single top type $\top$. The subtype order over well-formed types $\updownarrow_{io} \updownarrow_{i'o'} 1$ is illustrated in Figure 2. Note that the well-formed types are not up, down or convex-closed under the subtype order on pre-types.

## 3.4 Colocality

A tag is *local* if it contains an L capability; two tags are *colocal* if they share a common L capability:

$$\begin{array}{rcl} \mathrm{local}(io) & \overset{\mathit{def}}{\Leftrightarrow} & i = \mathsf{L} \vee o = \mathsf{L} \\ \mathrm{colocal}(io, i'o') & \overset{\mathit{def}}{\Leftrightarrow} & (i = \mathsf{L} \wedge i' = \mathsf{L}) \vee (o = \mathsf{L} \wedge o' = \mathsf{L}) \end{array}$$

The key properties of these definitions are that $\mathrm{colocal}(io, io) \iff \mathrm{local}(io)$ and that, if $io \leqslant i'o' \leqslant i''o''$ and $\mathrm{colocal}(io, i''o'')$, then $\mathrm{colocal}(io, i'o')$ and $\mathrm{colocal}(i'o', i''o'')$. Note that the local tags are neither up, down or convex closed in the tag ordering. Further, colocal is a symmetric relation but is not reflexive or transitive, or closed under relational composition with the tag ordering. It does satisfy $\mathrm{colocal}(io, i'o') \Rightarrow (io \leqslant i'o' \vee i'o' \leqslant io)$. Colocality is lifted from tags to a relation on contexts and pairs of types, that are well formed in the context and in the subtype relation, as follows. colocal is the least subset of $\{\, \Gamma, S, T \mid \Gamma \vdash S : \mathrm{Type}_{--} \wedge \Gamma \vdash T : \mathrm{Type}_{--} \wedge S \leqslant T \,\}$ satisfying the following

$$\frac{\mathrm{colocal}(io, i'o')}{\mathrm{colocal}(\Gamma, \updownarrow_{io} S, \updownarrow_{i'o'} T)} \qquad \frac{\mathrm{colocal}(\Gamma, S_1, T_1) \vee \mathrm{colocal}(\Gamma, S_2, T_2)}{\mathrm{colocal}(\Gamma, S_1 \times S_2, T_1 \times T_2)}$$

$$\frac{\neg(\Gamma \vdash X : \mathrm{Type}_{\mathsf{G}-})}{\mathrm{colocal}(\Gamma, X, X)} \qquad \frac{\mathrm{colocal}(\Gamma, S', T') \quad S \Downarrow S' \quad T \Downarrow T'}{\mathrm{colocal}(\Gamma, S, T)}$$

The key properties lift as follows, anticipating the statements of lemmas from Section 4.4.

A type that is colocal with itself is local.

**Lemma 23** *If* $\mathrm{colocal}(\Gamma, T, T)$ *then* $\neg(\Gamma \vdash T : \mathrm{Type}_{\mathsf{G}-})$.

Colocality is convex-closed with respect to subtyping (for well-formed types).

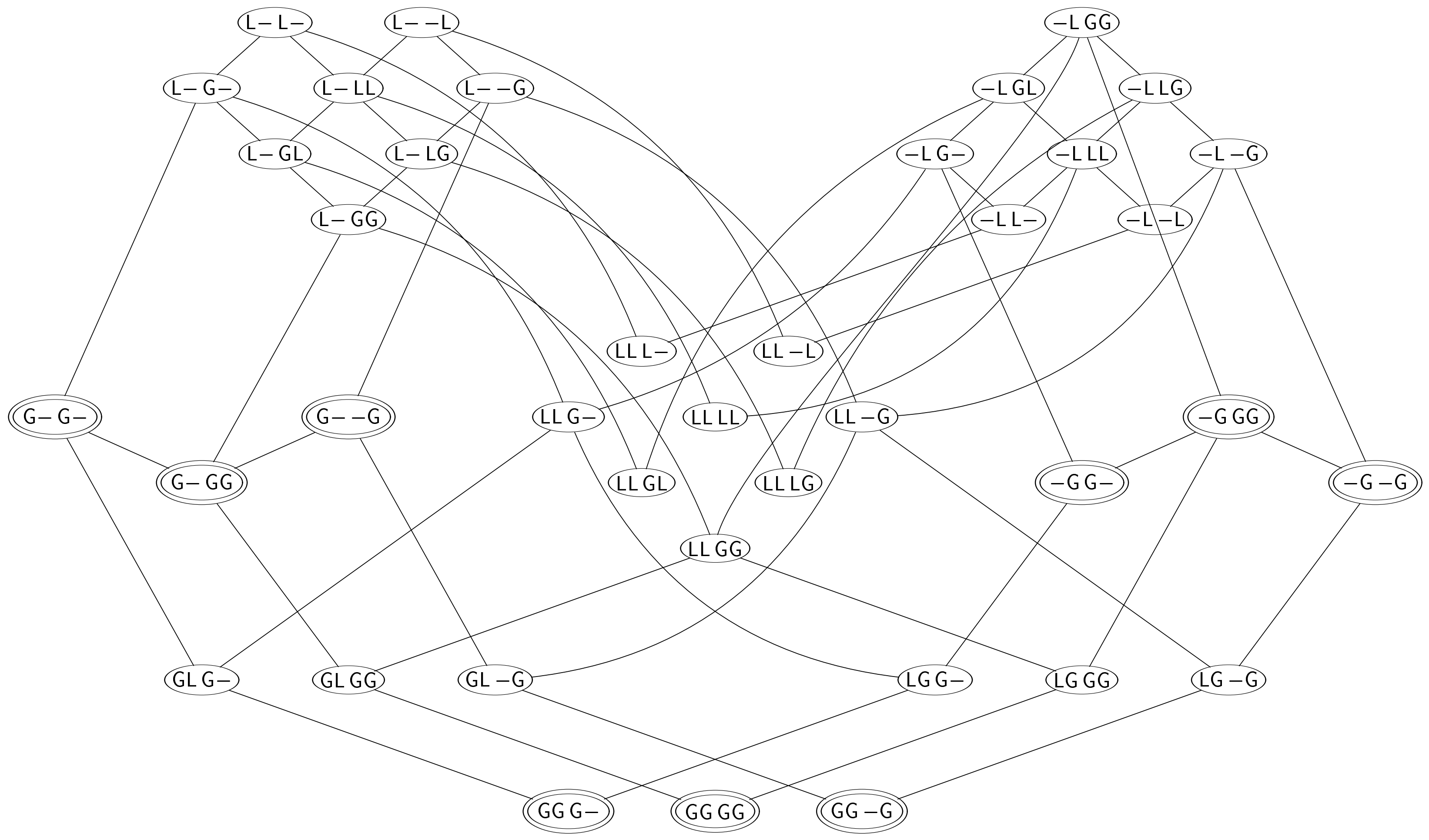


Figure 2: The subtype order over well-formed types $\updownarrow_{io}\updownarrow_{i''o''}1$. They are ordered by $\updownarrow_{io}\updownarrow_{i''o''}1 \leqslant \updownarrow_{i'o'}\updownarrow_{i'''o'''}1$ iff $io \leqslant i'o'$ and $o' = - \Rightarrow i'o' \leqslant i'''o'''$, $i' = - \Rightarrow i'''o''' \leqslant i'o'$ and $o' \neq - \wedge i' \neq - \Rightarrow i'o' = i'''o'''$.

**Lemma 24** *If* $T_1 \leqslant T_2 \leqslant T_3$, $\mathrm{colocal}(\Gamma, T_1, T_3)$ *and* $\Gamma \vdash T_2 : \mathrm{Type}_{\_\_}$ *then* $\mathrm{colocal}(\Gamma, T_1, T_2)$ *and* $\mathrm{colocal}(\Gamma, T_2, T_3)$.

We define the colocal names of a value with respect to two types that are in the subtype relation: if $\Gamma \vdash v : S$, $\Gamma \vdash T : \mathrm{Type}_{\_\_}$ and $S \leqslant T$ then $\mathrm{colocaln}(\Gamma, v, S, T)$ is the least subset of $\mathrm{fn}(v)$ satisfying

$$\frac{\mathrm{colocal}(\Gamma, S, T)}{x \in \mathrm{colocaln}(\Gamma, x, S, T)}$$

$$\frac{}{\mathrm{colocaln}(\Gamma, v_1, S_1, T_1) \cup \mathrm{colocaln}(\Gamma, v_2, S_1, T_1) \subseteq \mathrm{colocaln}(\Gamma, \langle v_1, v_2 \rangle, S_1 \times S_2, T_1 \times T_2)}$$

$$\frac{T \Downarrow T'}{\mathrm{colocaln}(\Gamma, v, S, T') \subseteq \mathrm{colocaln}(\Gamma, v, S, T)}$$

If a value has any colocal names with respect to two types then those types are colocal.

**Lemma 25** *If* $\mathrm{colocaln}(\Gamma, v, S, T) \neq \{\}$ *then* $\mathrm{colocal}(\Gamma, S, T)$.

The types of the colocal names of a value are themselves colocal.

**Lemma 27** *If* $x \in \mathrm{colocaln}(\Gamma, v, V, T)$ *then there exists* $S$ *such that* $\Gamma \vdash x : S$ *and* $\mathrm{colocal}(\Gamma, S, S)$.

## 3.5 Processes

Finally the typing rules for processes can be given.

$$\textsc{Out}\ \frac{\begin{array}{l}\Gamma \vdash l : \mathrm{loc} \\ \Gamma \vdash x : \updownarrow_{io} T \\ \Gamma \vdash v : T' \\ T' \leqslant T \\ o \leqslant \mathsf{L} \\ o = \mathsf{L} \Rightarrow \Gamma \vdash x@l \\ \forall a \in \mathrm{colocaln}(\Gamma, v, T', T)\ .\ \Gamma \vdash a@l\end{array}}{\Gamma \vdash @_l \overline{x} v : \mathrm{process}}
\qquad
\textsc{(Rep-)In}\ \frac{\begin{array}{l}\Gamma \vdash l : \mathrm{loc} \\ \Gamma \vdash x : \updownarrow_{io} T \\ \Gamma, y : @_l T \vdash P : \mathrm{process} \\ i \leqslant \mathsf{L} \\ i = \mathsf{L} \Rightarrow \Gamma \vdash x@l \\ \mathrm{fl}(P) \subseteq \{l\}\end{array}}{\begin{array}{l}\Gamma \vdash @_l x(y).P : \mathrm{process} \\ \Gamma \vdash @_l\, !\, x(y).P : \mathrm{process}\end{array}}$$

$$\textsc{Mig}\ \frac{\begin{array}{c}\Gamma \vdash l : \mathrm{loc} \\ \Gamma \vdash v : \mathrm{loc} \\ \Gamma \vdash P : \mathrm{process} \\ \mathrm{fl}(P) \subseteq \{l\}\end{array}}{\Gamma \vdash @_l \textbf{migrate\_to}\ v\ \textbf{then}\ P : \mathrm{process}}
\qquad
\textsc{Let}\ \frac{\begin{array}{l}\Gamma \vdash l : \mathrm{loc} \\ \Gamma \vdash v : T' \\ \Gamma, y_1 : @_l T_1, y_2 : @_l T_2 \vdash P : \mathrm{process} \\ T' \leqslant T_1 \times T_2 \\ \forall a \in \mathrm{colocaln}(\Gamma, v, T', T_1 \times T_2)\ .\ \Gamma \vdash a@l \\ \mathrm{fl}(P) \subseteq \{l\}\end{array}}{\Gamma \vdash @_l \textbf{let}\ \langle y_1 : T_1, y_2 : T_2 \rangle = v\ \textbf{in}\ P : \mathrm{process}}$$

$$\textsc{Nil}\ \frac{\vdash \Gamma\ \mathit{ok}}{\Gamma \vdash 0 : \mathrm{process}}
\qquad
\textsc{Par}\ \frac{\begin{array}{c}\Gamma \vdash P : \mathrm{process} \\ \Gamma \vdash Q : \mathrm{process}\end{array}}{\Gamma \vdash P \mid Q : \mathrm{process}}$$

$$\textsc{New}\ \frac{\begin{array}{c}\Gamma \vdash T : \mathrm{Type}_{\_\mathsf{E}} \\ \Gamma, x : @_l T \vdash P : \mathrm{process}\end{array}}{\Gamma \vdash (\textbf{new}\ x : @_l T) P : \mathrm{process}}$$

Most of the premises of these rules are routine; we discuss the others briefly.

**Out** • The first premise ensures that $l$ is a location.

- The second through fifth premises are analogous to those of the OUT rule of [PS96]. Name $x$ must be a channel, value $v$ must be of a subtype of the type carried by the channel, and the channel must have an output capability (either G or L). The fourth and fifth premises could be replaced by $\updownarrow_{io} T \leqslant \updownarrow_{-\mathsf{L}} T'$.
- The penultimate premise addresses the first phenomenon discussed at the beginning of this section, ensuring that if $x$ has only a local output capability then it can only be used at its own location.
- The last premise addresses the third such phenomenon, ensuring that any transmitted channel names that have a local capability which can be used by receivers on $x$ are located here.

**(REP-)IN** This is very similar to OUT except for the premise $\text{fl}(P) \subseteq \{l\}$, which prevents teleportation after the input. Note that for typing $P$ it is assumed that $y$ is located at $l$.

**NEW** This allows new-binding of names at any extensible type, particularly at channel types and $\text{loc}$, but also at $\top$ and suitable type variables.

**LET** This is similar to a combination of OUT and (REP-)IN (as, indeed, the reduction rule for LET is).

**MIG, PAR, NIL** These are all straightforward.

The rules above allow locations and channels, but not processes, to be located at $\text{top}$. This is consistent with the intuition that immediate sublocations of $\text{top}$ model physical machines. For other applications of the calculus different treatments of $\text{top}$ are appropriate and should be straightforward. In the fine grain case if channels can be at $\text{top}$ then messages and requests must be allowed also.

A few examples:

1. Local channels can be sent outside their location (with reduced capabilities) and then back inside. Their local capabilities cannot then be used, however. For example

$$\begin{array}{rcl} \Gamma & \overset{\textit{def}}{=} & k : @_{\text{top}}\text{loc},\ l : @_{\text{top}}\text{loc},\ x : @_{\text{top}} \updownarrow_{\mathsf{GG}} \updownarrow_{-\mathsf{G}} 1,\ z : @_l \updownarrow_{\mathsf{LG}} 1 \\ P & \overset{\textit{def}}{=} & @_l \overline{x} z \mid @_k x(y).@_k \overline{x} y \mid @_l x(y).Q \end{array}$$

   We have $\Gamma \vdash P : \text{process}$ and $P \longrightarrow\longrightarrow \{z/y\}Q$, but $Q$ must be typed with respect to $\Gamma, y : @_l \updownarrow_{-\mathsf{G}} 1$ so $\{z/y\}Q$ cannot use the input capability of $z$.

2. A name may be assumed to have a local type in a process $P$ and still, if $P$ is placed in a process context, engage in cross-location communication. For example if $P \overset{\textit{def}}{=} @_k y()$:

$$\begin{array}{rcl} \Gamma & \overset{\textit{def}}{=} & k : @_{\text{top}}\text{loc},\ l : @_{\text{top}}\text{loc},\ x : @_k \updownarrow_{\mathsf{LL}} \updownarrow_{\mathsf{LL}} 1,\ z : @_l \updownarrow_{\mathsf{GG}} 1 \\ \Gamma, y : @_k \updownarrow_{\mathsf{LL}} 1 & \vdash & P : \text{process} \\ \Gamma & \vdash & @_l \overline{z} \mid @_k \overline{x} z \mid @_k x(y).P : \text{process} \end{array}$$

3. The let construct includes an explicit type for its pattern, which may be a supertype of the type of its value. Without this the set of typable processes would be unduly restricted. With an untyped let, if $\Gamma \vdash v : T_1 \times T_2$ we would have

$$\Gamma \vdash @_l \textbf{let } \langle y_1, y_2 \rangle = v \textbf{ in } P : \text{process} \iff \Gamma \vdash @_l \textbf{let } \langle y_1 : T_1, y_2 : T_2 \rangle = v \textbf{ in } P : \text{process}$$

   Consider

$$\Gamma \overset{\textit{def}}{=} k : @_{\text{top}}\text{loc},\ l : @_{\text{top}}\text{loc},\ z : @_k \updownarrow_{\mathsf{LG}} 1$$

   and $P$ such that

$$\Gamma, y_1 : @_l 1, y_2 : @_l \updownarrow_{-\mathsf{L}} 1 \ \vdash\ P : \text{process}$$

We have

$$\Gamma \vdash @_l\mathbf{let}\ \langle y_1 : 1, y_2 : \updownarrow_{-\mathsf{L}} 1\rangle = \langle\langle\rangle, z\rangle\ \mathbf{in}\ P : \mathrm{process}$$

as $\mathrm{colocaln}(\Gamma, \langle\langle\rangle, z\rangle, 1 \times \updownarrow_{-\mathsf{L}} 1, 1 \times \updownarrow_{\mathsf{LG}} 1) = \{\}$, but

$$\Gamma \not\vdash @_l\mathbf{let}\ \langle y_1 : 1, y_2 : \updownarrow_{\mathsf{LG}} 1\rangle = \langle\langle\rangle, z\rangle\ \mathbf{in}\ P : \mathrm{process}$$

as $z \in \mathrm{colocaln}(\Gamma, \langle\langle\rangle, z\rangle, 1 \times \updownarrow_{\mathsf{LG}} 1, 1 \times \updownarrow_{\mathsf{LG}} 1)$ and $\Gamma \not\vdash z@l$, so

$$\Gamma \not\vdash @_l\mathbf{let}\ \langle y_1, y_2\rangle = \langle\langle\rangle, z\rangle\ \mathbf{in}\ P : \mathrm{process}$$

In the input construct the type of the pattern can be left implicit, as it is bounded by the type of the channel.

## 4 Soundness

This section gives the outline of the proof of subject reduction, organized as follows. Section 4.1 contains preliminary results about the unfolding of recursive types and Section 4.2 contains results about the judgements for context formation, kinding of types and typing of values, showing how these are preserved by context permutation and weakening. These are perhaps best skimmed on a first reading. Section 4.3 shows that the coinductive definition of subtyping is well-formed, and that its quotient by a recursion-unfolding equivalence is a partial order. Section 4.4 states the essential properties of colocality, together with its preservation under context permutation and weakening. Section 4.5 shows the preservation of process typing under context permutation and weakening.

Sections 4.6, 4.7 and 4.8 are the core of the soundness proof. We must show that process typing is preserved by *relocation* (occurring in migration reductions), *narrowing* (occurring in communication reductions, when variables are instantiated by values of subtypes) and *substitution* (occurring in communication and pair-splitting reductions). The main soundness result shows that typing is preserved by reduction.

**Theorem 1 (Subject reduction)** *If* $\Gamma \vdash P : \mathrm{process}$ *and* $P \longrightarrow Q$ *then* $\Gamma \vdash Q : \mathrm{process}$.

This is restated and proved in Section 4.9. The proofs of individual lemmas are in Appendix A, in corresponding subsections. In addition, we state an immediate-soundness result capturing the most interesting property guaranteed by the type system, that no well-typed process can immediately use a local capability outside its location.

**Proposition 2** *If* $\Gamma, \Delta \vdash x : \updownarrow_{io} T$ *and* $\Gamma, \Delta \vdash x@k$ *then*

1. *if* $\Gamma \vdash (\mathbf{new}\ \Delta)(@_l\overline{x}v \mid Q) : \mathrm{process}$ *and* $o = \mathsf{L}$ *then* $k = l$.
2. *if* $\Gamma \vdash (\mathbf{new}\ \Delta)(@_l x(y).P \mid Q) : \mathrm{process}$ *and* $i = \mathsf{L}$ *then* $k = l$.
3. *if* $\Gamma \vdash (\mathbf{new}\ \Delta)(@_l\, !x(y).P \mid Q) : \mathrm{process}$ *and* $i = \mathsf{L}$ *then* $k = l$.

PROOF Straightforward examination of the typing rules. □

### 4.1 Recursive types

**Lemma 3**

1. *If* $X$ guarded in $T$ *and* $T \Downarrow S$ *then* $S \neq X$.
2. *If* $T$ guarded *then there exists* $S$ *such that* $T \Downarrow S$.

3. *If* $T \Downarrow S$ *and* $S \neq Y$ *then* $\{U/Y\}T \Downarrow \{U/Y\}S$.
4. *If* $T \Downarrow S$ *and* $T \Downarrow S'$ *then* $S = S'$.
5. *If* $T$ guarded *then there exists a unique* $S$ *such that* $T \Downarrow S$.

**Lemma 4**

1. *If* $X \notin \mathrm{ftv}(T)$ *then* $X$ guarded in $T$.
2. *If* $Y$ guarded in $T$ *and* $Y$ guarded in $U$ *then* $Y$ guarded in $\{U/X\}T$.
3. *If* $S$ guarded *and* $U$ guarded *then* $\{U/X\}S$ guarded.
4. *If* $Y$ guarded in $\{U/X\}T$ *and* $Y \notin \{X\} \cup \mathrm{ftv}(U)$ *then* $Y$ guarded in $T$.
5. *If* $\{U/X\}S$ guarded *then* $S$ guarded *and* $X \in \mathrm{ftv}(S) \Rightarrow U$ guarded.
6. *If* $T \Downarrow S$ *then* $\mathrm{ftv}(T) = \mathrm{ftv}(S)$.
7. *If* $T \Downarrow S$ *then* $T$ guarded $\iff$ $S$ guarded.

**Lemma 5**

1. *If* $T \Downarrow S$ *then there do not exist* $X$ *and* $S_1$ *such that* $S = \mu X\ S_1$.
2. *If* $T \Downarrow S$ *then* $S \Downarrow S$.

## 4.2 Kinds, Contexts, Types and Values

**Lemma 6** *If* $\Gamma \vdash T : K$ *then*

1. $\vdash \Gamma$ *ok*.
2. $T$ guarded.
3. *There exists a unique* $S$ *such that* $T \Downarrow S$.

**Lemma 7** *If* $\Gamma \vdash v : T$ *then*

1. *If* $\Gamma \vdash v : T'$ *then* $T = T'$.
2. $\vdash \Gamma$ *ok*.
3. $T \Downarrow T$.

In the rest of this subsection we show that the basic typing judgements are invariant under legitimate permutations and weakening of contexts. The results should be unsurprising, although the mutual dependence of the judgements requires that some care be taken in the order in which they are proved. We define a permutation equivalence on pre-contexts by $\Delta \cong \Gamma$ iff:

(I) $\vdash \Gamma$ *ok* $\iff \vdash \Delta$ *ok*.

(II) $\Gamma \vdash X : K \iff \Delta \vdash X : K$

(III) $\Gamma \vdash x : T \iff \Delta \vdash x : T$.

(IV) $\Gamma \vdash x@l \iff \Delta \vdash x@l$.

In order to show that this is closed under weakening we define an auxiliary preorder on pre-contexts, saying $\Delta \lesssim \Gamma$ iff:

(i) $\vdash \Gamma$ *ok* $\Rightarrow \mathrm{dom}(\Gamma) = \mathrm{dom}(\Delta)$.

(ii) $\vdash \Gamma\ ok \Rightarrow \vdash \Delta\ ok$.

(iii) $\Gamma \vdash X : K \Rightarrow \Delta \vdash X : K$

(iv) $\Gamma \vdash x : \mathrm{loc} \Rightarrow \Delta \vdash x : \mathrm{loc}$.

(This will also used to show the basic properties of relocation and narrowing.)

**Lemma 8 (Weakening — Contexts)** *If* $\Delta \lesssim \Gamma$ *then*

1. $\Delta, X : K \lesssim \Gamma, X : K$.
2. $\Gamma \vdash T : K \Rightarrow \Delta \vdash T : K$.
3. $\Delta, x : @_l T \lesssim \Gamma, x : @_l T$.
4. *if* $\vdash \Gamma, \Theta$ ok *then* $\Delta, \Theta \lesssim \Gamma, \Theta$ *and* $\vdash \Delta, \Theta$ ok.

**Lemma 9 (Permutation — Types and Values)** *If* $\Delta \cong \Gamma$ *then:*

1. $\Delta \lesssim \Gamma$.
2. $\vdash \Delta, \Theta$ ok $\iff \vdash \Gamma, \Theta$ ok.
3. $\Delta, \Theta \cong \Gamma, \Theta$.
4. $\Delta \vdash T : K \iff \Gamma \vdash T : K$.
5. $\Delta \vdash v : T \iff \Gamma \vdash v : T$.

**Lemma 10** $\Gamma, X : K, X' : K' \cong \Gamma, X' : K', X : K$.

**Lemma 11 (Weakening — Types, Values and Locations — by type variable bindings)**

1. $\vdash \Gamma, X : K\ ok \wedge \Gamma \vdash T : K' \iff \Gamma, X : K \vdash T : K' \wedge X \notin \mathrm{ftv}(T)$
2. $\vdash \Gamma, X : K\ ok \wedge \Gamma \vdash v : T \iff \Gamma, X : K \vdash v : T$
3. $\vdash \Gamma, X : K\ ok \wedge \Gamma \vdash x@l \iff \Gamma, X : K \vdash x@l$

**Lemma 12** *If* $X \notin \mathrm{ftv}(T)$ *then* $\Gamma, x : @_l T, X : K \cong \Gamma, X : K, x : @_l T$.

**Lemma 13 (Weakening — Types, Values and Locations — by name bindings)**

1. $\vdash \Gamma, x : @_l T\ ok \wedge \Gamma \vdash S : K \iff \Gamma, x : @_l T \vdash S : K$
2. $\vdash \Gamma, x : @_l T\ ok \wedge \Gamma \vdash v : V \iff \Gamma, x : @_l T \vdash v : V \wedge x \notin \mathrm{fn}(v)$
3. $\vdash \Gamma, x : @_l T\ ok \wedge \Gamma \vdash y@k \iff \Gamma, x : @_l T \vdash y@k \wedge x \neq y$

The equivalence $\cong$ on pre-contexts contains all legitimate permutations. To make this precise we take the relation $\equiv$ on pre-contexts to be the least equivalence satisfying

$$\begin{array}{rcll} \Gamma, X : K, X' : K', \Delta & \equiv & \Gamma, X' : K', X : K, \Delta & \\ \Gamma, x : @_l T, X : K, \Delta & \equiv & \Gamma, X : K, x : @_l T, \Delta & X \notin \mathrm{ftv}(T) \\ \Gamma, x : @_k S, y : @_l T, \Delta & \equiv & \Gamma, y : @_l T, x : @_k S, \Delta & x \neq l \wedge y \neq k \end{array}$$

**Lemma 14** *If* $\Gamma \equiv \Delta$ *then* $\Gamma \cong \Delta$.

The implication is strict, as $\cong$ allows unfolding of recursion and relates any two ill-formed pre-contexts. For example $x : @_{\mathrm{top}} \mu X\ 1 \cong x : @_{\mathrm{top}} 1$.

Type formation is invariant under the unfolding of recursive types.

**Lemma 15**

1. *If* $\Gamma, X : K \vdash T : K'$ *and* $\Gamma \vdash U : K$ *then* $\Gamma \vdash \{U/X\}T : K'$.
2. *If* $\Gamma \vdash \{U/X\}S : K$ *and* $X \in \mathrm{ftv}(S) - \mathrm{dom}(\Gamma)$ *then there exists* $K'$ *such that* $\Gamma \vdash U : K'$ *and* $\Gamma, X : K' \vdash S : K$.
3. *If* $\Gamma \vdash T : K_1$ *and* $\Gamma \vdash T : K_2$ *then* $\Gamma \vdash T : K_1 \wedge K_2$.
4. *If* $T \Downarrow S$ *then* $\Gamma \vdash T : K \iff \Gamma \vdash S : K$.

**Lemma 16** *If* $\Gamma \vdash v : T$ *then* $\Gamma \vdash T : \mathrm{Type}_{\_\_}$.

## 4.3 Subtyping

The operator $\dot{\_}$ is monotonic and well behaved.

**Lemma 17**

1. *If* $\leqslant_1 \subseteq \leqslant_2$ *then* $\dot{\leqslant}_1 \subseteq \dot{\leqslant}_2$.
2. $\mathbf{id} \subseteq \dot{\mathbf{id}}$
3. *If* $\leqslant \subseteq \dot{\leqslant}$ *then* $\leqslant ; \leqslant \subseteq (\leqslant ; \leqslant)^{\cdot}$.

The subtype relation can therefore be defined as the greatest fixed point, taking $\leqslant \overset{def}{=} \cup\{\, R \mid R \subseteq \dot{R}\,\}$. It is a preorder.

**Lemma 18** $\leqslant = \dot{\leqslant}$ *and* $\leqslant$ *is a preorder.*

Subtyping is invariant under the unfolding of recursive types.

**Lemma 19** *For guarded pre-types* $T$ *and* $S$*, if* $T \Downarrow S$ *then* $T \leqslant S \leqslant T$.

We now define the natural equivalence relation over guarded pre-types, relating any two types that have the same infinitary unfolding. If $\equiv$ is a binary relation over the guarded pre-types then $\ddot{\equiv}$ is the least binary relation over the guarded pre-types such that

$$\frac{}{B \ddot{\equiv} B} \qquad \frac{}{1 \ddot{\equiv} 1} \qquad \frac{S_1 \equiv T_1 \quad S_2 \equiv T_2}{S_1 \times S_2 \ddot{\equiv} T_1 \times T_2} \qquad \frac{}{\mathrm{loc} \ddot{\equiv} \mathrm{loc}}$$

$$\frac{}{\top \ddot{\equiv} \top} \qquad \frac{}{X \ddot{\equiv} X} \qquad \frac{S \equiv T}{\updownarrow_{io} S \ddot{\equiv} \updownarrow_{io} T} \qquad \frac{\mu X\ S \Downarrow S' \quad S' \ddot{\equiv} T}{\mu X\ S \ddot{\equiv} T} \qquad \frac{S \ddot{\equiv} T' \quad \mu X\ T \Downarrow T'}{S \ddot{\equiv} \mu X\ T}$$

The operator $\ddot{\_}$ is monotonic and well behaved.

**Lemma 20**

1. *If* $\equiv_1 \subseteq \equiv_2$ *then* $\ddot{\equiv}_1 \subseteq \ddot{\equiv}_2$.
2. $\mathbf{id} \subseteq \ddot{\mathbf{id}}$
3. *If* $\equiv \subseteq \ddot{\equiv}$ *then* $\equiv ; \equiv \subseteq (\equiv ; \equiv)^{\cdot\cdot}$.
4. *If* $\equiv \subseteq \ddot{\equiv}$ *then* $\equiv^{-1} \subseteq (\equiv^{-1})^{\cdot\cdot}$.

The equivalence relation can therefore be defined as the greatest fixed point, taking $\equiv \overset{def}{=} \cup\{\, R \mid R \subseteq \ddot{R}\,\}$.

**Lemma 21** $\equiv = \ddot{\equiv}$ *and* $\equiv$ *is an equivalence relation.*

The quotient of $\leqslant$ by $\equiv$ is a partial order.

**Proposition 22** $S \leqslant T \leqslant S$ *iff* $S \equiv T$.

## 4.4 Colocality

A type that is colocal with itself is local.

**Lemma 23** *If* $\mathrm{colocal}(\Gamma, T, T)$ *then* $\neg(\Gamma \vdash T : \mathrm{Type}_{\mathsf{G}-})$.

Colocality is convex-closed with respect to subtyping (for well-formed types).

**Lemma 24** *If* $T_1 \leqslant T_2 \leqslant T_3$, $\mathrm{colocal}(\Gamma, T_1, T_3)$ *and* $\Gamma \vdash T_2 : \mathrm{Type}_{--}$ *then* $\mathrm{colocal}(\Gamma, T_1, T_2)$ *and* $\mathrm{colocal}(\Gamma, T_2, T_3)$.

If a value has any colocal names with respect to two types then those types are colocal.

**Lemma 25** *If* $\mathrm{colocaln}(\Gamma, v, S, T) \neq \{\}$ *then* $\mathrm{colocal}(\Gamma, S, T)$.

The set of colocal names of a value varies contravariantly with the upper type.

**Lemma 26**

$$\frac{\begin{array}{l}\Gamma \vdash v : V \\ \Gamma \vdash S : \mathrm{Type}_{--} \\ \Gamma \vdash T : \mathrm{Type}_{--} \\ V \leqslant S \leqslant T\end{array}}{\mathrm{colocaln}(\Gamma, v, V, T) \subseteq \mathrm{colocaln}(\Gamma, v, V, S)}$$

The types of the colocal names of a value are themselves colocal.

**Lemma 27** *If* $x \in \mathrm{colocaln}(\Gamma, v, V, T)$ *then there exists* $S$ *such that* $\Gamma \vdash x : S$ *and* $\mathrm{colocal}(\Gamma, S, S)$.

**Lemma 28 (Weakening — Colocality)**

1. $\vdash \Gamma, X : K\ \mathit{ok}\ \wedge\ \mathrm{colocal}(\Gamma, U, V) \iff \mathrm{colocal}((\Gamma, X : K), U, V)\ \wedge\ X \notin \mathrm{ftv}(U) \cup \mathrm{ftv}(V)$
2. $\vdash \Gamma, x : @_l T\ \mathit{ok}\ \wedge\ \mathrm{colocal}(\Gamma, U, V) \iff \mathrm{colocal}((\Gamma, x : @_l T), U, V)$
3. *If* $\vdash \Gamma, x : @_l T\ \mathit{ok}$, $\Gamma \vdash u : U$, $\Gamma \vdash V : \mathrm{Type}_{--}$ *and* $U \leqslant V$ *then*

$$\mathrm{colocaln}((\Gamma, x : @_l T), u, U, V) = \mathrm{colocaln}(\Gamma, u, U, V)$$

**Lemma 29 (Permutation — Colocality)** *If* $\Delta \cong \Gamma$ *then*

1. $\mathrm{colocal}(\Delta, S, T) \iff \mathrm{colocal}(\Gamma, S, T)$.
2. $\mathrm{colocaln}(\Delta, v, S, T) = \mathrm{colocaln}(\Gamma, v, S, T)$.

## 4.5 Processes

**Lemma 30** *If* $\Gamma \vdash P : \mathrm{process}$ *then* $\vdash \Gamma\ \mathit{ok}$.

**Lemma 31 (Permutation — Processes)** *If* $\Gamma \equiv \Delta$ *then* $\Gamma \vdash P : \mathrm{process} \iff \Delta \vdash P : \mathrm{process}$.

**Lemma 32 (Weakening — Processes)**

$$\vdash \Gamma, x : @_l T\ \mathit{ok}\ \wedge\ \Gamma \vdash P : \mathrm{process} \iff \Gamma, x : @_l T \vdash P : \mathrm{process}\ \wedge\ x \notin \mathrm{fn}(P)$$

## 4.6 Relocation

In order to show the type soundness of migrations (reduction rule 4) we must show that the typing judgements are preserved by changes of the parents of locations. We define a relocation equivalence on pre-contexts by $\Gamma \simeq \Delta$ iff:

(a) $\vdash \Gamma \; ok \iff \vdash \Delta \; ok.$

(b) $\Gamma \vdash X : K \iff \Delta \vdash X : K$

(c) $\Gamma \vdash x : T \iff \Delta \vdash x : T$

(d) $(\Gamma \vdash x : T \wedge \mathrm{colocal}(\Gamma, T, T)) \Rightarrow (\Gamma \vdash x@l \iff \Delta \vdash x@l)$

(This gives a stronger result than required, as it allows any non-local name to be relocated.)

**Lemma 33 (Relocation)** *If* $\Delta \simeq \Gamma$ *then*

1. $\Delta \lesssim \Gamma$
2. $\vdash \Delta, \Theta \text{ ok} \iff \vdash \Gamma, \Theta \text{ ok}.$
3. $\Delta, \Theta \simeq \Gamma, \Theta.$
4. $\Delta \vdash T : K \iff \Gamma \vdash T : K.$
5. $\Delta \vdash v : T \iff \Gamma \vdash v : T.$
6. $\mathrm{colocal}(\Delta, S, T) \iff \mathrm{colocal}(\Gamma, S, T).$
7. $\mathrm{colocaln}(\Delta, v, S, T) = \mathrm{colocaln}(\Gamma, v, S, T).$
8. $\Delta \vdash P : \mathrm{process} \iff \Gamma \vdash P : \mathrm{process}.$

Lemma 33.8 is used in the migration case of the proof of Theorem 1 (subject reduction), on Page 31.

## 4.7 Narrowing

The narrowing preorder is defined over pre-contexts by $\Delta \leqslant \Gamma$ iff:

(1) $\vdash \Gamma \; ok \iff \vdash \Delta \; ok.$

(2) $\Gamma \vdash X : K \iff \Delta \vdash X : K$

(3) $\Gamma \vdash x@l \iff \Delta \vdash x@l$

(4) $(\Delta \vdash x : T \wedge \Gamma \vdash x : S) \Rightarrow T \leqslant S$

**Lemma 34 (Narrowing)** *If* $\Delta \leqslant \Gamma$ *then*

1. $\Delta \lesssim \Gamma$
2. $\vdash \Delta, \Theta \text{ ok} \iff \vdash \Gamma, \Theta \text{ ok}.$
3. $\Delta, \Theta \leqslant \Gamma, \Theta.$
4. $\Delta \vdash T : K \iff \Gamma \vdash T : K.$
5. *If* $\Gamma \vdash v : S$ *then* $\exists T \,.\, \Delta \vdash v : T \wedge T \leqslant S.$
6. $\mathrm{colocal}(\Delta, S, T) \iff \mathrm{colocal}(\Gamma, S, T).$

7.

$$\frac{\begin{array}{l}\Gamma \vdash v : S' \\ \Delta \vdash v : T' \\ T' \leqslant S' \leqslant S\end{array}}{\mathrm{colocaln}(\Delta, v, T', S) \subseteq \mathrm{colocaln}(\Gamma, v, S', S)}$$

8. *If* $\Gamma \vdash P : \mathrm{process}$ *then* $\Delta \vdash P : \mathrm{process}$.

Lemma 34.8 is used in the (REP-)IN case of the proof of Lemma 38 (the substitution lemma for processes), on page 52.

## 4.8 Substitution

The set of inhabited locations of a process is unchanged by legitimate substitutions.

**Lemma 35** *If* $z \notin \mathrm{fl}(P)$ *then* $\mathrm{fl}(\{u/z\}P) = \mathrm{fl}(P)$.

The typing judgements must be preserved under substitution of values (of some type) for names (assumed to be of some supertype). For the value typing judgement this is straightforward.

**Lemma 36 (Substitution — Values)**

$$\frac{\begin{array}{l}\Gamma, z : @_j V \vdash w : S \\ \Gamma \vdash u : U \\ U \leqslant V\end{array}}{\exists T \,.\, \Gamma \vdash \{u/z\}w : T \wedge T \leqslant S}$$

For colocality it is more involved. In the following we write $\{A/z\}B$ for the set substitution $(B - \{z\}) \cup \{\, a \mid a \in A \wedge z \in B \,\}$.

**Lemma 37 (Substitution — Colocality)**

$$\frac{\begin{array}{lll}\Gamma \vdash u : U & \Gamma, z : @_j V \vdash v : S' & \\ \Gamma \vdash V : \mathrm{Type}_{--} & \Gamma, z : @_j V \vdash S : \mathrm{Type}_{--} & \Gamma \vdash \{u/z\}v : T' \\ U \leqslant V & S' \leqslant S & \end{array}}{\mathrm{colocaln}(\Gamma, \{u/z\}v, T', S) \subseteq \{\mathrm{colocaln}(\Gamma, u, U, V)/z\}\mathrm{colocaln}((\Gamma, z : @_j V), v, S', S)}$$

For processes the following holds. The first three premises are standard. The fourth ensures that any names of the substituted value $u$ are located at the same place as the substituted variable $z$ was assumed to be at, if their actual type and assumed types are colocal. The last premise ensures that no locators in $P$ can be affected by the substitution.

**Lemma 38 (Substitution — Processes)**

$$\frac{\begin{array}{l}\Gamma, z : @_j V \vdash P : \mathrm{process} \\ \Gamma \vdash u : U \\ U \leqslant V \\ \forall a \in \mathrm{colocaln}(\Gamma, u, U, V) \,.\, \Gamma \vdash a@j \\ z \notin \mathrm{fl}(P)\end{array}}{\Gamma \vdash \{u/z\}P : \mathrm{process}}$$

Lemma 38 is used in the communication and pair-splitting cases of the proof of Theorem 1 (subject reduction).

## 4.9 Subject reduction

The derived typing rule for multiple new-bindings is useful.

**Lemma 39** *If* $\mathrm{dom}(\Delta) \cap \mathrm{TVar} = \{\}$ *then*

$$(\vdash \Gamma, \Delta\ ok \wedge \Gamma \vdash (\mathbf{new}\ \Delta)P : \mathrm{process}) \iff \big(\forall T \in \mathrm{range}(\Delta)\ .\ \Gamma \vdash T : \mathrm{Type}_{-\mathsf{E}} \wedge \Gamma, \Delta \vdash P : \mathrm{process}\big)$$

**Lemma 40 (Type soundness of structural congruence)** *If* $P \equiv Q$ *then*

1. $\mathrm{fl}(P) = \mathrm{fl}(Q)$
2. $\Gamma \vdash P : \mathrm{process} \iff \Gamma \vdash Q : \mathrm{process}.$

**Theorem 1 (Subject reduction)** *If* $\Gamma \vdash P : \mathrm{process}$ *and* $P \longrightarrow Q$ *then* $\Gamma \vdash Q : \mathrm{process}.$

Proof By induction on transition derivations

**Rule 1** We have, using Lemma 7.1 on the type of $x$,

$$\textsc{Out}\ \frac{\begin{array}{l}\Gamma \vdash k : \mathrm{loc}\\ \Gamma \vdash x : \updownarrow_{io} T\\ \Gamma \vdash v : T'\\ T' \leqslant T\\ o \leqslant \mathsf{L}\\ o = \mathsf{L} \Rightarrow \Gamma \vdash x@k\\ \forall a \in \mathrm{colocaln}(\Gamma, v, T', T)\ .\ \Gamma \vdash a@k\end{array}}{\Gamma \vdash @_k \overline{x} v : \mathrm{process}} \qquad (\textsc{Rep-})\textsc{In}\ \frac{\begin{array}{l}\Gamma \vdash l : \mathrm{loc}\\ \Gamma \vdash x : \updownarrow_{io} T\\ \Gamma, y : @_l T \vdash P : \mathrm{process}\\ i \leqslant \mathsf{L}\\ i = \mathsf{L} \Rightarrow \Gamma \vdash x@l\\ \mathrm{fl}(P) \subseteq \{l\}\end{array}}{\begin{array}{l}\Gamma \vdash @_l x(y).P : \mathrm{process}\\ \Gamma \vdash @_l\, !x(y).P : \mathrm{process}\end{array}}$$

and need $\Gamma \vdash \{v/y\}P : \mathrm{process}$. We use the instance of the substitution lemma below.

$$\frac{\begin{array}{l}\Gamma, y : @_l T \vdash P : \mathrm{process}\\ \Gamma \vdash v : T'\\ T' \leqslant T\\ \forall a \in \mathrm{colocaln}(\Gamma, v, T', T)\ .\ \Gamma \vdash a@l\\ y \notin \mathrm{fl}(P)\end{array}}{\Gamma \vdash \{v/y\}P : \mathrm{process}}$$

The first three premises are trivial. For the fourth, if $a \in \mathrm{colocaln}(\Gamma, v, T', T)$ then by the premises of Out $\Gamma \vdash a@k$. Now, by Lemmas 25, 24 and 23 we have $\neg(\Gamma \vdash T : \mathrm{Type}_{\mathsf{G}-})$. By Lemma 16 $\Gamma \vdash \updownarrow_{io} T : \mathrm{Type}_{--}$ so by the typing rules $io \in \{\mathsf{LL}, -\mathsf{L}, \mathsf{L}-\}$ and hence $io = \mathsf{LL}$. By the premises of Out and (Rep-)In $k = l$ so $\Gamma \vdash a@l$.

For the fifth, by Lemma 30 $\vdash \Gamma, y : @_l T\ ok$ so $y \neq l$ so, as $\mathrm{fl}(P) \subseteq \{l\}$, $y \notin \mathrm{fl}(P)$.

**Rule 2** Similar

**Rule 3** We have

$$\textsc{Let}\ \frac{\begin{array}{l}\Gamma \vdash l : \mathrm{loc}\\ \Gamma \vdash \langle v_1, v_2\rangle : T'\\ \Gamma, y_1 : @_l T_1, y_2 : @_l T_2 \vdash P : \mathrm{process}\\ T' \leqslant T_1 \times T_2\\ \forall a \in \mathrm{colocaln}(\Gamma, \langle v_1, v_2\rangle, T', T_1 \times T_2)\ .\ \Gamma \vdash a@l\\ \mathrm{fl}(P) \subseteq \{l\}\end{array}}{\Gamma \vdash @_l \mathbf{let}\ \langle y_1 : T_1, y_2 : T_2\rangle = \langle v_1, v_2\rangle\ \mathbf{in}\ P : \mathrm{process}}$$

and need $\Gamma \vdash \{v_1/y_1\}\{v_2/y_2\}P : \mathrm{process}$. By $\Gamma \vdash \langle v_1, v_2 \rangle : T'$ and the value typing rules there exist $T'_1$ and $T'_2$ such that $T' = T'_1 \times T'_2$, $\Gamma \vdash v_1 : T'_1$ and $\Gamma \vdash v_2 : T'_2$. We use the instances of the substitution lemma below.

$$\frac{\begin{array}{l}\Gamma, y_1 : @_l T_1, y_2 : @_l T_2 \vdash P : \mathrm{process}\\ \Gamma, y_1 : @_l T_1 \vdash v_2 : T'_2\\ T'_2 \leqslant T_2\\ \forall a \in \mathrm{colocaln}(\Gamma, y_1 : @_l T_1, v_2, T'_2, T_2)\ .\ \Gamma, y_1 : @_l T_1 \vdash a@l\\ y_2 \notin \mathrm{fl}(P)\end{array}}{\Gamma, y_1 : @_l T_1 \vdash \{v_2/y_2\}P : \mathrm{process}} \qquad \frac{\begin{array}{l}\Gamma, y_1 : @_l T_1 \vdash \{v_2/y_2\}P : \mathrm{process}\\ \Gamma \vdash v_1 : T'_1\\ T'_1 \leqslant T_1\\ \forall a \in \mathrm{colocaln}(\Gamma, v_1, T'_1, T_1)\ .\ \Gamma \vdash a@l\\ y_1 \notin \mathrm{fl}(\{v_2/y_2\}P)\end{array}}{\Gamma \vdash \{v_1/y_1\}\{v_2/y_2\}P : \mathrm{process}}$$

By Lemma 30 $\vdash \Gamma, y_1 : @_l T_1, y_2 : @_l T_2\ ok$ (*). By the context formation rules $\vdash \Gamma, y_1 : @_l T_1\ ok$. By Lemma 13.2 $\Gamma, y_1 : @_l T_1 \vdash v_2 : T'_2$.

By (*) and the context formation rules $l \notin \{y_1, y_2\}$ so as $\mathrm{fl}(P) \subseteq \{l\}$ we have $y_1 \notin \mathrm{fl}(P)$ and $y_2 \notin \mathrm{fl}(P)$.

By Lemma 18 and the subtyping rules $T'_2 \leqslant T_2$ and $T'_1 \leqslant T_1$.

By a routine induction $\mathrm{colocaln}(\Gamma, v_1, T'_1, T_1) \cup \mathrm{colocaln}(\Gamma, v_2, T'_2, T_2) = \mathrm{colocaln}(\Gamma, \langle v_1, v_2 \rangle, T'_1 \times T'_2, T_1 \times T_2)$ so we have $\forall a \in \mathrm{colocaln}(\Gamma, v_1, T'_1, T_1)\ .\ \Gamma \vdash a@l$ and $\forall a \in \mathrm{colocaln}(\Gamma, v_2, T'_2, T_2)\ .\ \Gamma \vdash a@l$. By Lemma 28.3 and Lemma 13.3 we have $\forall a \in \mathrm{colocaln}((\Gamma, y_1 : @_l T_1), v_2, T'_2, T_2)\ .\ (\Gamma, y_1 : @_l T_1) \vdash a@l$. Hence, by the two instances of Lemma 38 above, we have $\Gamma \vdash \{v_1/y_1\}\{v_2/y_2\}P : \mathrm{process}$.

**Rule 4** We have

$$\Gamma \vdash (\textbf{new}\ l : @_j T)(\textbf{new}\ \Delta)(Q \mid @_l \textbf{migrate\_to}\ k\ \textbf{then}\ P) : \mathrm{process}$$

and $\{k, l\} \cap \mathrm{dom}(\Delta) = \{\} \wedge k \neq l$ and need

$$\Gamma \vdash (\textbf{new}\ l : @_k T)(\textbf{new}\ \Delta)(Q \mid P) : \mathrm{process}$$

By the definition of the reduction rule $\mathrm{dom}(\Delta) \cap \mathrm{TVar} = \{\}$ so $\mathrm{dom}(l : @_j T, \Delta) \cap \mathrm{TVar} = \{\}$.

We can assume w.l.g. (alpha converting if necessary) that $\vdash \Gamma, l : @_j T, \Delta\ ok$.

By Lemma 39 $\Gamma, l : @_j T, \Delta \vdash Q \mid @_l \textbf{migrate\_to}\ k\ \textbf{then}\ P : \mathrm{process}$ and for all $S \in \mathrm{range}(l : @_j T, \Delta)$ we have $\Gamma \vdash S : \mathrm{Type}_{-\mathsf{E}}$.

By the typing rules

$$\begin{array}{l}\Gamma, l : @_j T, \Delta \vdash Q : \mathrm{process}\\ \Gamma, l : @_j T, \Delta \vdash P : \mathrm{process}\\ \Gamma, l : @_j T, \Delta \vdash l : \mathrm{loc}\\ \Gamma, l : @_j T, \Delta \vdash k : \mathrm{loc}\end{array}$$

and $\mathrm{fl}(P) \subseteq \{l\}$.

By Lemma 13.2 and $k \notin \mathrm{dom}(\Delta) \cup \{l\}$ we have $\Gamma \vdash k : \mathrm{loc}$ so $\vdash \Gamma, l : @_k T\ ok$.

By the name typing rules $T \Downarrow \mathrm{loc}$ so $\neg\mathrm{colocal}(\Gamma, T, T)$. Clearly $\Gamma, l : @_j T \simeq \Gamma, l : @_k T$ so by Lemmas 33.2 and 33.3 $\vdash \Gamma, l : @_k T, \Delta\ ok$ and $\Gamma, l : @_j T, \Delta \simeq \Gamma, l : @_k T, \Delta$.

By Lemma 33.8

$$\begin{array}{l}\Gamma, l : @_k T, \Delta \vdash Q : \mathrm{process}\\ \Gamma, l : @_k T, \Delta \vdash P : \mathrm{process}\end{array}$$

Finally by Lemma 39 $\Gamma \vdash (\textbf{new}\ l : @_k T)(\textbf{new}\ \Delta)(Q \mid P) : \mathrm{process}$.

**Rule 5** By induction.

**Rule 6** By induction.

**Rule 7** By Lemma 40 and induction.

□

## 5 Conclusion

We conclude by briefly mentioning some related work and some possible future work.

**Related calculi** The distributed $\pi$-calculus presented here shares the notion of tree-structured locations with both the Distributed Join Calculus [FGL+96] and the Ambient Calculus of Cardelli and Gordon [CG97]. The communication primitives differ, however. The Distributed Join Calculus primitives are more restricted than those of dpi, in that the receivers on a channel must be replicated and declared with the channel; they are more general in that multi-way synchronisation is incorporated. In both calculi communication can occur across the location structure. This expressiveness is desirable for programming migratory applications within a single administrative domain. In contrast, the focus of the Ambient Calculus is the modeling of many administrative domains. Primitive interactions are only permitted between ambients that are adjacent (parent-child, or sibling) in the tree structure, so that all crossings of domain boundaries are explicit.

Several authors have considered distributed calculi with *site failure*. In [AP94] Amadio and Prasad give an extension of the $\pi$-calculus with nodes and node failure; they show it can be translated (up to notions of barbed equivalence) into a pure $\pi$-calculus. This has been developed further in [Ama97], using a type system that ensures receptiveness. In [FGL+96] it is shown that the Distributed Join Calculus, which includes primitives modelling the failure of subtrees of locations, can be encoded into the calculus without distribution primitives. In [RH97] Riely and Hennessy give direct characterisations of barbed bisimulation congruences for a CCS-like calculus extended with locations and failure.

Finally, in [SY97] Sekiguchi and Yonezawa give a calculus in which a number of different mechanisms for code and data migration can be described.

**Related type systems and static analyses** In [Ste96] Steckler has given a static analysis technique for distributed Poly/ML with similar motivation to ours — to detect when channels are guaranteed to be local to a single processor. It incorporates also some reachability analysis, but does not separate input and output capabilities. Nielson and Nielson [NN94, NN95] have proposed effect-based static analysis techniques for higher-order concurrent languages, taking effects to be process expressions. In [KNY95] Kobayashi et al propose a static analysis technique for the concurrent calculus HACL. It gives upper bounds for the sizes of queues required to implement communication, thereby allowing implementation optimization. Kleist et al [KHJH95] have proposed a type system based on process expression effects for $\lambda$inda. A type system for the Spi Calculus of Abadi and Gordon [AG97] has been proposed by Abadi in [Aba97]. It enforces freshness, guaranteeing that well-typed processes do not leak secrets in a rather strong sense. Finally in [HR97] Heintze and Riecke give a lambda calculus with a type system that enforces certain properties about direct and indirect information flow.

**Special cases** Three special cases of the type system may be of interest. In the Join calculus the names introduced by a definition **def** $D$ **in** $P$ can only be used for output in $P$ (to a first approximation $D$ declares a single replicated reader on these names). For typing $P$, therefore, they are analogous to channels with capability $-$G. One could allow the output capability to be local, taking the suborder of tags on the left below.

$$\begin{tikzcd}[arrows=dash]
-\mathsf{L} \arrow[dr] & \\
 & -\mathsf{G}
\end{tikzcd}
\qquad
\begin{tikzcd}[arrows=dash]
\mathsf{LL} \arrow[d] \\
\mathsf{GG}
\end{tikzcd}
\qquad
\begin{tikzcd}[arrows=dash]
 & \mathsf{L}- \arrow[dl] \arrow[dr] & & -\mathsf{L} \arrow[dl] \arrow[dr] & \\
\mathsf{G}- \arrow[drr] & & \mathsf{LL} \arrow[d] & & -\mathsf{G} \arrow[dll] \\
 & & \mathsf{GG} & &
\end{tikzcd}$$

In some circumstances it may not be necessary to allow the input and output capabilities of channels to vary separately, cutting down to the suborder of tags in the centre above. This greatly reduces the complexity (although also the expressiveness) of the type system as all channel type constructors become nonvariant and colocality collapses — for such tags

$$\mathrm{colocal}(io, i'o') \iff io = i'o' = \mathsf{LL}.$$

A milder simplification is to take the tags shown on the right above, i.e. the product of the tags $\pm, -, +$ of [PS96] with the two-point lattice $\mathsf{G} \leqslant \mathsf{L}$. For such tags, if $io \leqslant i'o'$ then

$$\mathrm{colocal}(io, i'o') \iff \mathrm{local}(io) \wedge \mathrm{local}(i'o')$$

Channels with tag GG would have to be used instead of channels with tag GL or LG. Whether the additional simplicity is worth the cost of this is not clear.

**Type inference** One would like a type inference result, allowing a compiler to infer the most local types possible for new-bound channels and let-bound names and hence to implement them efficiently, without requiring the programmer to provide too many explicit type annotations.

**Linearity** In a distributed application one would expect many channels to be in some sense *linear*; in particular many servers will have a single replicated receiver (this observation motivates the introduction of join patterns in [FG96]). In the type system given such channels, e.g. $\mathrm{print}$, $\mathrm{pair}$ and $\mathrm{getApplet}$ in examples 1, 3 and 4 of Section 2.3, can only be of types $\updownarrow_{\mathsf{LG}} T$. The integration of global/local typing with some form of linearity or receptiveness [Ama97, KPT96, San97] would allow more precise typing, and hence further optimizations, while retaining the expressiveness of general channel communication.

**Typing locations** In the type system given location names are all of type $\mathrm{loc}$. This has kind $\mathrm{Type}_{\mathsf{GE}}$ and so location names can be communicated freely between locations. It would be straightforward to refine the system to allow location names to be local, with types $\mathrm{loc}_{\mathsf{G}}$ and $\mathrm{loc}_{\mathsf{L}}$, enabling migration to locations to be restricted. One could also restrict the migration *of* locations, allowing locations to be *immobile* or *mobile*. This would be a (small) step towards the type-theoretic enforcement of security policies for mobile agents. Linearity would again be useful — a common case is that of *one-hop* locations (c.f. Java Applets).

**Dependent located types** The types of the system given are not dependent — they do not mention location names. One could instead take capabilities not simply from the set $\{\mathsf{G}, \mathsf{L}, -\}$ but from the set of location names extended with $\{\mathsf{G}, -\}$. This would allow types such as $\updownarrow_{\mathsf{GG}} \updownarrow_{kk} 1$, allowing local (to $k$) channels to be sent outside of their location, returned to $k$ and then used. More interestingly, one could take tags to be from the lattice of arbitrary sets of location names, lifted above a bottom element G, allowing optimized implementation of channels for which the set of locations at which they will be used can be bounded when the channel is created.

**Typing for secrecy properties** The focus of this paper has been on locality information that can be used for implementation optimization. The enforcement of certain secrecy properties, such as the invariant that a secret key is known only by authorised agents, is a closely related problem. The dependent types described above would allow new names (keys, for example) to be created that are restricted to a dynamically calculated set of individuals. For example, supposing a type $\mathrm{Key}$ of keys taking a single tag, one could express a location, modelling individual $a$, that receives the name $b$ of another individual and then creates a key $k$, which the type system guarantees is used (for decryption and encryption) only by $a$ and $b$:

$$@_a x(b).(\mathbf{new}\ k : \mathrm{Key}_{\{a,b\}}) \ldots$$

One would want a rather strong soundness result — the analogue of Theorem 1 would only show that secrecy is preserved by well-typed processes, whereas an attacker may perform some ill-typed computation. A slightly different calculus would be required. Primitives for encryption and decryption, such as in the Spi calculus of Abadi and Gordon [AG97] would have to be added. For some applications, taking locations to model individuals, the tree structure and migration of locations become superfluous.

**Behavioural equivalences** In order to reason about dpi processes a labelled transition system and behavioural congruence are required, perhaps building on the work of Riely and Hennessy [RH97], together with an understanding of the appropriate extensional equivalence, building on [Sew97].

**Acknowledgements** The author would like to thank Cédric Fournet, Benjamin Pierce, Pawel Wojciechowski and the Cambridge Thursday morning group, for interesting discussions about this work, and the organisers of Dagstuhl-Seminar 97041 for the opportunity to present a preliminary version. Support from EPSRC grant GR/K 38403 and Esprit Working group 21836 (CONFER-2) is acknowledged. Xy-pic was used for the diagrams.

# A Soundness proofs

## A.1 Recursive types

**Proof (of Lemma 3)**

**3.1** Induction on $T$. Cases $B, 1, T \times T', \mathrm{loc}, \top$ and $\updownarrow_{io} T$ are easy. Case $Y$ for $Y \neq X$ is easy. Case $X$ contradicts the first premise. For case $\mu Y\ T$ we have

$$\frac{X \text{ guarded in } T}{X \text{ guarded in } \mu Y\ T} \qquad \frac{T \Downarrow S \quad S \neq Y}{\mu Y\ T \Downarrow \{\mu Y\ T\ /\ Y\}S}$$

By induction $S \neq X$ so $\{\mu Y\ T\ /\ Y\}S \neq X$.

**3.2** Induction on $T$ guarded. All cases except $\mu X\ T$ are immediate. For $\mu X\ T$, by induction there exists $S$ such that $T \Downarrow S$. By Lemma 3.1 $S \neq X$ so $\mu X\ T \Downarrow \{\mu X\ T\ /\ X\}S$.

**3.3** Induction on $T \Downarrow S$. Cases $B, 1, \mathrm{loc}, \top, \updownarrow_{io} T, T \times T'$ and $X$ for $Y \neq X$ are easy.

Case $Y \Downarrow Y$ contradicts the premises.

For case $\mu X\ T$ we have

$$\frac{T \Downarrow S \quad S \neq X}{\mu X\ T \Downarrow \{\mu X\ T\ /\ X\}S} \qquad \{\mu X\ T\ /\ X\}S \neq Y$$

w.l.g. $X \notin \{Y\} \cup \mathrm{ftv}(U)$ so $\{U/Y\}\mu X\ T = \mu X\ \{U/Y\}T$. By the above $S \neq Y$ so by induction $\{U/Y\}T \Downarrow \{U/Y\}S$. We have $\{U/Y\}S \neq X$ so

$$\{U/Y\}\mu X\ T = \mu X\{U/Y\}T \Downarrow \{\mu X\ \{U/Y\}T\ /\ X\}\{U/Y\}S = \{U/Y\}\{\mu X\ T\ /\ X\}S.$$

**3.4** By induction on the size of $T$. All cases except $\mu X\ T$ are immediate. For $\mu X\ T$, suppose $\mu X\ T \Downarrow R$ and $\mu X\ T \Downarrow R'$. Then there exist $Y, Y' \notin \mathrm{ftv}(T) - \{X\}$ and $S, S'$ such that

$$\frac{\{Y/X\}T \Downarrow S \quad S \neq Y}{\mu Y\ \{Y/X\}T \Downarrow \{\mu Y\ \{Y/X\}T\ /\ Y\}S} \qquad \frac{\{Y'/X\}T \Downarrow S' \quad S' \neq Y'}{\mu Y'\ \{Y'/X\}T \Downarrow \{\mu Y'\ \{Y'/X\}T\ /\ Y'\}S'}$$

with $R = \{\mu Y\ \{Y/X\}T\ /\ Y\}S$ and $R' = \{\mu Y'\ \{Y'/X\}T\ /\ Y'\}S'$.

By Lemma 3.3 $T \Downarrow \{X/Y\}S$ and $T \Downarrow \{X/Y'\}S'$ so by induction $\{X/Y\}S = \{X/Y'\}S'$ and $R = R'$.

**3.5** Immediate from Lemmas 3.2 and 3.4.

□

**Proof (of Lemma 4)**

**4.1** Induction on $T$. Cases $B, 1, \mathrm{loc}, \top$ and $Y$ for $Y \neq X$ are easy. Case $T \times T'$ is by induction. Case $\updownarrow_{io} T$ is easy. Case $X$ contradicts the premise. Case $\mu Y\ T$ is by induction.

**4.2** Induction on $T$. Cases $B, 1, \mathrm{loc}, \top$ and $Z$ for $Z \notin \{X, Y\}$ are easy. Case $T \times T'$ is by induction. Case $\updownarrow_{io} T$ is easy. Case $Y$ contradicts the premise. Case $X$ is easy. Case $\mu Z\ T$ is by induction.

**4.3** Induction on $S$ guarded. Cases $B, 1, \mathrm{loc}, \top$ and $Y$ for $Y \neq X$ are easy. Cases $T \times T'$, and $\updownarrow_{io} T$ are by ind. Case $X$ is easy. The remaining case is

$$\frac{S \text{ guarded} \quad Y \text{ guarded in } S}{\mu Y\ S \text{ guarded}}$$

w.l.g. $Y \notin \{X\} \cup \mathrm{ftv}(U)$ so $\{U/X\}\mu Y\ S = \mu Y\ \{U/X\}S$. By induction $\{U/X\}S$ guarded. By Lemma 4.1 $Y$ guarded in $U$. By Lemma 4.2 $Y$ guarded in $\{U/X\}S$ so $\mu Y\ \{U/X\}S$ guarded.

**4.4** Induction on $T$. Cases $B, 1, \mathrm{loc}, \top$, $\updownarrow_{io} T$ and $Z$ for $Z \notin \{X, Y\}$ are easy. Case $T \times T'$ is by induction. Case $Y$ contradicts the premise. Case $X$ is by Lemma 4.1. Case $\mu Z\ T$: We have to show

$$\frac{Y \text{ guarded in } \{U/X\}\mu Z\ T \quad Y \notin \{X\} \cup \mathrm{ftv}(U)}{Y \text{ guarded in } \mu Z\ T}$$

w.l.g. $Z \notin \{X\} \cup \mathrm{ftv}(U)$ so $Y$ guarded in $\mu Z\ \{U/X\}T$ so $Y$ guarded in $\{U/X\}T$. By induction $Y$ guarded in $T$ so $Y$ guarded in $\mu Z\ T$.

**4.5** Induction on $S$. Cases $B, 1, \mathrm{loc}, \top$ and $Y$ for $Y \neq X$ are easy. Cases $T \times T'$ and $\updownarrow_{io} T$ are by induction. Case $X$ is easy. Case $\mu Y\ S$: w.l.g. $Y \notin \{X\} \cup \mathrm{ftv}(U)$ so we have

$$\frac{\{U/X\}S \text{ guarded} \quad Y \text{ guarded in } \{U/X\}S}{\mu Y\ \{U/X\}S \text{ guarded}}$$

If $X \in \mathrm{ftv}(\mu Y\ S)$ then $X \in \mathrm{ftv}(S)$ so by induction $U$ guarded. By Lemma 4.4 $Y$ guarded in $S$ so $\mu Y\ S$ guarded.

**4.6** Induction on $T \Downarrow S$. All cases except $\mu X\ T$ are immediate. For $\mu X\ T$:

Suppose $X \notin \mathrm{ftv}(S)$. We have $\mathrm{ftv}(\{\mu X\ T\ /\ X\}S) = \mathrm{ftv}(S) = \mathrm{ftv}(S) - \{X\} = \mathrm{ftv}(T) - \{X\} = \mathrm{ftv}(\mu X\ T)$.

Alternatively, suppose $X \in \mathrm{ftv}(S)$. We have $\mathrm{ftv}(\{\mu X\ T\ /\ X\}S) = \mathrm{ftv}(\{\mu X\ T) \cup (\mathrm{ftv}(S) - \{X\}) = (\mathrm{ftv}(T) - \{X\}) \cup (\mathrm{ftv}(S) - \{X\})$. By induction $(\mathrm{ftv}(T) - \{X\}) \cup (\mathrm{ftv}(S) - \{X\}) = (\mathrm{ftv}(T) - \{X\}) \cup (\mathrm{ftv}(T) - \{X\}) = \mathrm{ftv}(T) - \{X\} = \mathrm{ftv}(\mu X\ T)$.

**4.7** Induction on $T \Downarrow S$. All cases except $\mu X\ T$ are immediate. Consider

$$\frac{T \Downarrow S \quad S \neq X}{\mu X\ T \Downarrow \{\mu X\ T\ /\ X\}S}$$

Suppose $\mu X\ T$ guarded. By definition $T$ guarded $\wedge$ $X$ guarded in $T$. By induction $S$ guarded. By Lemma 4.3 $\{\mu X\ T\ /\ X\}S$ guarded.

Suppose $\{\mu X\ T\ /\ X\}S$ guarded. By Lemma 4.5 $S$ guarded $\wedge$ $X \in \mathrm{ftv}(S) \Rightarrow \mu X\ T$ guarded. Case $X \in \mathrm{ftv}(S)$: we have $\mu X\ T$ guarded. Case $X \notin \mathrm{ftv}(S)$: By Lemma 4.6 we have $X \notin \mathrm{ftv}(T)$. By Lemma 4.1 $X$ guarded in $T$. By induction $T$ guarded, so $\mu X\ T$ guarded.

□

**PROOF (of Lemma 5)**

**5.1** Induction on $T \Downarrow S$.

**5.2** By Lemma 5.1 and the definition of $\Downarrow$.

□

## A.2 Kinds, Contexts, Types and Values

**Proof (of Lemma 6)**

1. Induction on $\Gamma \vdash T : K$, using the fact that $\vdash \Gamma, X : K\ ok$ implies $\vdash \Gamma\ ok$.
2. Induction on $\Gamma \vdash T : K$.
3. Immediate from Lemmas 6.2 and 3.5.

□

**Proof (of Lemma 7)**

1. Induction on $v$. Names: By disjointness if $\Gamma \vdash x : T$ it must be by the name rule. If $\Gamma \vdash x : T_1$ and $\Gamma \vdash x : T_2$ then
$$\begin{array}{ll} \Gamma = \Gamma_1, x : @_{l_1} S_1, \Delta_1 & S_1 \Downarrow T_1 \\ \Gamma = \Gamma_2, x : @_{l_2} S_2, \Delta_2 & S_2 \Downarrow T_2 \end{array}$$
   By context formation $S_1 = S_2$ and $\Gamma_1 \vdash S_1 : K$ for some kind $K$. By Lemma 6.3 $T_1 = T_2$.

   Base values and unit: By disjointness.

   Pairs: By induction
2. Induction on $\Gamma \vdash v : T$.
3. Cases of $v$. Names are by Lemma 5.2; base values, unit and pairs are by definition of $\Downarrow$.

□

**Proof (of Lemma 8)**

**8.1** Part (i): If $\vdash \Gamma, X : K$ ok then $\vdash \Gamma$ ok so $\mathrm{dom}(\Gamma) = \mathrm{dom}(\Delta)$ so $\mathrm{dom}(\Gamma, X : K) = \mathrm{dom}(\Delta, X : K)$. Parts (ii), (iii) and (iv) are as follows.

$$\begin{array}{lll}
 & \vdash \Gamma, X : K \text{ ok} & \\
\iff & \vdash \Gamma \text{ ok} \wedge X \notin \mathrm{dom}(\Gamma) & \\
\Rightarrow & \vdash \Delta \text{ ok} \wedge X \notin \mathrm{dom}(\Delta) & \text{by (ii) and (i) for } \Delta \lesssim \Gamma \\
\iff & \vdash \Delta, X : K \text{ ok} &
\end{array}$$

$$\begin{array}{lll}
 & \Gamma, X : K \vdash X' : K' & \\
\iff & \vdash \Gamma, X : K \text{ ok} \wedge (\Gamma \vdash X' : K' \vee (X = X' \wedge K \leqslant K')) & \\
\Rightarrow & \vdash \Delta, X : K \text{ ok} \wedge (\Gamma \vdash X' : K' \vee (X = X' \wedge K \leqslant K')) & \text{by part (ii) above} \\
\Rightarrow & \vdash \Delta, X : K \text{ ok} \wedge (\Delta \vdash X' : K' \vee (X = X' \wedge K \leqslant K')) & \text{by part (iii) for } \Delta \lesssim \Gamma \\
\iff & \Delta, X : K \vdash X' : K' &
\end{array}$$

$$\begin{array}{lll}
 & \Gamma, X : K \vdash x : \mathrm{loc} & \\
\iff & \vdash \Gamma, X : K \text{ ok} \wedge \Gamma \vdash x : \mathrm{loc} & \\
\Rightarrow & \vdash \Delta, X : K \text{ ok} \wedge \Delta \vdash x : \mathrm{loc} & \text{by part (ii) above and part (iv) for } \Delta \lesssim \Gamma \\
\iff & \Delta, X : K \vdash x : \mathrm{loc} &
\end{array}$$

**8.2** Induction on $\Gamma \vdash T : K$. Cases $B$, $1$, $\mathrm{loc}$ and $\top$ are by (ii). Cases $T \times T'$, $\updownarrow_{io} T$ and kind subsumption are by induction. Case $X$ is by (iii). For $\mu X\ T$, suppose $\Gamma, X : K \vdash T : K$. By Lemma 8.1 $\Delta, X : K \lesssim \Gamma, X : K$ so by induction $\Delta, X : K \vdash T : K$.

**8.3** Part (i): If $\vdash \Gamma, x : @_l T$ ok then $\vdash \Gamma$ ok so $\mathrm{dom}(\Gamma) = \mathrm{dom}(\Delta)$ so $\mathrm{dom}(\Gamma, x : @_l T) = \mathrm{dom}(\Delta, x : @_l T)$. Parts (ii), (iii) and (iv) are as follows.

$$\begin{array}{lll}
 & \vdash \Gamma, x : @_l T \text{ ok} & \\
\iff & \exists K \,.\, \Gamma \vdash T : K \wedge (\Gamma \vdash l : \text{loc} \vee l = \text{top}) \wedge x \notin \mathrm{dom}(\Gamma) \cup \{\text{top}\} & \\
\Rightarrow & \exists K \,.\, \Delta \vdash T : K \wedge (\Delta \vdash l : \text{loc} \vee l = \text{top}) \wedge x \notin \mathrm{dom}(\Gamma) \cup \{\text{top}\} & \text{by 8.2 and (iv)} \\
\Rightarrow & \exists K \,.\, \Delta \vdash T : K \wedge (\Delta \vdash l : \text{loc} \vee l = \text{top}) \wedge x \notin \mathrm{dom}(\Delta) \cup \{\text{top}\} & \text{by 6.1 and (i)} \\
\iff & \vdash \Delta, x : @_l T \text{ ok} &
\end{array}$$

$$\begin{array}{lll}
 & \Gamma, x : @_l T \vdash X : K & \\
\iff & \vdash \Gamma, x : @_l T \text{ ok} \wedge \Gamma \vdash X : K & \\
\Rightarrow & \vdash \Delta, x : @_l T \text{ ok} \wedge \Gamma \vdash X : K & \text{by part (ii) above} \\
\Rightarrow & \vdash \Delta, x : @_l T \text{ ok} \wedge \Delta \vdash X : K & \text{by part (iii) for } \Delta \lesssim \Gamma \\
\iff & \Delta, x : @_l T \vdash X : K &
\end{array}$$

$$\begin{array}{lll}
 & \Gamma, x : @_l T \vdash x' : \text{loc} & \\
\iff & \vdash \Gamma, x : @_l T \text{ ok} \wedge (\Gamma \vdash x' : \text{loc} \vee (x = x' \wedge T \Downarrow \text{loc})) & \\
\Rightarrow & \vdash \Delta, x : @_l T \text{ ok} \wedge (\Gamma \vdash x' : \text{loc} \vee (x = x' \wedge T \Downarrow \text{loc})) & \text{by part (ii) above} \\
\Rightarrow & \vdash \Delta, x : @_l T \text{ ok} \wedge (\Delta \vdash x' : \text{loc} \vee (x = x' \wedge T \Downarrow \text{loc})) & \text{by part (iv) for } \Delta \lesssim \Gamma \\
\iff & \Delta, x : @_l T \vdash x' : \text{loc} &
\end{array}$$

**8.4** Induction on $\Theta$, using 8.1 and 8.3.

□

**Proof (of Lemma 9)**

**9.1** For part (i), if $\vdash \Gamma$ *ok* then $\vdash \Delta$ *ok* so using (II) and (III) $\mathrm{dom}(\Gamma) = \mathrm{dom}(\Delta)$. The other parts are trivial.

**9.2** Suppose $\Delta \cong \Gamma$ and $\vdash \Gamma, \Theta$ ok. By 9.1 $\Delta \lesssim \Gamma$ so by 8.4. $\vdash \Delta, \Theta$ ok.

**9.3** We show

- If $\Delta \cong \Gamma$ then $\Delta, X : K \cong \Gamma, X : K$.
- If $\Delta \cong \Gamma$ then $\Delta, x : @_l T \cong \Gamma, x : @_l T$.

The result is then a trivial induction on $\Theta$. For the first, part (I) is by 9.2. Parts (II) and (III) are as follows, part (IV) is similar to (III).

$$\begin{array}{lll}
 & \Gamma, X : K \vdash X' : K' & \\
\iff & \vdash \Gamma, X : K \text{ ok} \wedge (\Gamma \vdash X' : K' \vee (X = X' \wedge K \leqslant K')) & \\
\iff & \vdash \Delta, X : K \text{ ok} \wedge (\Gamma \vdash X' : K' \vee (X = X' \wedge K \leqslant K')) & \text{by 9.2} \\
\iff & \vdash \Delta, X : K \text{ ok} \wedge (\Delta \vdash X' : K' \vee (X = X' \wedge K \leqslant K')) & \text{by part (II) for } \Delta \cong \Gamma \\
\iff & \Delta, X : K \vdash X' : K' &
\end{array}$$

$$\begin{array}{lll}
 & \Gamma, X : K \vdash x : T & \\
\iff & \vdash \Gamma, X : K \text{ ok} \wedge \Gamma \vdash x : T & \\
\iff & \vdash \Delta, X : K \text{ ok} \wedge \Delta \vdash x : T & \text{by 9.2 and part (III) for } \Delta \cong \Gamma \\
\iff & \Delta, X : K \vdash x : T &
\end{array}$$

For the second, part (I) is by 9.2. Part (II) and (III) are as follows, part (IV) is similar to (III).

$$\begin{array}{lll}
 & \Gamma, x : @_l T \vdash X : K & \\
\iff & \vdash \Gamma, x : @_l T \text{ ok} \wedge \Gamma \vdash X : K & \\
\iff & \vdash \Delta, x : @_l T \text{ ok} \wedge \Gamma \vdash X : K & \text{by 9.2} \\
\iff & \vdash \Delta, x : @_l T \text{ ok} \wedge \Delta \vdash X : K & \text{by part (II) for } \Delta \cong \Gamma \\
\iff & \Delta, x : @_l T \vdash X : K &
\end{array}$$

$$\begin{array}{lll}
 & \Gamma, x : @_l T \vdash x' : T' & \\
\iff & \vdash \Gamma, x : @_l T \text{ ok} \wedge (\Gamma \vdash x' : T' \vee (x = x' \wedge T \Downarrow T')) & \\
\iff & \vdash \Delta, x : @_l T \text{ ok} \wedge (\Gamma \vdash x' : T' \vee (x = x' \wedge T \Downarrow T')) & \text{by 9.2} \\
\iff & \vdash \Delta, x : @_l T \text{ ok} \wedge (\Delta \vdash x' : T' \vee (x = x' \wedge T \Downarrow T')) & \text{by part (III) for } \Delta \cong \Gamma \\
\iff & \Delta, x : @_l T \vdash x' : T' &
\end{array}$$

**9.4** By 9.1 and 8.2.

**9.5** Induction on type derivations using (I) and (III).

□

**Proof (of Lemma 10)** Straightforward. □

**Proof (of Lemma 11)**

**11.1** $\Rightarrow$: Induction on $\Gamma \vdash T : K'$. Cases $B, 1, \top, \text{loc}$ are by $\vdash \Gamma, X : K \ ok$. Cases $T \times T'$, $\updownarrow_{io} T$ and kind subsumption are by induction. Case $Y$ is by the typing and context formation rules. Case $\mu Y\ T$. w.l.g. $Y \notin \text{dom}(\Gamma)$ so $\vdash \Gamma, Y : K'\ ok$. By the typing rules $Y$ guarded in $T$. By induction $\Gamma, Y : K', X : K \vdash T : K'$ and $X \notin \text{ftv}(T)$. By 10 and 9.4 $\Gamma, X : K, Y : K' \vdash T : K'$. By the typing rules $\Gamma, X : K \vdash \mu Y\ T : K'$.

$\Leftarrow$: The first conjunct is by 6.1, from which by the context formation rules $\vdash \Gamma\ ok$ and $X \notin \text{dom}(\Gamma)$. The second is by induction. Cases $B, 1, \top, \text{loc}$ are by $\vdash \Gamma\ ok$. Cases $T \times T'$, $\updownarrow_{io} T$ and kind subsumption are by induction. Case $Y$ for $Y \neq X$ is by $\vdash \Gamma\ ok$. Case $X$ contradicts the premise. For case $\mu Y\ T$ we have $Y$ guarded in $T$ and $\Gamma, X : K, Y : K' \vdash T : K'$. By 10 and 9.4 $\Gamma, Y : K', X : K \vdash T : K'$. By induction $\Gamma, Y : K' \vdash T : K'$. By the typing rules $\Gamma \vdash \mu Y\ T : K'$.

**11.2** Routine inductions, using 7.2

**11.3** Immediate from the location rule.

□

**Proof (of Lemma 12)** For part (I):

$$\begin{array}{ll}
 & \vdash \Gamma, x : @_l T, X : K\ ok \\
\iff & \vdash \Gamma, x : @_l T\ ok \wedge X \notin \text{dom}(\Gamma, x : @_l T) \\
\iff & \exists K' \,.\, \Gamma \vdash T : K' \wedge (\Gamma \vdash l : \text{loc} \vee l = \text{top}) \wedge x \notin \text{dom}(\Gamma) \cup \{\text{top}\} \wedge X \notin \text{dom}(\Gamma) \\
\iff & \exists K' \,.\, \Gamma, X : K \vdash T : K' \wedge (\Gamma, X : K \vdash l : \text{loc} \vee l = \text{top}) \wedge x \notin \text{dom}(\Gamma, X : K) \cup \{\text{top}\} \\
 & \quad \text{by 6.1, 11.1 and 11.2} \\
\iff & \vdash \Gamma, X : K, x : @_l T\ ok
\end{array}$$

Parts (II), (III) and (IV) are immediate from the typing rules and part (I). □

**Proof (of Lemma 13)**

**13.1** $\Rightarrow$: Induction on $\Gamma \vdash S : K$. Cases $B, 1, \top, \text{loc}, X$ are by $\vdash \Gamma, x : @_l T\ ok$. Cases $S \times S'$, $\updownarrow_{io} S$ and kind subsumption are by induction. Case $\mu Y\ S$. w.l.g. $Y \notin \text{dom}(\Gamma)$ so $\vdash \Gamma, Y : K\ ok$. By the typing rules $Y$ guarded in $S$. By induction $\Gamma, Y : K, x : @_l T \vdash S : K$. By 12 and 9.4 $\Gamma, x : @_l T, Y : K \vdash S : K$. By the typing rules $\Gamma, x : @_l T \vdash \mu Y\ S : K$.

$\Leftarrow$: The first conjunct is by 6.1, from which by the context formation rules $\vdash \Gamma\ ok$. The second is by induction. Cases $B, 1, \top, \mathrm{loc}, Y$ are by $\vdash \Gamma\ ok$. Cases $S \times S'$, $\updownarrow_{io} S$ and kind subsumption are by induction. For case $\mu Y\ S$ we have $Y$ guarded in $S$ and $\Gamma, x : @_l T, Y : K \vdash S : K$. By 12 and 9.4 $\Gamma, Y : K, x : @_l T \vdash S : K$. By induction $\Gamma, Y : K \vdash S : K$. By the typing rules $\Gamma \vdash \mu Y\ S : K$.

**13.2** Routine inductions, using 7.2.

**13.3** Immediate from the context formation and location rule.

□

**Proof (of Lemma 14)** We first show

$$\Gamma, x : @_k S, y : @_l T \cong \Gamma, y : @_l T, x : @_k S \qquad \text{if } x \neq l \wedge y \neq k$$

For part (I):

$$\begin{array}{lr}
\vdash \Gamma, x : @_k S, y : @_l T\ ok & \\
\iff \exists K' .\ \Gamma, x : @_k S \vdash T : K' \wedge (\Gamma, x : @_k S \vdash l : \mathrm{loc} \vee l = \mathrm{top}) \wedge y \notin \mathrm{dom}(\Gamma, x : @_k S) \cup \{\mathrm{top}\} & \\
\iff \exists K' .\ \Gamma \vdash T : K' \wedge (\Gamma, x : @_k S \vdash l : \mathrm{loc} \vee l = \mathrm{top}) \wedge y \notin \mathrm{dom}(\Gamma) \cup \{x, \mathrm{top}\} \wedge \vdash \Gamma, x : @_k S\ ok & 13.1 \\
\iff \exists K' .\ \Gamma \vdash T : K' \wedge (\Gamma \vdash l : \mathrm{loc} \vee l = \mathrm{top}) \wedge y \notin \mathrm{dom}(\Gamma) \cup \{x, \mathrm{top}\} \wedge \vdash \Gamma, x : @_k S\ ok & 13.2 \\
\iff \exists K, K' .\ \Gamma \vdash T : K' \wedge (\Gamma \vdash l : \mathrm{loc} \vee l = \mathrm{top}) \wedge y \notin \mathrm{dom}(x, \Gamma) \cup \{\mathrm{top}\} & \\
\qquad\qquad \wedge\ \Gamma \vdash S : K \wedge (\Gamma \vdash k : \mathrm{loc} \vee k = \mathrm{top}) \wedge x \notin \mathrm{dom}(\Gamma) \cup \{\mathrm{top}\} & \\
\iff \vdash \Gamma, y : @_l T, x : @_k S\ ok & \text{sym.}
\end{array}$$

Parts (II), (III) and (IV) are immediate from the typing rules and part (I).

The result follows from this, 10, 12 and 9.3. □

**Proof (of Lemma 15)**

**15.1** Induction on $\Gamma, X : K \vdash T : K'$. Cases $B, 1, \mathrm{loc}, \top$ and $Y$, for $Y \neq X$, are trivial. Cases $T_1 \times T_2$, $\updownarrow_{io} T_1$ and subsumption are by induction. Case $X$. By the typing rules $K = K'$. Case $\mu Y\ T_1$. By the typing rules $\Gamma, X : K, Y : K' \vdash T_1 : K'$ and $Y$ guarded in $T_1$. By Lemmas 10 and 9.4 $\Gamma, Y : K', X : K \vdash T_1 : K'$. By induction $\Gamma, Y : K' \vdash \{U/X\}T_1 : K'$. By Lemmas 11.1 and 4.1 $Y$ guarded in $U$. By Lemma 4.2 $Y$ guarded in $\{U/X\}T_1$. By the typing rules $\Gamma \vdash \mu Y\ (\{U/X\}T_1) : K'$. Finally $\mu Y\ (\{U/X\}T_1) = \{U/X\}(\mu Y\ T_1)$.

**15.2** Induction on $S$. Cases $B, 1, \mathrm{loc}, \top$ and $Y$, for $Y \neq X$, contradict the premise. Case $X$ is trivial. Cases $S_1 \times S_2$, $\updownarrow_{io} S_1$ and subsumption are by induction. Case $\mu Y\ S_1$. w.l.g. $Y \notin \mathrm{dom}(\Gamma) \cup \{X\}$, so $\{U/X\}(\mu Y\ S_1) = \mu Y\ (\{U/X\}S_1)$. By the typing rules $\Gamma, Y : K \vdash \{U/X\}S_1 : K$ and $Y$ guarded in $\{U/X\}S_1$. By induction there exists $K'$ such that $\Gamma, Y : K \vdash U : K'$ and $\Gamma, Y : K, X : K' \vdash S_1 : K$. By Lemma 11.1 $\Gamma \vdash U : K'$. By Lemmas 10 and 9.4 $\Gamma, X : K', Y : K \vdash S_1 : K$. By Lemma 4.4 $Y$ guarded in $S_1$. By the typing rules $\Gamma, X : K' \vdash \mu Y\ S_1 : K$.

**15.3** Induction on the pair of derivations of $\Gamma \vdash T : K_1$ and $\Gamma \vdash T : K_2$.

**15.4** Induction on $T \Downarrow S$. The only non-trivial case is

$$\frac{T \Downarrow S \quad S \neq X}{\mu X\ T \Downarrow \{\mu X\ T\ /\ X\}S}$$

Suppose $\Gamma \vdash \mu X\ T : K$. By the typing rules there exists $K'$ such that $X$ guarded in $T$, $\Gamma, X : K' \vdash T : K'$ and $K' \leqslant K$. By induction $\Gamma, X : K' \vdash S : K'$ and by the typing rules $\Gamma \vdash \mu X\ T : K'$. By Lemma 15.1 $\Gamma \vdash \{\mu X\ T/X\}S : K'$. By subsumption $\Gamma \vdash \{\mu X\ T/X\}S : K$.

Suppose $\Gamma \vdash \{\mu X\ T/X\}S : K$. w.l.g. $X \notin \mathrm{dom}(\Gamma)$.

Case $X \notin \mathrm{ftv}(S)$. We have $\Gamma \vdash S : K$, so by induction $\Gamma \vdash T : K$. By Lemma 11.1 $\Gamma, X : K \vdash T : K$ and $X \notin \mathrm{ftv}(T)$. By Lemma 4.1 $X$ guarded in $T$. By the typing rules $\Gamma \vdash \mu X\ T : K$.

Case $X \in \mathrm{ftv}(S)$. By Lemma 15.2 there exists $K'$ such that $\Gamma \vdash \mu X\ T : K'$ and $\Gamma, X : K' \vdash S : K$. By the type formation rules there exists $K''$ such that $\Gamma, X : K'' \vdash T : K''$ and $K'' \leqslant K'$. By induction $\Gamma, X : K' \vdash T : K$. By narrowing $\Gamma, X : K \wedge K'' \vdash T : K''$ and $\Gamma, X : K \wedge K'' \vdash T : K$. By Lemma 15.3 $\Gamma, X : K \wedge K'' \vdash T : K \wedge K''$. By the typing rules $\Gamma \vdash \mu X\ T : K \wedge K''$. By subsumption $\Gamma \vdash \mu X\ T : K$.

□

**PROOF (of Lemma 16)** Induction on $\Gamma \vdash v : T$.

Names: By context formation $\Gamma \vdash S : K$ for some kind $K$. By Lemma 15.4 $\Gamma \vdash T : K$. By subsumption $\Gamma \vdash T : \mathrm{Type}_{\_\_}$.

Base values and unit: By typing rules and subsumption.

Pairs: By induction and typing rules. □

## A.3 Subtyping

**PROOF (of Lemma 17)**

1. By induction on derivations one can show $S \dot{\leqslant}_1 T \Rightarrow S \dot{\leqslant}_2 T$.
2. We must check that for all guarded pre-types $S$ we have $S\ \dot{\mathbf{id}}\ S$. First, it is straightforward to check this for all cases of $S$ except $\mu X\ S$. Now for case $\mu X\ S$: by Lemma 3.2 there exists $T$ such that $\mu X\ S \Downarrow T$. Lemma 5.1 shows that $T$ is not of the form $\mu Y\ T'$ and by Lemma 4.7 $T\ \mathrm{guarded}$. By the first part $T\ \dot{\mathbf{id}}\ T$. Using the last two rules for $\dot{\_}$ shows that $\mu X\ S\ \dot{\mathbf{id}}\ \mu X\ S$.
3. Write $\overset{\circ}{\leqslant}$ for the least relation over guarded pre-types satisfying the rules of the definition of $\dot{\leqslant}$ except the two unfolding rules. We first show

   **Lemma 41** $R \overset{\circ}{\leqslant} S \wedge S \overset{\circ}{\leqslant} T \Rightarrow R(\leqslant;\leqslant)^{\cdot}T$.

   **PROOF** Consider $T$. Cases $B, 1, \mathrm{loc}, X$: here $R = S = T$ and $R\ \dot{\mathcal{R}}\ T$ for any $\mathcal{R}$.

   Case $\top$: $R\ \dot{\mathcal{R}}\ \top$ for any $\mathcal{R}$.

   Case $T_1 \times T_2$: then $S = S_1 \times S_2$, $S_1 \leqslant T_1$ and $S_2 \leqslant T_2$. Similarly $R = R_1 \times R_2$, $R_1 \leqslant S_1$ and $R_2 \leqslant S_2$. Hence $R_i(\leqslant;\leqslant)T_i$ so $R_1 \times R_2(\leqslant;\leqslant)^{\cdot}T_1 \times T_2$.

   Case $\updownarrow_{i''o''} T_1$: similar. □

   Secondly, suppose $R \leqslant S \wedge S \leqslant T$. By the premise $\leqslant \subseteq \dot{\leqslant}$ we have $R \dot{\leqslant} S \wedge S \dot{\leqslant} T$. One can show $R \dot{\leqslant} S \wedge S \dot{\leqslant} T \Rightarrow R(\leqslant;\leqslant)^{\cdot}T$ by considering cases of the derivations of the premises.

□

**PROOF (of Lemma 18)** First show $\leqslant \subseteq \dot{\leqslant}$: If $S \leqslant T$ then there exists $\mathcal{R} \subseteq \leqslant$ such that $S\ \mathcal{R}\ T \wedge \mathcal{R} \subseteq \dot{\mathcal{R}}$. By Lemma 41.1 $\dot{\mathcal{R}} \subseteq \dot{\leqslant}$ so by transitivity $\mathcal{R} \subseteq \dot{\leqslant}$ so $S \dot{\leqslant} T$. Now, by Lemma 41.1 again $\dot{\leqslant} \subseteq \dot{\leqslant}$ so $\dot{\leqslant} \subseteq \leqslant$. Reflexivity of $\leqslant$ is by Lemma 41.2, transitivity is by $\leqslant = \dot{\leqslant}$ and Lemma 41.3. □

**PROOF (of Lemma 19)** Suppose $T$ is not of the form $\mu X\ T_1$, then $T \Downarrow T$ so by Lemma 3.4 $T = S$ and by Lemma 18 $T \leqslant S \leqslant T$. Otherwise, suppose $T = \mu X\ T_1$. By Lemma 18 $S \dot{\leqslant} S$; by the $\leqslant$ rules $T \dot{\leqslant} S \dot{\leqslant} T$ and by Lemma 18 $T \leqslant S \leqslant T$. □

**PROOF (of Lemma 20)** Parts 1–3 are similar to the proof of Lemma 17. For part 4, one can show $S \ddot{\equiv} T \Rightarrow T\ (\equiv^{-1})^{\cdot\cdot}\ S$ by a routine induction. □

**Proof (of Lemma 21)** Similar to the proof of Lemma 18, except for symmetry. By $\equiv = \ddot{\equiv}$ and Lemma 20.4 we have $\equiv^{-1} \subseteq (\equiv^{-1})^{\cdot\cdot}$ so by definition of $\equiv$ we have $\equiv^{-1} \subseteq \equiv$. □

**Proof (of Proposition 22)**

> **Lemma 42** *If $R = R^{-1}$ then $\ddot{R} \subseteq \dot{R}$.*
>
> **Proof** Induction on $S \ddot{R} T$. □
>
> **Lemma 43** $\equiv \subseteq \leqslant$.
>
> **Proof** By Lemmas 18 and 42 $\ddot{\equiv} \subseteq \dot{\equiv}$. By Lemma 18 $\equiv \subseteq \dot{\equiv}$. By definition of $\leqslant$ we have $\equiv \subseteq \leqslant$. □
>
> **Lemma 44** $(\leqslant \cap \leqslant^{-1}) \subseteq (\leqslant \cap \leqslant^{-1})^{\cdot\cdot}$
>
> **Proof** Suppose $S \leqslant T \leqslant S$. By Lemma 18 $S \dot{\leqslant} T \dot{\leqslant} S$. One can show first that $S \overset{\circ}{\leqslant} T \overset{\circ}{\leqslant} S \Rightarrow S\ (\leqslant \cap \leqslant^{-1})^{\cdot\cdot}\ T$ by cases of $S \overset{\circ}{\leqslant} T$, and then that $S \dot{\leqslant} T \dot{\leqslant} S \Rightarrow S\ (\leqslant \cap \leqslant^{-1})^{\cdot\cdot}\ T$ by cases of $S \dot{\leqslant} T$ and $T \dot{\leqslant} S$. □

Immediate from the above. □

## A.4 Colocality

**Proof (of Lemma 23)** Induction on $\mathrm{colocal}(\Gamma, T, T)$.

Case $T = \updownarrow_{io} T_1$ and $\mathrm{colocal}(io, io)$: We have $i = \mathsf{L} \vee o = \mathsf{L}$, so the only applicable kinding rules for $T$ are kind subsumption and the last two for channels, which give only kinds greater then or equal to $\mathrm{Type}_{-\mathsf{E}}$.

Case $T = X$ and $\neg(\Gamma \vdash X : \mathrm{Type}_{\mathsf{G}-})$: Immediate.

Case $T \Downarrow T'$ and $\mathrm{colocal}(\Gamma, T', T')$: By induction $\neg(\Gamma \vdash T' : \mathrm{Type}_{\mathsf{G}-})$ By Lemma 15.4 $\neg(\Gamma \vdash T : \mathrm{Type}_{\mathsf{G}-})$.

Case $T = T_1 \times T_2$ and $\mathrm{colocal}(\Gamma, T_1, T_1)$ (the case for 2 is symmetrical): By induction $\neg(\Gamma \vdash T_1 : \mathrm{Type}_{\mathsf{G}-})$. The only applicable kinding rules for $T_1 \times T_2$ are kind subsumption and the product rule, which give only kinds greater than or equal to $\mathrm{Type}_{-\mathsf{E}}$. □

**Proof (of Lemma 24)** Induction on $\mathrm{colocal}(\Gamma, T_1, T_3)$.

Case $T_1 = \updownarrow_{io} S_1$, $T_3 = \updownarrow_{i''o''} S_3$ and $\mathrm{colocal}(io, i''o'')$: As $T_2 \leqslant \updownarrow_{i''o''} S_3$ by Lemma 18 $T_2 \dot{\leqslant} \updownarrow_{i''o''} S_3$ so by the $\leqslant$ rules there exist $i'o'$ and $S_2$ such that $T_2 \Downarrow \updownarrow_{i'o'} S_2$. By Lemma 15.4 $\Gamma \vdash \updownarrow_{i'o'} S_2 : \mathrm{Type}_{--}$. By Lemmas 19 and 18 $T_1 \leqslant \updownarrow_{i'o'} S_2 \leqslant T_3$, so $io \leqslant i'o' \leqslant i''o''$, so $\mathrm{colocal}(io, i'o')$ and $\mathrm{colocal}(i'o', i''o'')$. By the definition of $\mathrm{colocal}(\_,\_,\_)$ $\mathrm{colocal}(\Gamma, T_1, \updownarrow_{i'o'} S_2)$ and $\mathrm{colocal}(\Gamma, \updownarrow_{i'o'} S_2, T_3)$, then $\mathrm{colocal}(\Gamma, T_1, T_2)$ and $\mathrm{colocal}(\Gamma, T_2, T_3)$.

Case $T_1 = T_3 = X$ and $\neg(\Gamma \vdash X : \mathrm{Type}_{\mathsf{G}-})$: By Lemma 18 and the $\leqslant$ rules $T_2 \Downarrow X$. By the definition of $\mathrm{colocal}(\_,\_,\_)$ $\mathrm{colocal}(\Gamma, T_1, X)$ and $\mathrm{colocal}(\Gamma, X, T_3)$, then $\mathrm{colocal}(\Gamma, T_1, T_2)$ and $\mathrm{colocal}(\Gamma, T_2, T_3)$.

Case $T_1 \Downarrow T_1'$, $T_3 \Downarrow T_3'$ and $\mathrm{colocal}(\Gamma, T_1', T_3')$: By Lemma 6.3 there exists $T_2'$ such that $T_2 \Downarrow T_2'$. By Lemma 15.4 $\Gamma \vdash T_2' : \mathrm{Type}_{--}$. By Lemma 19 $T_1' \leqslant T_2' \leqslant T_3'$. By induction $\mathrm{colocal}(\Gamma, T_1', T_2')$ and $\mathrm{colocal}(\Gamma, T_2', T_3')$. By the definition of $\mathrm{colocal}(\_,\_,\_)$ $\mathrm{colocal}(\Gamma, T_1, T_2)$ and $\mathrm{colocal}(\Gamma, T_2, T_3)$.

Case $T_1 = R_1 \times S_1$, $T_3 = R_3 \times S_3$ and $\mathrm{colocal}(\Gamma, R_1, R_3)$ (the case for $S$ is symmetrical): By Lemma 18 and the $\leqslant$ rules there exist $R_2, S_2$ such that $T_2 \Downarrow R_2 \times S_2$. By Lemmas 19 and 18 $R_1 \times S_1 \leqslant R_2 \times S_2 \leqslant R_3 \times S_3$ so $R_1 \leqslant R_2 \leqslant R_3$. By Lemma 15.4 $\Gamma \vdash R_2 : \mathrm{Type}_{--}$. By induction $\mathrm{colocal}(\Gamma, R_1, R_2)$ and

$\mathrm{colocal}(\Gamma, R_2, R_3)$. By the definition of $\mathrm{colocal}(\_,\_,\_)$ and the fact that $R_1 \times S_1 \Downarrow R_1 \times S_1$ and $R_3 \times S_3 \Downarrow R_3 \times S_3$ we have $\mathrm{colocal}(\Gamma, T_1, T_2)$ and $\mathrm{colocal}(\Gamma, T_2, T_3)$. □

**PROOF (of Lemma 25)** By induction on $x \in \mathrm{colocaln}(\Gamma, v, S, T)$. The name case is immediate, the pair case is by induction and the $\mathrm{colocal}$ rule for pairs, the unfolding case is by induction and the $\mathrm{colocal}$ rule for unfolding, using the fact that $\Gamma \vdash v : S$ implies $S \Downarrow S$ (Lemma 7). □

**PROOF (of Lemma 26)** Induction on $x \in \mathrm{colocaln}(\Gamma, v, V, T)$.

Case $x = v$ and $\mathrm{colocal}(\Gamma, V, T)$: By Lemma 24.

Case $x \in \mathrm{colocaln}(\Gamma, v_1, V_1, T_1)$ (the case for 2 is symmetrical): We have $S \Downarrow S_1 \times S_2$ for $V_1 \leqslant S_1 \leqslant T_1$ and $V_2 \leqslant S_2 \leqslant T_2$. By induction $x \in \mathrm{colocaln}(\Gamma, v_1, V_1, S_1)$ so by definition of $\mathrm{colocaln}(\_,\_,\_,\_)$ $x \in \mathrm{colocaln}(\Gamma, \langle v_1, v_2 \rangle, V_1 \times V_2, S_1 \times S_2)$ and hence $x \in \mathrm{colocaln}(\Gamma, \langle v_1, v_2 \rangle, V_1 \times V_2, S)$.

Case $x \in \mathrm{colocaln}(\Gamma, v, V, T')$ and $T \Downarrow T'$: By Lemma 15.4 $\Gamma \vdash T' : \mathrm{Type}_{\_\_}$ and by Lemma 19 $V \leqslant S \leqslant T'$. By induction $x \in \mathrm{colocaln}(\Gamma, v, V, S)$. □

**PROOF (of Lemma 27)** Straightforward induction, using Lemma 24 and $S \leqslant S \leqslant T$ in the base case. □

**PROOF (of Lemma 28)**

**28.1** By 11.1 and 4.6.

**28.2** Induction, using 13.1.

**28.3** Induction, using 28.2.

□

**PROOF (of Lemma 29)**

**29.1** Induction on the definition of $\mathrm{colocal}$, using 9.4 and using part (b) of the definition of $\Delta \cong \Gamma$ in the type variable case.

**29.2** Induction on the definition of $\mathrm{colocaln}$, using 9.5, 9.4 and 29.1.

□

## A.5 Processes

**PROOF (of Lemma 30)** Induction on derivations, using Lemmas 7.2 and 6.1. □

**PROOF (of Lemma 31)** We show $\Gamma \cong \Delta$ implies $\Gamma \vdash P : \mathrm{process} \iff \Delta \vdash P : \mathrm{process}$, the result then follows by Lemma 14.

**OUT** By Lemma 9.5, part (IV) of the definition of $\Gamma \cong \Delta$ and Lemma 29.2.

**(REP-)IN** By Lemma 9.5, Lemma 9.3 and induction and part (IV) of the definition of $\Gamma \cong \Delta$.

**MIG** By Lemma 9.5 and induction.

**LET** By Lemma 9.5, Lemma 9.3 and induction, Lemma 29.2 and part (IV) of the definition of $\Gamma \cong \Delta$.

**NEW** By Lemma 9.4, Lemma 9.3 and induction.

**NIL** By part (I) of the definition of $\Gamma \cong \Delta$.

**PAR** By induction.

□

**PROOF (of Lemma 32)**

**OUT** By 30, 13.2, 13.3 and 28.3.

**(REP-)IN** By 30, 13.2, 13.3, 31 and induction.

**MIG** By 30, 13.2 and induction.

**LET** By 30, 13.2, 13.3, 28.3, 31 and induction.

**NEW** By 13.1, 31 and induction.

**NIL** Trivial.

**PAR** By induction.

□

## A.6 Relocation

**PROOF (of Lemma 33)**

**33.1** For part (i), if $\vdash \Gamma\ ok$ then $\vdash \Delta\ ok$ so using (b), (c) and 6.3 $\mathrm{dom}(\Gamma) = \mathrm{dom}(\Delta)$. The other parts are trivial.

**33.2** Suppose $\vdash \Gamma, \Theta$ ok. By 33.1 $\Delta \lesssim \Gamma$ so by 8.4 $\vdash \Delta, \Theta$ ok.

**33.3** We show

- If $\Delta \simeq \Gamma$ then $\Delta, X : K \simeq \Gamma, X : K$.
- If $\Delta \simeq \Gamma$ then $\Delta, x : @_l T \simeq \Gamma, x : @_l T$.

The result is then a trivial induction on $\Theta$. For the first, part (a) is by 33.2. Part (b) is by 33.2 and part (b) for $\Delta \simeq \Gamma$. Part (c) is by 33.2 and part (c) for $\Delta \simeq \Gamma$. For Part (d), suppose $\Gamma, X : K \vdash x : T \wedge \mathrm{colocal}((\Gamma, X : K), T, T)$. By 11.2, 16 and 11.1 $\Gamma \vdash x : T$ and $X \notin \mathrm{ftv}(T)$. By 28.1 $\mathrm{colocal}(\Gamma, T, T)$. By (d) $\Gamma \vdash x@l \iff \Delta \vdash x@l$. By 11.3 $\Gamma, X : K \vdash x@l \iff \Delta, X : K \vdash x@l$.

For the second, part (a) is by 33.2. Part (b) is by 33.2 and part (b) for $\Delta \simeq \Gamma$. Part (c) is by 33.2 and part (c) for $\Delta \simeq \Gamma$. For Part (d), suppose $\Gamma, x : @_l T \vdash x' : T' \wedge \mathrm{colocal}((\Gamma, x : @_l T), T', T')$. By 28.2 $\mathrm{colocal}(\Gamma, T', T')$. Now, either $x' = x$, in which case using Part (a) above $\Gamma, x : @_l T \vdash x'@k \iff k = l \iff \Delta, x : @_l T \vdash x'@k$, or $\Gamma \vdash x' : T'$. In this case by (d) $\Gamma \vdash x'@k \iff \Delta \vdash x'@k$ and by 13.3 $\Gamma, x : @_l T \vdash x'@k \iff \Delta, x : @_l T \vdash x'@k$.

**33.4** By 33.1 and 8.2.

**33.5** Induction on type derivations using (c).

**33.6** Induction on the definition of colocal, using 33.4 and using (b) in the type variable case.

**33.7** Induction on the definition of colocaln, using 33.5, 33.4 and 33.6.

**33.8** Induction on type derivations. We show the implication $\Gamma \vdash P : \mathrm{process} \Rightarrow \Delta \vdash P : \mathrm{process}$.

**Out**

$$\textsc{Out}\ \frac{\begin{array}{l}\Gamma \vdash l : \mathrm{loc}\\ \Gamma \vdash x : \updownarrow_{io} T\\ \Gamma \vdash v : T'\\ T' \leqslant T\\ o \leqslant \mathsf{L}\\ o = \mathsf{L} \Rightarrow \Gamma \vdash x@l\\ \forall a \in \mathrm{colocaln}(\Gamma, v, T', T)\ .\ \Gamma \vdash a@l\end{array}}{\Gamma \vdash @_l \overline{x} v : \mathrm{process}} \qquad \textsc{Out}\ \frac{\begin{array}{l}\Delta \vdash l : \mathrm{loc}\\ \Delta \vdash x : \updownarrow_{io} T\\ \Delta \vdash v : T'\\ T' \leqslant T\\ o \leqslant \mathsf{L}\\ o = \mathsf{L} \Rightarrow \Delta \vdash x@l\\ \forall a \in \mathrm{colocaln}(\Delta, v, T', T)\ .\ \Delta \vdash a@l\end{array}}{\Delta \vdash @_l \overline{x} v : \mathrm{process}}$$

The first three premises of the right hand Out are by 33.5. The next two are immediate. Suppose $o = \mathsf{L}$, then $\Gamma \vdash x@l$ and $\mathrm{colocal}(\Gamma, \updownarrow_{io} T, \updownarrow_{io} T)$ so $\Delta \vdash x@l$.

Suppose $a \in \mathrm{colocaln}(\Delta, v, T', T)$. By Lemma 33.7 $a \in \mathrm{colocaln}(\Gamma, v, T', T)$ so by the premise $\Gamma \vdash a@l$. By Lemma 27 there exists $S$ such that $\Gamma \vdash a : S$ and $\mathrm{colocal}(\Gamma, S, S)$. By (d) we have $\Delta \vdash a@l$.

**(Rep-)In**

$$(\textsc{Rep-})\textsc{In}\ \frac{\begin{array}{l}\Gamma \vdash l : \mathrm{loc}\\ \Gamma \vdash x : \updownarrow_{io} T\\ \Gamma, y : @_l T \vdash P : \mathrm{process}\\ i \leqslant \mathsf{L}\\ i = \mathsf{L} \Rightarrow \Gamma \vdash x@l\\ \mathrm{fl}(P) \subseteq \{l\}\end{array}}{\begin{array}{l}\Gamma \vdash @_l x(y).P : \mathrm{process}\\ \Gamma \vdash @_l\, !\, x(y).P : \mathrm{process}\end{array}} \qquad (\textsc{Rep-})\textsc{In}\ \frac{\begin{array}{l}\Delta \vdash l : \mathrm{loc}\\ \Delta \vdash x : \updownarrow_{io} T\\ \Delta, y : @_l T \vdash P : \mathrm{process}\\ i \leqslant \mathsf{L}\\ i = \mathsf{L} \Rightarrow \Delta \vdash x@l\\ \mathrm{fl}(P) \subseteq \{l\}\end{array}}{\begin{array}{l}\Delta \vdash @_l x(y).P : \mathrm{process}\\ \Delta \vdash @_l\, !\, x(y).P : \mathrm{process}\end{array}}$$

The first two premises are by (c). By 33.3 we have $\Gamma, y : @_l T \simeq \Delta, y : @_l T$ so by induction $\Delta, y : @_l T \vdash P : \mathrm{process}$. Suppose $i = \mathsf{L}$, then $\Gamma \vdash x@l$ and $\mathrm{colocal}(\Gamma, \updownarrow_{io} T, \updownarrow_{io} T)$ so $\Delta \vdash x@l$.

**Mig**

$$\textsc{Mig}\ \frac{\begin{array}{l}\Gamma \vdash l : \mathrm{loc}\\ \Gamma \vdash v : \mathrm{loc}\\ \Gamma \vdash P : \mathrm{process}\\ \mathrm{fl}(P) \subseteq \{l\}\end{array}}{\Gamma \vdash @_l \mathbf{migrate\_to}\ v\ \mathbf{then}\ P : \mathrm{process}} \qquad \textsc{Mig}\ \frac{\begin{array}{l}\Delta \vdash l : \mathrm{loc}\\ \Delta \vdash v : \mathrm{loc}\\ \Delta \vdash P : \mathrm{process}\\ \mathrm{fl}(P) \subseteq \{l\}\end{array}}{\Delta \vdash @_l \mathbf{migrate\_to}\ v\ \mathbf{then}\ P : \mathrm{process}}$$

Straightforward, by 33.5 and induction.

**Let**

$$\textsc{Let}\ \frac{\begin{array}{l}\Gamma \vdash l : \mathrm{loc}\\ \Gamma \vdash v : T'\\ \Gamma, y_1 : @_l T_1, y_2 : @_l T_2 \vdash P : \mathrm{process}\\ T' \leqslant T_1 \times T_2\\ \forall a \in \mathrm{colocaln}(\Gamma, v, T', T_1 \times T_2)\ .\ \Gamma \vdash a@l\\ \mathrm{fl}(P) \subseteq \{l\}\end{array}}{\Gamma \vdash @_l \mathbf{let}\ \langle y_1 : T_1, y_2 : T_2\rangle = v\ \mathbf{in}\ P : \mathrm{process}} \qquad \textsc{Let}\ \frac{\begin{array}{l}\Delta \vdash l : \mathrm{loc}\\ \Delta \vdash v : T'\\ \Delta, y_1 : @_l T_1, y_2 : @_l T_2 \vdash P : \mathrm{process}\\ T' \leqslant T_1 \times T_2\\ \forall a \in \mathrm{colocaln}(\Delta, v, T', T_1 \times T_2)\ .\ \Delta \vdash a@l\\ \mathrm{fl}(P) \subseteq \{l\}\end{array}}{\Delta \vdash @_l \mathbf{let}\ \langle y_1 : T_1, y_2 : T_2\rangle = v\ \mathbf{in}\ P : \mathrm{process}}$$

The first two premises are by 33.5. By 33.3 we have $\Gamma, y_1 : @_l T_1, y_1 : @_l T_1 \simeq \Delta, y_1 : @_l T_1, y_2 : @_l T_2$ so by induction $\Delta, y_1 : @_l T_1, y_2 : @_l T_2 \vdash P : \mathrm{process}$.

Suppose $a \in \mathrm{colocaln}(\Delta, v, T', T_1 \times T_2)$. By Lemma 33.7 $a \in \mathrm{colocaln}(\Gamma, v, T', T_1 \times T_2)$ so by the premise $\Gamma \vdash a@l$. By Lemma 27 there exists $S$ such that $\Gamma \vdash a : S$ and $\mathrm{colocal}(\Gamma, S, S)$. By (d) we have $\Delta \vdash a@l$.

**New**

$$\textsc{New}\ \frac{\Gamma \vdash T : \mathrm{Type}_{-\mathsf{E}} \quad \Gamma, x : @_l T \vdash P : \mathrm{process}}{\Gamma \vdash (\mathbf{new}\ x : @_l T) P : \mathrm{process}} \qquad \textsc{New}\ \frac{\Delta \vdash T : \mathrm{Type}_{-\mathsf{E}} \quad \Delta, x : @_l T \vdash P : \mathrm{process}}{\Delta \vdash (\mathbf{new}\ x : @_l T) P : \mathrm{process}}$$

Straightforward by 33.4, 33.3 and induction.

**Nil**

$$\textsc{Nil}\ \frac{\vdash \Gamma\ ok}{\Gamma \vdash 0 : \mathrm{process}} \qquad \textsc{Nil}\ \frac{\vdash \Delta\ ok}{\Delta \vdash 0 : \mathrm{process}}$$

Straightforward by (a).

**Par**

$$\textsc{Par}\ \frac{\Gamma \vdash P : \mathrm{process} \quad \Gamma \vdash Q : \mathrm{process}}{\Gamma \vdash P \mid Q : \mathrm{process}} \qquad \textsc{Par}\ \frac{\Delta \vdash P : \mathrm{process} \quad \Delta \vdash Q : \mathrm{process}}{\Delta \vdash P \mid Q : \mathrm{process}}$$

Straightforward by induction.

□

## A.7 Narrowing

**Proof (of Lemma 34)**

**34.1** For part (i), if $\vdash \Gamma\ ok$ then $\vdash \Delta\ ok$ so using (2) and (3) $\mathrm{dom}(\Gamma) = \mathrm{dom}(\Delta)$. Parts (ii) and (iii) are by (1) and (2). For part (iv) if $\Gamma \vdash x : \mathrm{loc}$ then by (1) there exists $S$ such that $\Delta \vdash x : S$ so by (4) $S = \mathrm{loc}$.

**34.2** Suppose $\vdash \Gamma, \Theta$ ok. By 34.1 $\Delta \lesssim \Gamma$ so by 8.4 $\vdash \Delta, \Theta$ ok.

**34.3** We show

- If $\Delta \leqslant \Gamma$ then $\Delta, X : K \leqslant \Gamma, X : K$.
- If $\Delta \leqslant \Gamma$ then $\Delta, x : @_l T \leqslant \Gamma, x : @_l T$.

The result is then a trivial induction on $\Theta$. For the first, part (1) is by 34.2. Part (2) is by 34.2 and part (2) for $\Delta \leqslant \Gamma$. Part (3) is by 34.2 and part (3) for $\Delta \leqslant \Gamma$. For Part (4), suppose $\Delta, X : K \vdash x : T \wedge \Gamma, X : K \vdash x : S$.

By 11.2 $\Delta \vdash x : T \wedge \Gamma \vdash x : S$ so by part (3) for $\Delta \leqslant \Gamma$ we have $T \leqslant S$.

For the second, part (1) is by 34.2. Part (2) is by 34.2 and part (2) for $\Delta \leqslant \Gamma$. Part (3) is by 34.2 and part (3) for $\Delta \leqslant \Gamma$. For Part (4) suppose $\Delta, x : @_l T \vdash x' : T' \wedge \Gamma, x : @_l T \vdash x' : S'$.

Case $x' = x$: by typing rules $T' = S'$ so $T' \leqslant S'$. Case $x' \neq x$: By 13.2 $\Delta \vdash x' : T' \wedge \Gamma \vdash x' : S'$ so by part (3) for $\Delta \leqslant \Gamma$ we have $T' \leqslant S'$.

**34.4** By Lemma 34.1 and 8.2.

**34.5** Induction on $\Gamma \vdash v : S$. Names are by the definition of $\Delta \leqslant \Gamma$, base values and unit by (1) and the reflexivity of subtyping, pairs by induction and the subtyping rules for $\times$.

**34.6** Induction on $\mathrm{colocal}$, using 34.4 and (2) in the type variable case.

**34.7** Consider $x$ in the left hand side of the conclusion.

Case $x = v \wedge \mathrm{colocal}(\Delta, T', S)$. By Lemma 34.6 $\mathrm{colocal}(\Gamma, T', S)$. By Lemma 24 $\mathrm{colocal}(\Gamma, S', S)$. By definition of colocaln $x$ is an element of the right hand side.

Case $x \in \mathrm{colocaln}(\Delta, v_1, T_1', S_1)$. We have $S' = S_1' \times S_2'$. By induction $x \in \mathrm{colocaln}(\Gamma, v_1, S_1', S_1)$. By definition of colocaln $x \in \mathrm{colocaln}(\Gamma, \langle v_1, v_2\rangle, S_1' \times S_2', S_1 \times S_2)$ so $x$ is an element of the right hand side.

Case $x \in \mathrm{colocaln}(\Delta, v_1, T', S_0)$ and $S \Downarrow S_0$. By Lemma 19 we have $T' \leqslant S' \leqslant S_0$. By induction $x \in \mathrm{colocaln}(\Gamma, v, S', S_0)$. By definition of colocaln $x \in \mathrm{colocaln}(\Gamma, v, S', S)$.

**34.8** By induction on type derivations.

**Out**

$$\textsc{Out}\ \dfrac{\begin{array}{l}\Gamma \vdash l : \mathrm{loc}\\ \Gamma \vdash x : \updownarrow_{io} S\\ \Gamma \vdash v : S'\\ S' \leqslant S\\ o \leqslant \mathsf{L}\\ o = \mathsf{L} \Rightarrow \Gamma \vdash x@l\\ \forall a \in \mathrm{colocaln}(\Gamma, v, S', S)\ .\ \Gamma \vdash a@l\end{array}}{\Gamma \vdash @_l \overline{x} v : \mathrm{process}} \qquad \textsc{Out}\ \dfrac{\begin{array}{l}\Delta \vdash l : \mathrm{loc}\\ \Delta \vdash x : \updownarrow_{i'o'} T\\ \Delta \vdash v : T'\\ T' \leqslant T\\ o' \leqslant \mathsf{L}\\ o' = \mathsf{L} \Rightarrow \Delta \vdash x@l\\ \forall a \in \mathrm{colocaln}(\Delta, v, T', T)\ .\ \Delta \vdash a@l\end{array}}{\Delta \vdash @_l \overline{x} v : \mathrm{process}}$$

By Lemma 34.5 there exist $L$, $\hat{T}$ and $T'$ such that

$$\begin{array}{ll}\Delta \vdash l : L & L \leqslant \mathrm{loc}\\ \Delta \vdash x : \hat{T} & \hat{T} \leqslant \updownarrow_{io} S\\ \Delta \vdash v : T' & T' \leqslant S'\end{array}$$

By Lemma 18 $L \mathrel{\dot{\leqslant}} \mathrm{loc}$ and $\hat{T} \mathrel{\dot{\leqslant}} \updownarrow_{io} S$. By Lemma 7.3 $L \Downarrow L$ and $\hat{T} \Downarrow \hat{T}$ so by Lemma 5.1 $L$ and $\hat{T}$ are not of the form $\mu X\_$. By the $\leqslant$ rules $L = \mathrm{loc}$ and there exist $i'o'$ and $T$ such that $\hat{T} = \updownarrow_{i'o'} T$. By the $\leqslant$ rules as $o \leqslant \mathsf{L}$ we have $S \leqslant T$. By transitivity as have $T' \leqslant S' \leqslant S \leqslant T$ we have $T' \leqslant T$. By the $\leqslant$ rules $i'o' \leqslant io$ so $o' \leqslant \mathsf{L}$.

Suppose $o' = \mathsf{L}$. Then $o = \mathsf{L}$ so $\Gamma \vdash x@l$. By $\Delta \leqslant \Gamma$ we have $\Delta \vdash x@l$.

Suppose $a \in \mathrm{colocaln}(\Delta, v, T', T)$. By Lemma 26 $a \in \mathrm{colocaln}(\Delta, v, T', S)$. By Lemma 34.7 $a \in \mathrm{colocaln}(\Gamma, v, S', S)$, so $\Gamma \vdash a@l$, so by definition of $\Delta \leqslant \Gamma$ we have $\Delta \vdash a@l$.

**(Rep-)In**

$$(\textsc{Rep-})\textsc{In}\ \dfrac{\begin{array}{l}\Gamma \vdash l : \mathrm{loc}\\ \Gamma \vdash x : \updownarrow_{io} S\\ \Gamma, y : @_l S \vdash P : \mathrm{process}\\ i \leqslant \mathsf{L}\\ i = \mathsf{L} \Rightarrow \Gamma \vdash x@l\\ \mathrm{fl}(P) \subseteq \{l\}\end{array}}{\begin{array}{l}\Gamma \vdash @_l x(y).P : \mathrm{process}\\ \Gamma \vdash @_l\, !x(y).P : \mathrm{process}\end{array}} \qquad (\textsc{Rep-})\textsc{In}\ \dfrac{\begin{array}{l}\Delta \vdash l : \mathrm{loc}\\ \Delta \vdash x : \updownarrow_{i'o'} T\\ \Delta, y : @_l T \vdash P : \mathrm{process}\\ i' \leqslant \mathsf{L}\\ i' = \mathsf{L} \Rightarrow \Delta \vdash x@l\\ \mathrm{fl}(P) \subseteq \{l\}\end{array}}{\begin{array}{l}\Delta \vdash @_l x(y).P : \mathrm{process}\\ \Delta \vdash @_l\, !x(y).P : \mathrm{process}\end{array}}$$

As in Out there exist $i'o'$ and $T$ such that

$$\begin{array}{l}\Delta \vdash l : \mathrm{loc}\\ \Delta \vdash x : \updownarrow_{i'o'} T\end{array}$$

and $\updownarrow_{i'o'} T \leqslant \updownarrow_{io} S$. By the $\leqslant$ rules as $i \leqslant \mathsf{L}$ we have $T \leqslant S$.

By Lemma 34.3 $\Delta, y:@_l T \leqslant \Gamma, y:@_l T$. Using Lemma 34.4 $\Gamma, y:@_l T \leqslant \Gamma, y:@_l S$ so by transitivity $\Delta, y:@_l T \leqslant \Gamma, y:@_l S$.

By induction $\Delta, y:@_l T \vdash P:\mathrm{process}$.

By the $\leqslant$ rules $i'o' \leqslant io$ so $i' \leqslant \mathsf{L}$.

Suppose $i' = \mathsf{L}$. Then $i = \mathsf{L}$ so $\Gamma \vdash x@l$. By $\Delta \leqslant \Gamma$ we have $\Delta \vdash x@l$.

**MIG**

$$\textsc{Mig}\ \frac{\begin{array}{c}\Gamma \vdash l:\mathrm{loc}\\ \Gamma \vdash v:\mathrm{loc}\\ \Gamma \vdash P:\mathrm{process}\\ \mathrm{fl}(P) \subseteq \{l\}\end{array}}{\Gamma \vdash @_l\mathbf{migrate\_to}\ v\ \mathbf{then}\ P:\mathrm{process}} \qquad \textsc{Mig}\ \frac{\begin{array}{c}\Delta \vdash l:\mathrm{loc}\\ \Delta \vdash v:\mathrm{loc}\\ \Delta \vdash P:\mathrm{process}\\ \mathrm{fl}(P) \subseteq \{l\}\end{array}}{\Delta \vdash @_l\mathbf{migrate\_to}\ v\ \mathbf{then}\ P:\mathrm{process}}$$

Similar reasoning for $l$ and $v$ as in the (REP)-IN case, and induction.

**LET**

$$\textsc{Let}\ \frac{\begin{array}{l}\Gamma \vdash l:\mathrm{loc}\\ \Gamma \vdash v:S'\\ \Gamma, y_1:@_l T_1, y_2:@_l T_2 \vdash P:\mathrm{process}\\ S' \leqslant T_1 \times T_2\\ \forall a \in \mathrm{colocaln}(\Gamma, v, S', T_1 \times T_2)\ .\ \Gamma \vdash a@l\\ \mathrm{fl}(P) \subseteq \{l\}\end{array}}{\Gamma \vdash @_l\mathbf{let}\ \langle y_1:T_1, y_2:T_2\rangle = v\ \mathbf{in}\ P:\mathrm{process}} \qquad \textsc{Let}\ \frac{\begin{array}{l}\Delta \vdash l:\mathrm{loc}\\ \Delta \vdash v:T'\\ \Delta, y_1:@_l T_1, y_2:@_l T_2 \vdash P:\mathrm{process}\\ T' \leqslant T_1 \times T_2\\ \forall a \in \mathrm{colocaln}(\Delta, v, T', T_1 \times T_2)\ .\ \Delta \vdash a@l\\ \mathrm{fl}(P) \subseteq \{l\}\end{array}}{\Delta \vdash @_l\mathbf{let}\ \langle y_1:T_1, y_2:T_2\rangle = v\ \mathbf{in}\ P:\mathrm{process}}$$

As in OUT there exists $T'$ such that $T' \leqslant S'$ and

$$\begin{array}{l}\Delta \vdash l:\mathrm{loc}\\ \Delta \vdash v:T'\end{array}$$

By Lemma 34.3 $\Delta, y_1:@_l T_1, y_2:@_l T_2 \leqslant \Gamma, y_1:@_l T_1, y_2:@_l T_2$.

By induction $\Delta, y_1:@_l T_1, y_2:@_l T_2 \vdash P:\mathrm{process}$.

By transitivity $T' \leqslant T_1 \times T_2$.

Suppose $a \in \mathrm{colocaln}(\Delta, v, T', T_1 \times T_2)$. By Lemma 34.7 $a \in \mathrm{colocaln}(\Gamma, v, S', T_1 \times T_2)$, so $\Gamma \vdash a@l$, so by definition of $\Delta \leqslant \Gamma$ we have $\Delta \vdash a@l$.

**NEW**

$$\textsc{New}\ \frac{\begin{array}{c}\Gamma \vdash S:\mathrm{Type}_{-\mathsf{E}}\\ \Gamma, x:@_l S \vdash P:\mathrm{process}\end{array}}{\Gamma \vdash (\mathbf{new}\ x:@_l S)P:\mathrm{process}} \qquad \textsc{New}\ \frac{\begin{array}{c}\Delta \vdash S:\mathrm{Type}_{-\mathsf{E}}\\ \Delta, x:@_l S \vdash P:\mathrm{process}\end{array}}{\Delta \vdash (\mathbf{new}\ x:@_l S)P:\mathrm{process}}$$

By Lemma 34.4 $\Delta \vdash S:\mathrm{Type}_{-\mathsf{E}}$.

By Lemma 34.3 $\Delta, x:@_l S \leqslant \Gamma, x:@_l S$.

By induction $\Delta, x:@_l S \vdash P:\mathrm{process}$.

**NIL**

$$\textsc{Nil}\ \frac{\vdash \Gamma\ ok}{\Gamma \vdash 0:\mathrm{process}} \qquad \textsc{Nil}\ \frac{\vdash \Delta\ ok}{\Delta \vdash 0:\mathrm{process}}$$

By part (1) of $\Delta \leqslant \Gamma$.

**Par**

$$\textsc{Par}\ \frac{\Gamma \vdash P : \text{process} \quad \Gamma \vdash Q : \text{process}}{\Gamma \vdash P \mid Q : \text{process}} \qquad \textsc{Par}\ \frac{\Delta \vdash P : \text{process} \quad \Delta \vdash Q : \text{process}}{\Delta \vdash P \mid Q : \text{process}}$$

By induction.

□

## A.8 Substitution

**Proof (of Lemma 35)** Induction on $P$. □

**Proof (of Lemma 36)** Induction on $\Gamma, z : @_j V \vdash w : S$.

Case $w = z$. We have $S = V$ so take $T \overset{def}{=} U$.

Case $w = x \neq z$. Use reflexivity of $\leqslant$.

Cases $b, \langle\rangle$. Use reflexivity of $\leqslant$.

Case $\langle w_1, w_2 \rangle$. By the typing rules $S = S_1 \times S_2$ and for $i \in \{1, 2\}$ we have $\Gamma, z : @_j V \vdash w_i : S_i$. By induction there exist $T_1, T_2$ such that for $i \in \{1, 2\}$ we have $\Gamma \vdash \{u/z\}w_i : T_i \wedge T_i \leqslant S_i$. By the typing and subtyping rules $\Gamma \vdash \{u/z\}\langle w_1, w_2 \rangle : T_1 \times T_2 \wedge T_1 \times T_2 \leqslant S_1 \times S_2$. □

**Proof (of Lemma 37)** By Lemmas 36 and 7.1 $T' \leqslant S'$.

Suppose $a$ is an element of the left hand side.

- Base case $a = \{u/z\}v \wedge \text{colocal}(\Gamma, T', S)$:
  - subcase $a = v \neq z$: We have $S' = T'$ so $a \in \text{colocaln}(\Gamma, v, S', S)$ so by Lemma 28.3 $a \in \text{colocaln}((\Gamma, z : @_j V), v, S', S)$.
  - subcase $a = u \wedge v = z$: By value typing rules $V \Downarrow S'$ and by Lemma 7.1 $U = T'$. By Lemma 19 $T' \leqslant S'$ so we have

    $$\begin{array}{ccccc} U & \leqslant & V & & \\ \| & & \Leftarrow & & \\ T' & \leqslant & S' & \leqslant & S \end{array}$$

    By Lemmas 16 and 13.1 for $S'$, Lemma 13.1 for $S$, Lemma 16 for $U$ and $T'$ and the premise for $V$ each of these has kind $\text{Type}_{\_\_}$ in $\Gamma$. We have $U \leqslant S' \leqslant S$ and (by Lemma 19) $U \leqslant V \leqslant S$ so by Lemma 24 $\text{colocal}(\Gamma, U, V)$ and $\text{colocal}(\Gamma, S', S)$. By Lemma 28.2 $\text{colocal}((\Gamma, z : @_j V), S', S)$. By the definition of colocaln we have $a \in \text{colocaln}(\Gamma, u, U, V)$ and $z \in \text{colocaln}((\Gamma, z : @_j V), v, S', S)$.

- Inductive case

$$\begin{array}{rcl} \{u/z\}v & = & \langle w_1, w_2 \rangle \\ T' & = & T'_1 \times T'_2 \\ S & = & S_1 \times S_2 \\ a & \in & \text{colocaln}(\Gamma, w_1, T'_1, S_1) \end{array}$$

- subcase $v = \langle v_1, v_2\rangle \wedge \{u/z\}v_1 = w_1 \wedge \{u/z\}v_2 = w_2$: By the typing rules for pairs there exist $S_1'$ and $S_2'$ such that $S' = S_1' \times S_2'$ and $\Gamma, z:@_j V \vdash v_1 : S_1'$. By the $\leqslant$ rules $S_1' \leqslant S_1$. By induction

$$a \in \{\mathrm{colocaln}(\Gamma, u, U, V)/z\}\mathrm{colocaln}((\Gamma, z:@_j V), v_1, S_1', S_1).$$

  By the definition of colocaln and monotonicity of $\{\_/\_\}\_$ we have

$$a \in \{\mathrm{colocaln}(\Gamma, u, U, V)/z\}\mathrm{colocaln}((\Gamma, z:@_j V), \langle v_1, v_2\rangle, S_1' \times S_2', S_1 \times S_2).$$

- subcase $v = z \wedge u = \langle w_1, w_2\rangle$: As in the subcase $a = u \wedge v = z$ we have

$$\begin{array}{ccccc} U & \leqslant & V & & \\ \| & & \Leftarrow & & \\ T' & \leqslant & S' & \leqslant & S \end{array}$$

  By definition of colocaln we have $a \in \mathrm{colocaln}(\Gamma, \langle w_1, w_2\rangle, T_1' \times T_2', S_1 \times S_2)$ so $a \in \mathrm{colocaln}(\Gamma, u, U, S)$. By Lemma 26 $a \in \mathrm{colocaln}(\Gamma, u, U, S')$. By definition of colocaln $a \in \mathrm{colocaln}(\Gamma, u, U, V)$. Now, by Lemma 25 $\mathrm{colocal}(\Gamma, T', S)$. By Lemma 24 $\mathrm{colocal}(\Gamma, S', S)$. By the definition of colocaln we have $z \in \mathrm{colocaln}((\Gamma, z:@_j V), v, S', S)$.

- Inductive case $a \in \mathrm{colocaln}(\Gamma, \{u/z\}v, T', S_0)$ and $S \Downarrow S_0$: By Lemma 15.4 $\Gamma, z:@_j V \vdash S_0 : \mathrm{Type}_{-\,-}$. By Lemma 19 $S' \leqslant S_0$. By induction $a \in \{\mathrm{colocaln}(\Gamma, u, U, V)/z\}\mathrm{colocaln}((\Gamma, z:@_j V), v, S', S_0)$. By the definition of colocaln and monotonicity of $\{\_/\_\}\_$ we have

$$a \in \{\mathrm{colocaln}(\Gamma, u, U, V)/z\}\mathrm{colocaln}((\Gamma, z:@_j V), v, S', S).$$

□

**PROOF (of Lemma 38)** By induction on type derivations.

**OUT** To show this instance of the substitution lemma we suppose we have its premises, and so the upper instance of OUT, and show the lower instance of OUT.

$$\dfrac{\begin{array}{l}\Gamma, z:@_j V \vdash @_l\overline{x}v : \mathrm{process}\\ \Gamma \vdash u : U\\ U \leqslant V\\ \forall a \in \mathrm{colocaln}(\Gamma, u, U, V)\ .\ \Gamma \vdash a@j\\ z \notin \mathrm{fl}(@_l\overline{x}v)\end{array}}{\Gamma \vdash \{u/z\}@_l\overline{x}v : \mathrm{process}}\ \textsc{Out}\ \dfrac{\begin{array}{l}\Gamma, z:@_j V \vdash l : \mathrm{loc}\\ \Gamma, z:@_j V \vdash x : \updownarrow_{io} S\\ \Gamma, z:@_j V \vdash v : S'\\ S' \leqslant S\\ o \leqslant \mathsf{L}\\ o = \mathsf{L} \Rightarrow \Gamma, z:@_j V \vdash x@l\\ \forall a \in \mathrm{colocaln}(\Gamma, z:@_j V, v, S', S)\ .\ \Gamma, z:@_j V \vdash a@l\end{array}}{\Gamma, z:@_j V \vdash @_l\overline{x}v : \mathrm{process}}$$

$$\textsc{Out}\ \dfrac{\begin{array}{l}\Gamma \vdash \{u/z\}l : \mathrm{loc}\\ \Gamma \vdash \{u/z\}x : \updownarrow_{i'o'} T\\ \Gamma \vdash \{u/z\}v : T'\\ T' \leqslant T\\ o' \leqslant \mathsf{L}\\ o' = \mathsf{L} \Rightarrow \Gamma \vdash \{u/z\}x@\{u/z\}l\\ \forall a \in \mathrm{colocaln}(\Gamma, \{u/z\}v, T', T)\ .\ \Gamma \vdash a@\{u/z\}l\end{array}}{\Gamma \vdash @_{\{u/z\}l}\overline{\{u/z\}x}\{u/z\}v : \mathrm{process}}$$

By $z \notin \mathrm{fl}(@_l\overline{x}v)$ have $z \neq l$ so $\{u/z\}l = l$. By Lemma 13.2 $\Gamma \vdash \{u/z\}l : \mathrm{loc}$.

By Lemma 36 there is a type $\hat{T}$ such that $\Gamma \vdash \{u/z\}x : \hat{T} \wedge \hat{T} \leqslant \updownarrow_{io} S$.

By Lemma 7.3 $\hat{T} \Downarrow \hat{T}$ so by Lemma 5.1 $\hat{T}$ is not of the form $\mu X\ \_$ and by the subtyping rules there are $i'o'$ and $T$ such that $\hat{T} = \updownarrow_{i'o'} T$. By the subtyping rules $i'o' \leqslant io$. Also, as $o \leqslant \mathsf{L}$ we have $\mathrm{contravariant}(io) \vee \mathrm{nonvariant}(io)$ so $S \leqslant T$.

By Lemma 36 there is a type $T'$ such that $\Gamma \vdash \{u/z\}v : T' \wedge T' \leqslant S'$.

By transitivity $T' \leqslant T$.

By transitivity $o' \leqslant \mathsf{L}$.

Now suppose $o' = \mathsf{L}$. It follows that $o = \mathsf{L}$ so $\Gamma, z : @_j V \vdash x@l$.

> Case $z \neq x$: By Lemma 13.3 $\Gamma \vdash x@l$ so $\Gamma \vdash (\{u/z\}x)@(\{u/z\}l)$ .
>
> Case $z = x$: By value typing and location rules $V \Downarrow \updownarrow_{io} S$ and $j = l$.
>
> By Lemma 7.1 $U = \hat{T}$ so $U = \updownarrow_{i'o'} T$.
>
> By Lemmas 16 and 13.1 we have $\Gamma \vdash \updownarrow_{i'o'} T : \mathrm{Type}_{--}$, $\Gamma \vdash \updownarrow_{io} S : \mathrm{Type}_{--}$ and $\updownarrow_{i'o'} T \leqslant \updownarrow_{io} S$. As $\mathrm{colocal}(io, i'o')$ we have $\mathrm{colocal}(\Gamma, \updownarrow_{i'o'} T, \updownarrow_{io} S)$.
>
> By the value typing rules $u$ is a name, so $u \in \mathrm{colocaln}(\Gamma, u, \updownarrow_{i'o'} T, \updownarrow_{io} S)$.
>
> By the definition of $\mathrm{colocaln}$ $u \in \mathrm{colocaln}(\Gamma, u, \updownarrow_{i'o'} T, V) = \mathrm{colocaln}(\Gamma, u, U, V)$.
>
> By the premise of the substitution lemma $\Gamma \vdash u@j$ so $\Gamma \vdash u@l$.

Suppose $a \in \mathrm{colocaln}(\Gamma, \{u/z\}v, T', T)$.

By Lemma 26 $a \in \mathrm{colocaln}(\Gamma, \{u/z\}v, T', S)$.

By Lemma 37 one of the following hold.

1. $a \neq z$ and $a \in \mathrm{colocaln}((\Gamma, z : @_j V), v, S', S)$
2. $a \in \mathrm{colocaln}(\Gamma, u, U, V)$ and $z \in \mathrm{colocaln}((\Gamma, z : @_j V), v, S', S)$

In the first case by the premise of the upper instance of OUT we have $\Gamma, z : @_j V \vdash a@l$ so by Lemma 13.3 $\Gamma \vdash a@l$.

In the second case by the premise of the substitution lemma we have $\Gamma \vdash a@j$. Moreover, by the premise of the upper instance of OUT we have $\Gamma, z : @_j V \vdash z@l$ so $j = l$ so $\Gamma \vdash a@l$.

**(REP-)IN** To show the instance of the substitution lemma on the left we suppose we have its premises, and hence the left instance of (REP-)IN. We show, using the right instance of the substitution lemma, that we have the right instance of (REP-)IN.

$$\frac{\begin{array}{l}\Gamma, z : @_j V \vdash @_l x(y).P : \mathrm{process}\\ \Gamma \vdash u : U\\ U \leqslant V\\ \forall a \in \mathrm{colocaln}(\Gamma, u, U, V)\ .\ \Gamma \vdash a@j\\ z \notin \mathrm{fl}(@_l x(y).P)\end{array}}{\Gamma \vdash \{u/z\}@_l x(y).P : \mathrm{process}} \qquad \frac{\begin{array}{l}\Gamma, y : @_l S, z : @_j V \vdash P : \mathrm{process}\\ \Gamma, y : @_l S \vdash u : U\\ U \leqslant V\\ \forall a \in \mathrm{colocaln}(\Gamma, y : @_l S, u, U, V)\ .\ \Gamma, y : @_l S \vdash a@j\\ z \notin \mathrm{fl}(P)\end{array}}{\Gamma, y : @_l S \vdash \{u/z\}P : \mathrm{process}}$$

$$\text{(REP-)IN}\ \frac{\begin{array}{l}\Gamma, z : @_j V \vdash l : \mathrm{loc}\\ \Gamma, z : @_j V \vdash x : \updownarrow_{io} S\\ \Gamma, z : @_j V, y : @_l S \vdash P : \mathrm{process}\\ i \leqslant \mathsf{L}\\ i = \mathsf{L} \Rightarrow \Gamma, z : @_j V \vdash x@l\\ \mathrm{fl}(P) \subseteq \{l\}\end{array}}{\begin{array}{l}\Gamma, z : @_j V \vdash @_l x(y).P : \mathrm{process}\\ \Gamma, z : @_j V \vdash @_l\, !\, x(y).P : \mathrm{process}\end{array}} \quad \text{(REP-)IN}\ \frac{\begin{array}{l}\Gamma \vdash \{u/z\}l : \mathrm{loc}\\ \Gamma \vdash \{u/z\}x : \updownarrow_{i'o'} T\\ \Gamma, y : @_{\{u/z\}l} T \vdash \{u/z\}P : \mathrm{process}\\ i' \leqslant \mathsf{L}\\ i' = \mathsf{L} \Rightarrow \Gamma \vdash \{u/z\}x@\{u/z\}l\\ \mathrm{fl}(\{u/z\}P) \subseteq \{\{u/z\}l\}\end{array}}{\begin{array}{l}\Gamma \vdash @_{\{u/z\}l} \{u/z\}x(y).\{u/z\}P : \mathrm{process}\\ \Gamma \vdash @_{\{u/z\}l}\, !\, \{u/z\}x(y).\{u/z\}P : \mathrm{process}\end{array}}$$

Premises of right hand substitution lemma:

By Lemma 30 $\vdash \Gamma, z : @_j V, y : @_l S\ \mathit{ok}$.

By context formation $z \neq y$, $j \neq y$ and by $z \notin \mathrm{fl}(@_l ...)$ we have $z \neq l$.

By Lemma 31 $\Gamma, y : @_l S, z : @_j V \vdash P : \mathrm{process}$.

By Lemma 13.2 $\Gamma, y : @_l S \vdash u : U$.

By Lemma 28.3 $\mathrm{colocaln}((\Gamma, y : @_l S), u, U, V) = \mathrm{colocaln}(\Gamma, u, U, V)$.

By Lemma 13.3 $\Gamma \vdash a@j \Rightarrow \Gamma, y : @_l S \vdash a@j$.

By $z \notin \mathrm{fl}(@_l ...)$ and $z \neq y$ we have $z \notin \mathrm{fl}(P)$.

Now the premises of the right hand instance of (Rep-)In:

By Lemma 13.2 $\Gamma \vdash \{u/z\}l : \mathrm{loc}$.

By Lemma 36 there is a type $\hat{T}$ such that $\Gamma \vdash \{u/z\}x : \hat{T} \wedge \hat{T} \leqslant \updownarrow_{io} S$.

By Lemma 7.3 $\hat{T} \Downarrow \hat{T}$ so by Lemma 5.1 $\hat{T}$ is not of the form $\mu X$ _ so by the subtyping rules there are $i'o'$ and $T$ such that $\hat{T} = \updownarrow_{i'o'} T$. By the subtyping rules $i'o' \leqslant io$.

Also, as $i \leqslant \mathsf{L}$ we have $\mathrm{covariant}(io) \vee \mathrm{nonvariant}(io)$ so $T \leqslant S$.

By the right hand substitution lemma $\Gamma, y : @_l S \vdash \{u/z\}P : \mathrm{process}$.

It is clear that $\Gamma, y : @_l T \leqslant \Gamma, y : @_l S$ so by Lemma 34.8 $\Gamma, y : @_l T \vdash \{u/z\}P : \mathrm{process}$.

By transitivity $i' \leqslant \mathsf{L}$.

Now suppose $i' = \mathsf{L}$. It follows that $i = \mathsf{L}$ so $\Gamma, z : @_j V \vdash x@l$. The next part is exactly as in Out.

Case $z \neq x$: By Lemma 13.3 $\Gamma \vdash x@l$ so $\Gamma \vdash (\{u/z\})x@(\{u/z\}l)$ .

Case $z = x$: By value typing and location rules $V \Downarrow \updownarrow_{io} S$ and $j = l$.

By Lemma 7.1 $U = \hat{T}$ so $U = \updownarrow_{i'o'} T$.

By Lemmas 16 and 13.1 we have $\Gamma \vdash \updownarrow_{i'o'} T : \mathrm{Type}_{\_\_}$, $\Gamma \vdash \updownarrow_{io} S : \mathrm{Type}_{\_\_}$ and $\updownarrow_{i'o'} T \leqslant \updownarrow_{io} S$. As $\mathrm{colocal}(io, i'o')$ we have $\mathrm{colocal}(\Gamma, \updownarrow_{i'o'} T, \updownarrow_{io} S)$.

By the value typing rules $u$ is a name, so $u \in \mathrm{colocaln}(\Gamma, u, \updownarrow_{i'o'} T, \updownarrow_{io} S)$.

By the definition of colocaln $u \in \mathrm{colocaln}(\Gamma, u, \updownarrow_{i'o'} T, V) = \mathrm{colocaln}(\Gamma, u, U, V)$.

By the premise of the substitution lemma $\Gamma \vdash u@j$ so $\Gamma \vdash u@l$.

By Lemma 35 $\mathrm{fl}(\{u/z\}P) = \mathrm{fl}(P) \subseteq \{l\} = \{\{u/z\}l\}$.

**Mig** To show the instance of the substitution lemma on the left we suppose we have its premises, and hence the left instance of Mig. We show, using the right instance of the substitution lemma, that we have the right instance of Mig.

$$\frac{\begin{array}{l}\Gamma, z : @_j V \vdash @_l \textbf{migrate\_to}\ v\ \textbf{then}\ P : \mathrm{process}\\ \Gamma \vdash u : U\\ U \leqslant V\\ \forall a \in \mathrm{colocaln}(\Gamma, u, U, V)\ .\ \Gamma \vdash a@j\\ z \notin \mathrm{fl}(@_l \textbf{migrate\_to}\ v\ \textbf{then}\ P)\end{array}}{\Gamma \vdash \{u/z\}@_l \textbf{migrate\_to}\ v\ \textbf{then}\ P : \mathrm{process}} \qquad \frac{\begin{array}{l}\Gamma, z : @_j V \vdash P : \mathrm{process}\\ \Gamma \vdash u : U\\ U \leqslant V\\ \forall a \in \mathrm{colocaln}(\Gamma, u, U, V)\ .\ \Gamma \vdash a@j\\ z \notin \mathrm{fl}(P)\end{array}}{\Gamma \vdash \{u/z\}P : \mathrm{process}}$$

$$\textsc{Mig}\ \frac{\begin{array}{l}\Gamma, z : @_j V \vdash l : \mathrm{loc}\\ \Gamma, z : @_j V \vdash v : \mathrm{loc}\\ \Gamma, z : @_j V \vdash P : \mathrm{process}\\ \mathrm{fl}(P) \subseteq \{l\}\end{array}}{\Gamma, z : @_j V \vdash @_l \textbf{migrate\_to}\ v\ \textbf{then}\ P : \mathrm{process}} \qquad \textsc{Mig}\ \frac{\begin{array}{l}\Gamma \vdash \{u/z\}l : \mathrm{loc}\\ \Gamma \vdash \{u/z\}v : \mathrm{loc}\\ \Gamma \vdash \{u/z\}P : \mathrm{process}\\ \mathrm{fl}(\{u/z\}P) \subseteq \{\{u/z\}l\}\end{array}}{\Gamma \vdash @_{\{u/z\}l} \textbf{migrate\_to}\ \{u/z\}v\ \textbf{then}\ \{u/z\}P : \mathrm{process}}$$

The premises of the right instance of the substitution lemma are all immediate except for $z \notin \text{fl}(P)$, which follows from the last premise of the left instance of MIG.

For the premises of the right instance of MIG:

By $z \notin \text{fl}(@_l\ldots)$ have $z \neq l$ so $\{u/z\}l = l$. By Lemma 13.2 $\Gamma \vdash \{u/z\}l : \text{loc}$.

By Lemma 36 there is a type $T'$ such that $\Gamma \vdash \{u/z\}v : T' \wedge T' \leqslant \text{loc}$.

By Lemma 7.3 $T' \Downarrow T'$ so $T'$ is not of the form $\mu X\ \_$ so by the subtyping rules $T' = \text{loc}$ so $\Gamma \vdash \{u/z\}v : \text{loc}$.

By the right instance of the substitution lemma $\Gamma \vdash \{u/z\}P : \text{process}$.

By Lemma 35 $\text{fl}(\{u/z\}P) = \text{fl}(P)$. By the premise of the left hand instance of MIG $\text{fl}(P) \subseteq \{l\} = \{\{u/z\}l\}$.

**LET** To show the instance of the substitution lemma at the top we suppose we have its premises, and hence the upper instance of LET. We show, using the lower instance of the substitution lemma, that we have the lower instance of LET.

$$\frac{\begin{array}{l}\Gamma, z : @_j V \vdash @_l \textbf{let}\ \langle y_1 : T_1, y_2 : T_2\rangle = v\ \textbf{in}\ P : \text{process}\\ \Gamma \vdash u : U\\ U \leqslant V\\ \forall a \in \text{colocaln}(\Gamma, u, U, V)\ .\ \Gamma \vdash a@j\\ z \notin \text{fl}(@_l \textbf{let}\ \langle y_1 : T_1, y_2 : T_2\rangle = v\ \textbf{in}\ P)\end{array}}{\Gamma \vdash \{u/z\}@_l \textbf{let}\ \langle y_1 : T_1, y_2 : T_2\rangle = v\ \textbf{in}\ P : \text{process}}$$

$$\frac{\begin{array}{l}\Gamma, y_1 : @_l T_1, y_2 : @_l T_2, z : @_j V \vdash P : \text{process}\\ \Gamma, y_1 : @_l T_1, y_2 : @_l T_2 \vdash u : U\\ U \leqslant V\\ \forall a \in \text{colocaln}(\Gamma, y_1 : @_l T_1, y_2 : @_l T_2, u, U, V)\ .\ \Gamma, y_1 : @_l T_1, y_2 : @_l T_2 \vdash a@j\\ z \notin \text{fl}(P)\end{array}}{\Gamma, y_1 : @_l T_1, y_2 : @_l T_2 \vdash \{u/z\}P : \text{process}}$$

$$\text{LET}\ \frac{\begin{array}{l}\Gamma, z : @_j V \vdash l : \text{loc}\\ \Gamma, z : @_j V \vdash v : S'\\ \Gamma, z : @_j V, y_1 : @_l T_1, y_2 : @_l T_2 \vdash P : \text{process}\\ S' \leqslant T_1 \times T_2\\ \forall a \in \text{colocaln}(\Gamma, z : @_j V, v, S', T_1 \times T_2)\ .\ \Gamma, z : @_j V \vdash a@l\\ \text{fl}(P) \subseteq \{l\}\end{array}}{\Gamma, z : @_j V \vdash @_l \textbf{let}\ \langle y_1 : T_1, y_2 : T_2\rangle = v\ \textbf{in}\ P : \text{process}}$$

$$\text{LET}\ \frac{\begin{array}{l}\Gamma \vdash \{u/z\}l : \text{loc}\\ \Gamma \vdash \{u/z\}v : T'\\ \Gamma, y_1 : @_{\{u/z\}l} T_1, y_2 : @_{\{u/z\}l} T_2 \vdash \{u/z\}P : \text{process}\\ T' \leqslant T_1 \times T_2\\ \forall a \in \text{colocaln}(\Gamma, \{u/z\}v, T', T_1 \times T_2)\ .\ \Gamma \vdash a@l\\ \text{fl}(\{u/z\}P) \subseteq \{\{u/z\}l\}\end{array}}{\Gamma \vdash @_{\{u/z\}l} \textbf{let}\ \langle y_1 : T_1, y_2 : T_2\rangle = \{u/z\}v\ \textbf{in}\ \{u/z\}P : \text{process}}$$

Premises of second instance of substitution lemma:

By Lemma 30 $\vdash \Gamma, z : @_j V, y_1 : @_l T_1, y_2 : @_l T_2$ ok.

By context formation $j \notin \{y_1, y_2\}$ and by $z \notin \text{fl}(\ldots)$ $l \neq z$.

By Lemma 31 $\Gamma, y_1 : @_l T_1, y_2 : @_l T_2, z : @_j V \vdash P : \text{process}$.

By Lemma 13.2 $\Gamma, y_1 : @_l T_1, y_2 : @_l T_2 \vdash u : U$.

By Lemma 28.3 $\text{colocaln}((\Gamma, y_1 : @_l T_1, y_2 : @_l T_2), u, U, V) = \text{colocaln}(\Gamma, u, U, V)$.

By Lemma 13.3 $\Gamma \vdash a@j \Rightarrow \Gamma, y_1 : @_l T_1, y_2 : @_l T_2 \vdash a@j$.

By $z \notin \{y_2, y_2\}$ we have $z \notin \mathrm{fl}(P)$.

Premises of lower instance of LET:

By Lemma 13.2 $\Gamma \vdash \{u/z\}l : \mathrm{loc}$.

By Lemma 36 there is a type $T'$ such that $\Gamma \vdash \{u/z\}v : T' \wedge T' \leqslant S'$.

By transitivity $T' \leqslant T_1 \times T_2$.

Suppose $a \in \mathrm{colocaln}(\Gamma, \{u/z\}v, T', T_1 \times T_2)$.

By Lemma 37 one of the following hold.

1. $a \neq z$ and $a \in \mathrm{colocaln}((\Gamma, z : @_j V), v, S', T_1 \times T_2)$
2. $a \in \mathrm{colocaln}(\Gamma, u, U, V)$ and $z \in \mathrm{colocaln}((\Gamma, z : @_j V), v, S', T_1 \times T_2)$

In the first case by the premise of the upper instance of LET we have $\Gamma, z : @_j V \vdash a@l$ so by Lemma 13.3 $\Gamma \vdash a@l$.

In the second case by the premise of the substitution lemma we have $\Gamma \vdash a@j$. Moreover, by the premise of the upper instance of LET we have $\Gamma, z : @_j V \vdash z@l$ so $j = l$ so $\Gamma \vdash a@l$.

Finally fl(...) as in MIG.

**NEW**

$$\frac{\begin{array}{l}\Gamma, z : @_j V \vdash (\mathbf{new}\, y : @_l S)P : \mathrm{process} \\ \Gamma \vdash u : U \\ U \leqslant V \\ \forall a \in \mathrm{colocaln}(\Gamma, u, U, V)\ .\ \Gamma \vdash a@j \\ z \notin \mathrm{fl}((\mathbf{new}\, y : @_l S)P)\end{array}}{\Gamma \vdash \{u/z\}(\mathbf{new}\, y : @_l S)P : \mathrm{process}} \qquad \frac{\begin{array}{l}\Gamma, y : @_l S, z : @_j V \vdash P : \mathrm{process} \\ \Gamma, y : @_l S \vdash u : U \\ U \leqslant V \\ \forall a \in \mathrm{colocaln}(\Gamma, y : @_l S, u, U, V)\ .\ \Gamma, y : @_l S \vdash a@j \\ z \notin \mathrm{fl}(P)\end{array}}{\Gamma, y : @_l S \vdash \{u/z\}P : \mathrm{process}}$$

$$\textsc{New}\ \frac{\begin{array}{c}\Gamma, z : @_j V \vdash S : \mathrm{Type}_{-\mathsf{E}} \\ \Gamma, z : @_j V, y : @_l S \vdash P : \mathrm{process}\end{array}}{\Gamma, z : @_j V \vdash (\mathbf{new}\, y : @_l S)P : \mathrm{process}} \qquad \textsc{New}\ \frac{\begin{array}{c}\Gamma \vdash S : \mathrm{Type}_{-\mathsf{E}} \\ \Gamma, y : @_{\{u/z\}l} S \vdash \{u/z\}P : \mathrm{process}\end{array}}{\Gamma \vdash (\mathbf{new}\, y : @_{\{u/z\}l} S)\{u/z\}P : \mathrm{process}}$$

Premises of right hand substitution lemma (exactly as in (REP-)IN):

By Lemma 30 $\vdash \Gamma, z : @_j V, y : @_l S\ ok$.

By context formation $j \neq y$ and by $z \notin \mathrm{fl}(@_l ...)$ we have $z \neq l$.

By Lemma 31 $\Gamma, y : @_l S, z : @_j V \vdash P : \mathrm{process}$.

By Lemma 13.2 $\Gamma, y : @_l S \vdash u : U$.

By Lemma 28.3 $\mathrm{colocaln}((\Gamma, y : @_l S), u, U, V) = \mathrm{colocaln}(\Gamma, u, U, V)$.

By Lemma 13.3 $\Gamma \vdash a@j \Rightarrow \Gamma, y : @_l S \vdash a@j$.

By $z \notin \mathrm{fl}(@_l ...)$ and $z \neq y$ we have $z \notin \mathrm{fl}(P)$.

Now the premises of the right hand instance of NEW:

By Lemma 13.1 $\Gamma \vdash S : \mathrm{Type}_{-\mathsf{E}}$.

**NIL** Trivial.

**PAR** By induction.

□

## A.9 Subject reduction

**Proof (of Lemma 39)** Induction on $\Delta$. □

**Proof (of Lemma 40)** Inductions on $P \equiv Q$. The $\mathrm{fl}(\_)$ part is straightforward. For the other part, we check each equation:

1: By Lemma 30.

2,3: Trivial.

4: Assume w.l.g. that $x$, $y$ and $\mathrm{dom}(\Gamma)$ are all distinct. By typing $u$ and $v$ must be names, say $k$ and $l$. The result then follows from Lemmas 13.1 and 31.

5: By Lemma 32. □

## References

[Aba97] Martín Abadi. Secrecy by typing in security protocols. To appear in TACS '97 (open lecture), September 1997.

[AG97] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Proceedings of the Fourth ACM Conference on Computer and Communications Security, Zürich*, pages 36–47. ACM Press, April 1997. Long version as Technical Report 414, University of Cambridge.

[Ama97] R. M. Amadio. An asynchronous model of locality, failure, and process mobility. In *COORDINATION 97, Berlin. LNCS 1282*, 1997. Rapport Interne LIM February 1997, and INRIA Research Report 3109. To appear.

[AP94] R. M. Amadio and S. Prasad. Localities and failures. In P. S. Thiagarajan, editor, *Proceedings of $14^{th}$ FST and TCS Conference, FST-TCS'94. LNCS 880*, pages 205–216. Springer-Verlag, 1994.

[Bou92] Gérard Boudol. Asynchrony and the $\pi$-calculus (note). Rapport de Recherche 1702, INRIA Sofia-Antipolis, May 1992.

[CG97] Luca Cardelli and Andrew D. Gordon. Mobile ambients. Draft, July 1997.

[FG96] Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In *Proceedings of the 23rd POPL*, pages 372–385. ACM press, January 1996.

[FGL$^+$96] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *Proceedings of CONCUR '96. LNCS 1119*, pages 406–421. Springer-Verlag, August 1996.

[FLMR97] Cédric Fournet, Cosimo Laneve, Luc Maranget, and Didier Rémy. Implicit typing à la ML for the join-calculus. In *Proceedings of CONCUR '97. LNCS 1243*, pages 196–212. Springer-Verlag, July 1997.

[Gay93] Simon J. Gay. A sort inference algorithm for the polyadic $\pi$-calculus. In *Proceedings of the 20th POPL*. ACM Press, 1993.

[HH94] Andy Harter and Andy Hopper. A distributed location system for the active office. *IEEE Network*, 8(1), January 1994. See also Olivetti Research Ltd. Technical Report 94.1.

[HR97] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. Draft, 1997.

[HT92] Kohei Honda and Mario Tokoro. On asynchronous communication semantics. In M. Tokoro, O. Nierstrasz, and P. Wegner, editors, *Object-Based Concurrent Computing. LNCS 612*, pages 21–51, 1992.

[KHJH95] Josva Kleist, Martin Hansen, Bo Jensen, and Hans Hüttel. Inferring effect types in an applicative language with asynchronous concurrency. In *Proceedings of Second ACM SIGPLAN Workshop on State in Programming Languages (SIPL 95)*, pages 49–64, 1995.

[KNY95] Naoki Kobayashi, Motoki Nakade, and Akinori Yonezawa. Static analysis of communication for asynchronous concurrent programming languages. In *Second International Static Analysis Symposium (SAS'95), LNCS 983*, pages 225–242, 1995. See also Technical Report 95–04, University of Tokyo, Department of Information Science.

[Kob97] Naoki Kobayashi. A partially deadlock-free typed process calculus. In *Proceedings of LICS '97*, pages 128–139. Computer Science Press, July 1997. Full version as Technical Report 97-02, University of Tokyo.

[KPT96] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. In *Proceedings of the 23rd POPL*, pages 358–371. ACM press, January 1996.

[LW95] Xinxin Liu and David Walker. A polymorphic type system for the polyadic $\pi$-calculus. In *Proceedings of CONCUR '95. LNCS 962*, pages 103–116, 1995.

[Mil96] Robin Milner. Calculi for interaction. *Acta Informatica*, 33:707–737, 1996.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Parts I + II. *Information and Computation*, 100(1):1–77, 1992.

[Nie96] Joachim Niehren. Functional computation as concurrent computation. In *Proceedings of the 23rd POPL*, pages 333–343. ACM Press, January 1996.

[NN94] Hanne Riis Nielson and Flemming Nielson. Higher-order concurrent programs with finite communication topology. In *Proceedings of the 21st POPL*. ACM Press, 1994.

[NN95] Hanne Riis Nielson and Flemming Nielson. Static and dynamic processor allocation for higher-order concurrent languages. In *Proceedings of TAPSOFT 95 (FASE). LNCS 915*, 1995.

[NS97] Uwe Nestmann and Martin Steffen. Typing confluence. In Stefania Gnesi and Diego Latella, editors, *Proceedings of FMICS'97: Second International ERCIM Workshop on Formal Methods in Industrial Critical Systems (Cesena, Italy)*, July 1997.

[Ode95] Martin Odersky. Polarized name passing. In *Proceedings of FSTTCS '95. LNCS 1026*, December 1995.

[PS96] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996.

[PS97] Benjamin Pierce and Davide Sangiorgi. Behavioural equivalence in the polymorphic pi-calculus. In *Proceedings of the 24th POPL*, pages 242–255. ACM press, January 1997. Full version available as INRIA-Sophia Antipolis Rapport de Recherche No. 3042 and as Indiana University Computer Science Technical Report 486.

[PT97] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. Technical Report CSCI 476, Computer Science Department, Indiana University, 1997. To appear in *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, MIT Press.

[RH97] James Riely and Matthew Hennessy. Distributed processes and location failures. In *Proceedings of ICALP '97. LNCS 1256*, pages 471–481. Springer-Verlag, July 1997.

[San96] Davide Sangiorgi. An interpretation of typed objects into typed $\pi$-calculus. Rapport de Recherche RR-3000, INRIA Sophia-Antipolis, August 1996.

[San97] Davide Sangiorgi. The name discipline of uniform receptiveness. In *Proceedings of ICALP '97. LNCS 1256*, pages 303–313, 1997. Full version as Technical Report, INRIA Sophia-Antipolis, December 1996.

[Sew97] Peter Sewell. On implementations and semantics of a concurrent programming language. In *Proceedings of CONCUR '97. LNCS 1243*, pages 391–405. Springer-Verlag, 1997.

[Ste96] Paul Steckler. Detecting local channels in distributed Poly/ML. Technical Report ECS-LFCS-96-340, University of Edinburgh, January 1996.

[SY97] Tatsurou Sekiguchi and Akinori Yonezawa. A calculus with code mobility. In Howard Bowman and John Derrick, editors, *Formal Methods for Open Object-based Distributed Systems (Proceedings of FMOODS '97)*, pages 21–36. IFIP, Chapman and Hall, July 1997.

[TLK96] Bent Thomsen, Lone Leth, and Tsung-Min Kuo. A Facile tutorial. In *Proceedings of CONCUR '96. LNCS 1119*, pages 278–298. Springer-Verlag, August 1996.

[Tur96] David N. Turner. *The Polymorphic Pi-calculus: Theory and Implementation*. PhD thesis, University of Edinburgh, 1996.

[Vas94] Vasco Thudichum Vasconcelos. Predicative polymorphism in $\pi$-calculus. In *Proceedings of PARLE '94. LNCS 817*, pages 425–437, July 1994.

[VH93] Vasco Thudichum Vasconcelos and Kohei Honda. Principal typing schemes in a polyadic $\pi$-calculus. In *Proceedings of CONCUR '96. LNCS 715*, pages 524–538, 1993.

[VT97] J. Vitek and C. Tschudin, editors. *Mobile Object Systems – Towards the Programmable Internet. LNCS 1222*. Springer-Verlag, 1997.