Semantic subtyping for the π-calculus

Giuseppe Castagna — École Normale Supérieure, Paris
Rocco De Nicola — Università di Firenze
Daniele Varacca — Imperial College, London

Abstract

Subtyping relations for the π-calculus are usually defined in a syntactic way, by means of structural rules. We propose a semantic characterisation of channel types and use it to derive a subtyping relation.

The type system we consider includes read-only and write-only channel types, as well as boolean combinations of types. A set-theoretic interpretation of types is provided, in which boolean combinations are interpreted as the corresponding set-theoretic operations. Subtyping is defined as inclusion of the interpretations. We prove the decidability of the subtyping relation and sketch the subtyping algorithm.

In order to fully exploit the type system, we define a variant of the π-calculus where communication is subjected to pattern matching that performs dynamic typecase.

Contents

1 Introduction and motivations

In this paper we study a type system for a concurrent process language in which values are exchanged between agents via communication channels that can be dynamically generated. The language we consider is a variant of the asynchronous π-calculus, where communication is subjected to pattern matching.

There exists a well established literature on typing and subtyping for the π-calculus. However, all the approaches we are aware of rely on subtyping relations or on type equivalences that are defined syntactically, by means of structural rules.

In our view, such syntactic formalisations of typing relations miss a clean semantic intuition of types. Consider, for example, the type system defined by Hennessy and Riely [8], which is one of the most advanced type systems for variants of the π-calculus. It includes read-only and write-only channels, as well as union and intersection types.[1] In that system the following equality is introduced:

$$ch^+(\mathtt{t}_1) \vee ch^+(\mathtt{t}_2) = ch^+(\mathtt{t}_1 \vee \mathtt{t}_2) \qquad (1)$$

where $ch^+(t)$ is the type of channels from which we can only read values of type t, and $\vee$ denotes union. We would like to understand the precise semantic intuition that underlies an equation such as (1).

Semantic subtyping. The basic idea is simple: the semantics of a type is the set of its values, and union, intersection and negation types are interpreted using the corresponding set theoretical operators. Subtyping is then defined as inclusion of the interpretations. However, the subtyping relation is needed in order to type the values, usually by subsumption. We are therefore trapped in a circle, where we need subtyping to define typing, that defines the interpretation, that defines the subtyping. We are able to break this circle via a "fixed point" construction.

[1] As a matter of fact, union and intersections of [8] are meta-combinators used only by the type system. The left-hand side of equation (1) could be considered as an alternative notation for the right-hand side.

Before even having defined the language, and therefore before even knowing what values are, we define a "bootstrap" semantics of types, that is used to define the subtyping relation. This subtyping relation is then used to type values. This gives us another semantics of types, as sets of values. The key point is that, if we choose the right bootstrap semantics, the values semantics will correspond to the bootstrap semantics, and the circle will be closed.

Channels as boxes. In order to understand how channels and channel types relate, we have to provide a semantic account of channels. Our intuition is that a channel is a box in which we can put things (write) and from which we can take things (read). The type of a channel, then, is characterised by the set of the things the box can contain. That is, a channel of type $ch^+(t)$ is a box in which we must expect to find objects of type t and, similarly, a channel of type $ch^-(t)$ is a box in which we are allowed to put objects of type t. But if one takes this stand, the equality (1) does not seem to be justified. Consider the types $ch^+(\texttt{candy}) \vee ch^+(\texttt{coal})$ and $ch^+(\texttt{candy} \vee \texttt{coal})$. Both represent boxes. If we have a box of the first type, then we expect to find in it either a candy of a piece of charcoal, but we know it is always one of the two. For instance, if we use the box twice, the second time we will know what present it contains. A box of the second type, instead, is a "surprise box" as it can always give us both candies and charcoal. Our intuition suggests that the two types above are different because they characterise two different kinds of objects.

The role of the language. So why did Hennessy and Riely require (1)? The point is that, if in the language under consideration there is no syntactic construction that can tell apart a $ch^+(\texttt{candy})$ channel from a $ch^+(\texttt{coal})$ channel (e.g. a typecase), then it is not possible to operationally observe any difference between the types in (1). On the contrary, if it is possible to test whether on a channel c we are receiving a channel of type $ch^+(\texttt{candy})$ or a channel of type $ch^+(\texttt{coal})$, then a rule such as (1) would give rise to an unsound system, because it would allow c to carry a channel of type $ch^+(\texttt{candy} \vee \texttt{coal})$ which makes the test on c crash (since the possibility that the argument is a $ch^+(\texttt{candy} \vee \texttt{coal})$ box is not contemplated). We define a variant of the π-calculus, called the $\mathbb{C}\pi$-calculus, that exploits the full power of our new type system, and in particular that permits dynamically testing the type of values received on a channel. We implement the dynamic test by endowing input actions with patterns, and allowing synchronisation when pattern matching succeeds. The result is a simple and elegant formalism that can be easily extended with product types, to obtain a polyadic $\mathbb{C}\pi$-calculus, and with a restricted form of recursive types.

Advantages of a semantic approach. The main advantage of using a semantic approach is that the types have a natural and intuitive set theoretic interpretation as sets of their values. This property turns out to be very helpful not only to understand the meaning of the types, but also to reason about them. For instance, the subtyping algorithm is deduced just by applying set-theoretic properties, in the proofs we can rewrite types by using set-theoretic laws, and the typing of pattern matching can be better understood in terms of set-theoretic operations (e.g. the second pattern in an alternative will have to filter all that was not already matched by the first pattern: set theoretic difference).

The language $\mathbb{C}$Duce [2] also demonstrated the practical impact of the semantic approach: subtyping results are easier to understand for a programmer, since she does not have to reason in terms of subtyping rules but rather of set-theoretic operations. Furthermore, the compiler/interpreter can return much more precise and meaningful error messages.

For instance if type-checking fails the compiler returns a value or a witness that is in the set-theoretic difference between the deduced type and the expected type, and this information helps the programmer understand why type-checking failed.

For a wider discussion on the advantages of semantic subtyping we refer the reader to Castagna and Frisch's introductory paper [**?**].

Main contributions. This work provides several contributions: We define a very expressive type and subtype system for the π-calculus with read-only and write-only channel types, product types, and complete boolean combinations of types. We define a set-theoretic denotational model for the types, where boolean combinations are interpreted as the corresponding set-theoretic operations and channel types are interpreted as sets of boxes. We use the model to define subtyping as set-theoretic containment. We show how to extend the π-calculus in order to fully exploit the expressiveness of the type system, in particular by endowing input actions with pattern matching. Finally we show that in that setting the typing and subtyping relations are decidable. A further contribution of this work is the opening of a new way to integrate functional and concurrent features in the same calculus: this will be done by fully integrating (our new version of) π and $\mathbb{C}$Duce systems, to yield a calculus with dynamic type dispatch, overloading, channelled communications and where both functions and channels have first class citizenship.

For lack of space proofs are omitted, but they can be found in the extended version of this paper, available at `http://www.cduce.org`.

Related work. The first work on subtyping for π was done by Pierce and Sangiorgi [11] and successively extended in several other works [13, 5, 14].

The work closest to ours, at least for the expressiveness

of the types, is the already cited work of Hennessy and Riely [8]. As far as π-types are concerned, our work subsumes their system in the sense that it defines a richer subtyping relation; this can be checked by observing that their type $\mathsf{rw}\langle s,t\rangle$ corresponds to the intersection $ch^+(s) \wedge ch^-(t)$ of our formalism.

Brown *et al.* [4] enrich π with XML-like values that are deconstructed by pattern matching. The patterns they use are quite different from the one we introduce here: for example they have patterns to match the interleaving of values, which we do not consider. On the other hand they do not consider types, which are the main motivation of our work.

Acciai and Boreale [1] (independently from our work) define a language similar to ours, with XDuce-like pattern matching. However the type system they propose is less rich than ours and, most importantly, their subtyping relation is defined syntactically.

As for the technical issues of semantic subtyping, our starting point is the work developed by Frisch *et al.* for functional programming languages [7, 6], that led to the design of ℂDuce [2].

Plan of the paper: In Section 2 we introduce the type system and define the subtyping relation in terms of a set-theoretic interpretation of the types. We prove the decidability of subtyping, sketch the subtyping algorithm and conclude with the definition of patterns and pattern matching whose semantics is completely specified in terms of the model of types. In Section 3 we define the syntax and semantics of a pattern-based extension of π-calculus that fully exploits the previous type system, and give relevant examples of their use. In Section 4 we consider the polyadic version of our calculus, we enrich it with recursive types and show, when possible, how semantic and decidability properties extend to this setting. We conclude by

2 Types and subtyping

We shall start with a relatively simple system with just base types, channel types and boolean combinators. In a second moment we shall add the product type constructor and a restricted form of recursive types.

2.1 Types

In the simplest of our type systems, a type is inductively built by applying *type constructors*, namely base type constructors (e.g. integers, strings, etc...), the input or the output channel type constructor, or by applying a *boolean combinator*, i.e., union, intersection, and negation:

$$\begin{array}{lrcll} \textit{Types} & t & ::= & b \mid ch^+(t) \mid ch^-(t) & \text{constructors} \\ & & \mid & \mathbf{0} \mid \mathbf{1} \mid \neg t \mid t \vee t \mid t \wedge t & \text{combinators} \end{array}$$

Combinators are self-explanatory, with $\mathbf{0}$ being the empty type and $\mathbf{1}$ the type of all values. For what concerns type constructors, $ch^+(t)$ denotes the type of those channels that can be used to *input* only values of type t. Symmetrically $ch^-(t)$ denotes the type of those channels that can be used to *output* only values of type t. The read and write channel type $ch(t)$ is absent from our definition. We shall use it only as syntactic sugar for $ch^-(t) \wedge ch^+(t)$, that is the type of channels that can be used to read only *and* to write only values of type t. The set of all types (sometimes referred to as "type algebra") will be denoted by $\mathscr{T}$.

Although types $ch(t)$ are just syntactic sugar, they will play a crucial role in the rest of the paper. In particular, we shall see that the types of the form $ch(t)$ are all and the only types that are not base types and that denote a singleton. We shall use them quite often because they are the most precise type of channels (see, e.g., the typing rule (chan) in Section 3.4).

In our approach channels are physical boxes where one can insert and withdraw objects of a given type. Our intuition is that there is not such a thing as a read-only or write-only box: each box is associated to a type t and one can always write and read objects of that type into and from such a box. Thus the type of $ch^+(t)$ can be considered just a constraint telling that a variable of that type will be bound only to boxes from which one can read objects of type t. If we know that a message has type $ch^+(t)$, it *does not* mean that we cannot write into it, we simply do not have any information about what can be written in it: for instance this message could be a box that cannot contain any object. What the type tells us is simply that we had better avoid writing into it since, in the absence of further information, no writing will be safe. Similarly, if a message is of type $ch^-(t)$, then we know that it can only be a box in which writing an object of type t is safe, but we have no information about what could be read from that channel, since the message might be a box that can contain any object. Therefore we had better avoid reading from it, unless we are ready to accept anything. However, if we *are* ready to accept anything, then our type system guarantees that we can read on a channel with type $ch^-(t)$ because, as we will see, we have $ch^-(t) \leq ch^+(\mathbf{1})$.

It should be clearer now why we identify $ch(t)$ and $ch^+(t) \wedge ch^-(t)$: the intersection requires that on channels of type $ch^+(t) \wedge ch^-(t)$ we must be able both to write objects of type t and to read object of (the same) type t; this means that the channels can contain all and only messages of type t. To say it with other words, we have that a $ch^{\nu}(t)$ type (with ν standing for either $+$ or $-$) indicates what can be *safely done* relatively to its values, and *not* what is forbidden; thus, the values in the intersection of two types are permitted by both types.

Notice that, if we had interpreted types as interdictions then we should consider $ch^+(t) \wedge ch^-(t)$ as the channels on

which we cannot write *and* we cannot read: this would be the empty channel.

2.2 Intuitive semantics of types

Our leading intuition is that a type should denote the set of values of that type. That is

$$[\![t]\!] = \{v \mid \vdash v : t\} .$$

In our approach—where subtyping is defined as containment of type interpretations—this means that $s \leq t$ if and only if every value of type s is also of type t. The basic types (integers, strings) should denote subsets of a set of basic values $\mathbb{B}$. The boolean operators over types should be interpreted by using the boolean operators over sets. By following our intuition we shall have that the interpretation of the type $ch(t)$ has to denote the set of all boxes (i.e. channels) that can contain objects of type t:

$$[\![ch(t)]\!] = \{c \mid c \text{ is a box for objects in } [\![t]\!]\} . \qquad (2)$$

Since every box is uniquely associated to a type, then the interpretations of channel types are pairwise disjoint. This already gives invariance of channel types: $[\![ch(t)]\!] \subseteq [\![ch(s)]\!]$ if and only if $[\![t]\!] = [\![s]\!]$.

Starting from the above interpretation of $ch(t)$, we can now provide a semantics for $ch^+(t)$ and $ch^-(t)$. As said, the former should denote the set of all boxes from which one can safely expect to get only objects of type t. Thus we require that $ch^+(t)$ denote all boxes for objects of type t, but also all boxes for objects of type s, for any $s \leq t$. Indeed, by subsumption, objects of types s are also of type t. Dually, $ch^-(t)$ should denote the set of all boxes in which one can safely put objects of type t. Therefore it will denote all boxes that can contain objects of type s, for any $s \geq t$. Let us write c^t to denote a box for objects of type t. We have

$$[\![ch^+(t)]\!] = \left\{c^s \mid s \leq t\right\} , \qquad [\![ch^-(t)]\!] = \left\{c^s \mid s \geq t\right\} .$$

Given the above semantic interpretation, from the viewpoint of types all the boxes of one given type t are indistinguishable, because either they all belong to the interpretation of one type or they all do not. This implies that the subtyping relation is insensitive to the actual number of boxes of a given type. We can therefore assume that for every equivalence class of types, there is only one such box, which may as well be identified with $[\![t]\!]$, so that the intended semantics of channel types would be

$$[\![ch^+(t)]\!] = \left\{[\![s]\!] \mid s \leq t\right\} , \; [\![ch^-(t)]\!] = \left\{[\![s]\!] \mid s \geq t\right\} . \quad (3)$$

We have that this semantics induces covariance of input types and contravariance of output types. Moreover, as anticipated, we have that $ch(t) = ch^-(t) \wedge ch^+(t)$ since the types on both sides of the equality have the same semantics—namely, the singleton $\{[\![t]\!]\}$—and therefore it is justified to consider $ch(t)$ as syntactic sugar for the type on the right rather than a type constructor.

Our proposed interpretation has other interesting features that we shall describe later. In the meanwhile, the reader who wants to familiarise with our semantics can try to use the above definitions to verify that the difference $ch^+(t) \setminus ch^-(t)$ and the difference $ch^+(t) \setminus ch(t)$ have the same interpretation, and that the same holds for $ch^+(\mathbf{1})$ and $ch^-(\mathbf{0})$. [2] According to the discussion above, in order to define the semantics of a channel type, we need to know the subtyping relation. And here we are in the presence of a circular definition. We use the subtyping relation in order to build the interpretation that we need in order to define the subtyping relation. We devote the next section to solve this problem.

2.3 Building a model

The minimal requirement for an interpretation function is that boolean combinators should be interpreted in the corresponding set-theoretical operators, and that basic values and channels should have disjoint interpretations.

Definition 2.1 *Let $\mathscr{D}$,and $\mathbb{B}$ be sets such that $\mathbb{B} \subseteq \mathscr{D}$, and let $[\![\,]\!]$ be a function from $\mathscr{T}$ to $\mathscr{P}(\mathscr{D})$. $(\mathscr{D}, [\![\,]\!])$ is a* pre-model *if*

- $[\![b]\!] \subseteq \mathbb{B}$, $[\![ch^+(t)]\!] \cap \mathbb{B} = \varnothing$, $[\![ch^-(t)]\!] \cap \mathbb{B} = \varnothing$;
- $[\![\mathbf{1}]\!] = \mathscr{D}$, $[\![\mathbf{0}]\!] = \varnothing$;
- $[\![\neg t]\!] = \mathscr{D} \setminus [\![t]\!]$;
- $[\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$, $[\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$.

We use this interpretation to build another interpretation, according to the intended meaning of equations (3).

Definition 2.2 *Let $(\mathscr{D}, [\![\,]\!])$ be a pre-model. Let $[\![\mathscr{T}]\!]$ denote the image of the function $[\![\,]\!]$. The* extensional *interpretation of the types is the function $\mathscr{E}[\![\,]\!] : \mathscr{T} \to \mathscr{P}(\mathbb{B} + [\![\mathscr{T}]\!])$, defined as follows:*

- $\mathscr{E}[\![b]\!] = [\![b]\!]$;
- $\mathscr{E}[\![\mathbf{1}]\!] = \mathbb{B} + [\![\mathscr{T}]\!]$, $\mathscr{E}[\![\mathbf{0}]\!] = \varnothing$;
- $\mathscr{E}[\![\neg t]\!] = \mathscr{E}[\![\mathbf{1}]\!] \setminus \mathscr{E}[\![t]\!]$;
- $\mathscr{E}[\![t_1 \vee t_2]\!] = \mathscr{E}[\![t_1]\!] \cup \mathscr{E}[\![t_2]\!]$, $\mathscr{E}[\![t_1 \wedge t_2]\!] = \mathscr{E}[\![t_1]\!] \cap \mathscr{E}[\![t_2]\!]$;
- $\mathscr{E}[\![ch^+(t)]\!] = \{[\![s]\!] \mid [\![s]\!] \subseteq [\![t]\!]\}$;
- $\mathscr{E}[\![ch^-(t)]\!] = \{[\![s]\!] \mid [\![s]\!] \supseteq [\![t]\!]\}$.

A pre-model and its extensional interpretation induce, in principle, different preorders on types. We could use the extensional interpretation to build yet another interpretation, and so on. In order to close the circle, we shall consider a pre-model "acceptable" if it is a fixed point of this process, that is, if it induces the same containment relation as its extensional interpretation. This amounts to the following definition:

[2] The difference $s \setminus t$ is defined as $s \wedge \neg t$.

Definition 2.3 *A pre-model* $(\mathscr{D}, [\![\,]\!])$ *is a* model *if for every* t_1, t_2*, we have* $[\![t_1]\!] \subseteq [\![t_2]\!]$ *if and only if* $\mathscr{E}[\![t_1]\!] \subseteq \mathscr{E}[\![t_2]\!]$.

The last (and quite hard) point is to show that there actually exists a model, that is, that the condition imposed by Definition 2.3 can indeed be satisfied.

Paradoxically the model itself is not important. The subtyping relation is essentially characterised by the definition of extensional interpretation $\mathscr{E}[\![\,]\!]$. So what really matters is the proof that there exists at least one model. As the case of recursive types proves (see § 4.2), the existence of such a model is far from being trivial, and naive syntactic solutions —such as a term model— cannot be used.

Theorem 2.4 *There exists a model* $(\mathscr{D}, [\![\,]\!])$.

Types are stratified according to the nesting of the channel constructor. The model $(\mathscr{D}, [\![\,]\!])$ is obtained as the limit of a chain of models $(\mathscr{D}_n, [\![\,]\!]_n)$, built exploiting this stratification.

2.4 Examples of type (in)equalities

Given a model for the types, we define

$$s \leq t \stackrel{def}{\iff} [\![s]\!] \subseteq [\![t]\!]\,, \qquad\qquad s = t \stackrel{def}{\iff} [\![s]\!] = [\![t]\!]\,.$$

We list here some interesting equations and inequations between types that can be easily derived from the set-theoretic interpretation of types.

$$ch(t) \leq ch^-(\mathbf{0}) = ch^+(\mathbf{1}) \tag{4}$$

Every channel c can be safely used in a process that does not write on c and that does not care about what c returns.

$$ch^-(t_1) \wedge ch^-(t_2) = ch^-(t_1 \vee t_2) \tag{5}$$

If on a channel we can write values of type t_1 and values of type t_2, this means that we can write values of type $t_1 \vee t_2$. Dually

$$ch^+(t_1) \wedge ch^+(t_2) = ch^+(t_1 \wedge t_2) \tag{6}$$

if a channel is such that we always read from it values of type t_1 but also such that we always read from it values of type t_2, then what we read from it are actually values of type $t_1 \wedge t_2$.

Union of types, as we observed in the introduction, behaves differently from intersection; we only have:

$$\begin{array}{rcl} ch^+(t_1) \vee ch^+(t_2) & \leq & ch^+(t_1 \vee t_2)\,, \\ ch^-(t_1) \vee ch^-(t_2) & \leq & ch^-(t_1 \wedge t_2)\,. \end{array}$$

The type $ch^+(t_1) \wedge ch^-(t_2)$ is the type of a channel on which we can write values of type t_2 and from which we can read values of type t_1. We have

$$ch^+(t_1) \wedge ch^-(t_2) = \mathbf{0} \tag{7}$$

if and only if $t_2 \not\leq t_1$, i.e. we should expect to read at least what we can write.

In order to show the role of our definitions let us deduce this last equation. By definition, (7) holds if and only if $[\![ch^+(t_1) \wedge ch^-(t_2)]\!] = [\![\mathbf{0}]\!]$. By definition of model and the antisymmetry of $\subseteq$ this holds iff $\mathscr{E}[\![ch^+(t_1) \wedge ch^-(t_2)]\!] = \mathscr{E}[\![\mathbf{0}]\!]$. By definition of $\mathscr{E}[\![\,]\!]$ this holds iff $\{[\![t]\!] \,|\, [\![t]\!] \subseteq [\![t_1]\!]\} \cap \{[\![t]\!] \,|\, [\![t_2]\!] \subseteq [\![t]\!]\} = \varnothing$. By the reflexivity and transitivity of $\subseteq$ this holds iff $[\![t_2]\!] \not\subseteq [\![t_1]\!]$, that is, by definition of subtyping iff $t_2 \not\leq t_1$.

2.5 Decidability of subtyping

Using the semantic characterisation of the types, we are also able to prove the decidability of the subtyping relation. The decision procedure is quite complicated, and in particular it involves finite and atomic types.

Definition 2.5 *An* atom *is a minimal nonempty type. A type is* finite *if it is equivalent to a finite union of atoms.*

The reader can think of an atom roughly as a singleton.

We start by noting that deciding subtyping is equivalent to deciding the emptiness of a type.

$$s \leq t \iff s \wedge \neg t = \mathbf{0} \tag{8}$$

which can be derived as follows:

$$\begin{array}{rcl} s \leq t & \iff & [\![s]\!] \subseteq [\![t]\!] \iff [\![s]\!] \cap [\![t]\!]^{\mathsf{c}} = \varnothing \\ & \iff & [\![s \wedge \neg t]\!] = [\![\mathbf{0}]\!] \iff s \wedge \neg t = \mathbf{0}\,. \end{array}$$

Thanks to the semantic interpretation, we can directly apply set-theoretic equivalences to types (in the rest of the paper we will do it without explicitly passing via the interpretation function). We then deduce that every type can be effectively represented in disjunctive normal form, i.e. as the union of intersections of literals, where a literal is a base type or a channel type, possibly negated. Since a union is empty only if all its addenda are empty, then in order to decide emptiness of a type —and thus in virtue of (8) to decide subtyping— it suffices to be able to decide whether an intersection of literals is empty. Since base types and channel types are interpreted in disjoint sets, intersections that involve literals of both kinds are either trivial, or can be simplified to intersections involving literals of only one kind. The problem is therefore reduced to decide whether

$$(\bigwedge_{i \in P} b_i) \wedge (\bigwedge_{j \in N} \neg b_j) \quad \text{and} \quad (\bigwedge_{i \in P} ch^{\nu_i}(t_i)) \wedge (\bigwedge_{j \in N} \neg ch^{\nu_j}(t_j))$$

are equivalent to $\mathbf{0}$ (where ν stands for either $+$ or $-$). The decision of emptiness of the left-hand side depends on the basic types that are used. For what concerns the right-hand side, we decompose this problem into simpler subproblems. More precisely, we reduce this problem to the problem of

deciding subtyping between boolean combinations of the t_i's and t_j's. This problem is simpler, in the sense that it involves a strictly smaller nesting of channel types.

Using set-theoretic manipulations, the problem of deciding

$$(\bigwedge_{i\in P} ch^{\nu_i}(t_i)) \wedge (\bigwedge_{j\in N} \neg ch^{\nu_j}(t_j)) = \mathbf{0}$$

can be shown to be equivalent to

$$(\bigwedge_{i\in P} ch^{\nu_i}(t_i)) \leq (\bigvee_{j\in N} ch^{\nu_j}(t_j)) \,. \tag{9}$$

Because of equations (5) and (6), we can push the intersection on the left-hand side inside the constructors and reduce (9) to the case

$$ch^+(t_1) \wedge ch^-(t_2) \leq \bigvee_{h\in H} ch^+(t_3^h) \vee \bigvee_{k\in K} ch^-(t_4^k) \tag{10}$$

where we grouped covariant and contravariant types together. In this way we simplified the left-hand side. Similarly we can get rid of redundant addenda on the right-hand side of (10) by eliminating:

1. all the covariant channel types on a t_3^h for which there exists a covariant addendum on a smaller or equal $t_3^{h'}$ (since the former channel type is contained in the latter);
2. all contravariant channel type on a t_4^k for which there exists a contravariant addendum on a larger or equal $t_4^{k'}$ (for the same reason as the above);
3. all the covariant channels on a t_3^h that is not larger than or equal to t_2 (since then $ch^-(t_2) \wedge ch^+(t_3^h) = \mathbf{0}$, so it does not change the inequation);
4. all contravariant channel on a t_4^k that is not smaller than or equal to t_1 (since then $ch^+(t_1) \wedge ch^-(t_4^k) = \mathbf{0}$).

Then the key property for decomposing the problem (10) into simpler subproblems is given by the following theorem:

Theorem 2.6 *Suppose $t_1, t_2, t_3^h, t_4^k \in \mathscr{T}$, $k \in K$, $h \in H$. Suppose moreover that the following conditions hold:*

c1. for all distinct $h, h' \in H$, $t_3^h \not\leq t_3^{h'}$;
c2. for all distinct $k, k' \in K$, $t_4^k \not\leq t_4^{k'}$;
c3. for all $h \in H$ $t_2 \leq t_3^h$;
c4. for all $k \in K$ $t_4^k \leq t_1$.

For every $I \subseteq H$ define e_I as $t_1 \wedge \bigwedge_{h\in I} t_3^h \wedge \neg \bigvee_{h\notin I} t_3^h$. Then

$$ch^+(t_1) \wedge ch^-(t_2) \leq \bigvee_{h\in H} ch^+(t_3^h) \vee \bigvee_{k\in K} ch^-(t_4^k)$$

if and only if one of the following conditions holds

LE. $t_2 \not\leq t_1$ or

R1. $\exists h \in H$ such that $t_1 \leq t_3^h$ or

R2. $\exists k \in K$ such that $t_4^k \leq t_2$ or

CA. for every $\mathscr{X} \subseteq \mathscr{P}(H)$ such that $\bigcap \mathscr{X} = \varnothing$, for every choice of atoms $a_I \leq e_I$, $I \in \mathscr{X}$, there is $k \in K$ such that $t_4^k \leq t_2 \vee \bigvee_{I\in\mathscr{X}} a_I$.

The four hypotheses c1–c4 simply state that the right-hand side of the inequation was simplified according to the rules described right before the statement of the theorem. The first condition (LE) says that $ch^+(t_1) \wedge ch^-(t_2)$ is empty. The second condition (R1) and the third condition (R2) respectively make sure that one of the $ch^+(t_3^h)$ and, respectively, one of the $ch^-(t_4^h)$ contains $ch^+(t_1) \wedge ch^-(t_2)$. Finally the fourth and more involved condition (CA) says that, every time we add atoms to t_2 so that we are no longer below any t_3^h then we must end up above some of the t_4^k. The types e_I contain those atoms of t_1 which belong precisely to the t_3^h for $h \in I$. The condition $\bigcap \mathscr{X} = \varnothing$ implies that $t_2 \vee \bigvee_{I\in\mathscr{X}} a_I$ is not below any t_3^h.

As an example of how much our relation is sensitive to atoms, suppose there are three atoms $\mathtt{err}_1, \mathtt{err}_2, \mathtt{exc}$. Consider the case where

$$\begin{array}{lcl} t_2 & = & \mathtt{int}\,; \\ t_1 & = & t_2 \vee \mathtt{err}_1 \vee \mathtt{err}_2 \vee \mathtt{exc}\,; \\ t_3 & = & t_2 \vee \mathtt{exc}\,; \\ t_4 & = & t_2 \vee \mathtt{err}_1 \vee \mathtt{err}_2\,. \end{array}$$

It is easy to see that

$$ch^+(t_1) \wedge ch^-(t_2) \not\leq ch^+(t_3) \vee ch^-(t_4)$$

since, for example, the type $ch(t_2 \vee \mathtt{err}_1)$ is a subtype of the left-hand side, but not of the right-hand side. However if $\mathtt{err}_1 = \mathtt{err}_2$, the subtyping relation holds, because of condition (CA). Indeed in that case the indexing set H of Theorem 2.6 is a singleton. Every $\mathscr{X} \subseteq \mathscr{P}(H)$ such that $\bigcap \mathscr{X} = \varnothing$ contains $\varnothing$. The type $e_\varnothing$ is $t_1 \wedge \neg t_3$. The only atom in it is $\mathtt{err}_1$, and it is true that $t_4 \leq t_2 \vee \mathtt{err}_1$.

As announced, Theorem 2.6 decomposes the subtyping problem of (10) into a finite set of subtyping problems on simpler types (we must simplify the inequation RHS by verifying the inequalities of conditions c1–c4, and possibly perform the $|H| + |K| + 1$ checks for LE, R1 and R1) *and* into the verification of condition (CA).

The condition (CA) involves two universal quantifications. One is on the powerset of a finite set and does not pose problems, but the other is on atoms of a possibly infinite set e_I, and therefore it is not possible to use it for a decision algorithm as it is. This problem can be avoided thanks to the following proposition

Proposition 2.7 *If we replace condition (CA) with*

CA. for every $\mathscr{X} \subseteq \mathscr{P}(H)$ such that $\bigcap \mathscr{X} = \varnothing$, for every choice of atoms $a_I \leq e_I$, $I \in \mathscr{X}$, **e_I finite**, there is $k \in K$ such that $t_4^k \leq t_2 \vee \bigvee_{I\in\mathscr{X}} a_I$.*

then Theorem 2.6 still holds.

Therefore it suffices to check the condition just for the e_I that are finite. This can be done effectively provided that we are able to:

1. decide whether a type is finite;
2. if it is the case, list all its atoms.

We will assume that this is possible for base types. Then it is possible for all types.

Lemma 2.8 *There is an algorithm that decides whether a type t is finite and if it is the case, outputs all its atoms.*

Theorem 2.9 *The subtyping relation is decidable.*

We do not discuss here the complexity of the decision algorithm, nor the possibility of finding more efficient ways of doing it. We leave it for future work. We want to conclude this section by observing that reducing the subtyping problem to deciding type atomicity is not very surprising. On the contrary, it is quite characteristics of a semantic set-theoretic approach. This is much clearer when considering second order polymorphic types. As shown in [**?**], one can end up to solve constraints such as $(t \times X) \leq (t \times \neg t) \vee (X \times t)$, which is true for all type X if and only if t is atomic (this comes from the fact that an atomic type a is semantically characterized by the property that for all types X either $a \leq X$ or $a \leq \neg X$ holds). So once more a subtyping problem is reduced to testing atomicity. The whole point of [**?**] is to avoid such constraints since they are intractable. This is obtained by giving a more syntactic (actually, parametric) interpretation of type variables so as to avoid to interpret (i.e. substitute) them by atomic types (interestingly, models of parametricity intended as genericity are broken by the decidability of finite elements [**?**]). Here instead we showed that with channel types atomicity is decidable and thus the full power of the set-theoretic semantic can be exploited.

3 The $\mathbb{C}\pi$-calculus

3.1 Patterns

As we explained in the introduction, if we want to fully exploit the expressiveness of the type system, we must be able to check the type of the messages read on a channel.

The simplest solution would be to add an explicit typecase process (e.g. $[M : t]P$ which reduces to P or $\mathbf{0}$ depending on whether M is of type t or not). Here, instead, we choose a more general approach, by endowing input actions with $\mathbb{C}$Duce patterns. Pattern matching includes dynamic type checks as a special case, and fits nicely in the semantic subtyping framework.

Definition 3.1 *Given a type algebra $\mathscr{T}$, and a set of variables $\mathbb{V}$, a* pre-pattern *p on $(\mathbb{V}, \mathscr{T})$ is a possibly infinite term p generated by the following grammar*

$$
\begin{array}{rcll}
p & ::= & x & \text{capture, } x \in \mathbb{V} \\
 & | & t & \text{type constraint, } t \in \mathscr{T} \\
 & | & p_1 \wedge p_2 & \text{conjunction} \\
 & | & p_1 | p_2 & \text{alternative}
\end{array}
$$

Given a pre-pattern p on $(\mathbb{V}, \mathscr{T})$ we use $Var(p)$ to denote the set of variables of $\mathbb{V}$ occurring in p (in capture or constant patterns).

Definition 3.2 *Given a type algebra $\mathscr{T}$, and a set of variables $\mathbb{V}$, a pre-pattern p on $(\mathbb{V}, \mathscr{T})$ belongs to the set of (well-formed)* patterns $\mathbb{P}$ *on $(\mathbb{V}, \mathscr{T})$ if and only if it satisfies the following condition: for every subterm $p_1 \wedge p_2$ of p we have $Var(p_1) \cap Var(p_2) = \varnothing$, and for every subterm $p_1 | p_2$ of p we have $Var(p_1) = Var(p_2)$.*

These patterns and their semantics are borrowed from [7]: the reader can refer to [7, 2] for a detailed description. A pattern is matched against an element of the domain $\mathscr{D}$ of a model of the types. A matching returns either a substitution for the free variables of the pattern, or a failure, denoted by Ω:

Definition 3.3 *Given a model $[\![\,]\!] : \mathscr{T} \to \mathscr{D}$, an element $d \in \mathscr{D}$ and a pattern $p \in \mathbb{P}$ the matching of d with p, denoted by d/p, is the element of $\mathscr{D}^{Var(p)} \cup \{\Omega\}$ defined by induction on structure of p as follows:*

$$
\begin{array}{lcll}
d/x & = & \{x \mapsto d\} & \\
d/t & = & \{\} & \text{if } d \in [\![t]\!] \\
 & = & \Omega & \text{otherwise} \\
d/p_1 \wedge p_2 & = & d/p_1 \cup d/p_2 & \text{if } d/p_1, d/p_2 \neq \Omega \\
 & = & \Omega & \text{otherwise} \\
d/p_1 | p_2 & = & d/p_1 & \text{if } d/p_1 \neq \Omega \\
 & = & d/p_2 & \text{otherwise}
\end{array}
$$

In short, a variable pattern always succeeds and captures the matched element with the variable; a type pattern matches only the elements that belong to the interpretation of the type but does not capture them; a conjunction pattern matches only if both patterns match and returns the union of the two substitutions; the alternative pattern tries to match the first pattern and if it fails, it tries the second one.

One remarkable property of the pattern matching above is that the set of all elements for which a pattern p does not fail is the denotation of a type. Since this type is unique, we denote it by $\lbag p \rbag$. In other terms, for every (well-formed) pattern p, there exists a unique type $\lbag p \rbag$ such that $[\![\lbag p \rbag]\!] = \{d \in Dom \mid d/p \neq \Omega\}$. Not only, but this type can be calculated. Similarly, consider a pattern p and a type $t \leq \lbag p \rbag$, then there is also an algorithm that calculates the type environment t/p that associates to each variable x of p the *exact* set of values that x can capture when p is matched against values of type t. Formally

Theorem 3.4 *There is an algorithm mapping every pattern p to a type $\lbag p \rbag$ such that $[\![\lbag p \rbag]\!] = \{d \in \mathscr{D} \mid d/p \neq \Omega\}$.*

Theorem 3.5 *There is an algorithm mapping every pair (t,p), where p is a pattern and t a type such that $t \leq \lfloor p \rfloor$, to a type environment $(t/p) \in \mathscr{T}^{Var(p)}$ such that $[\![(t/p)(x)]\!] = \{(d/p)(x) \mid d \in [\![t]\!]\}$.*

The proofs are similar to those found in [7]. The term (t/p) denotes the environment that assigns to the free variables of p, the types obtained deconstructing t. It is defined as follows.

$$\begin{array}{rcl} t/t' & = & \varnothing \\ t/x & = & x:t \\ t/p_1 \wedge p_2 & = & t/p_1 \uplus t/p_2 \\ t/p_1|p_2 & = & map(\cup)(t/p_1, t/p_2) \end{array}$$

The well definedness of the above is guaranteed by the condition on the free variables of the patterns..

3.2 The language

The syntax of the $\mathbb{C}\pi$-calculus is very similar to that of the asynchronous π-calculus [3, 9], a variant of the π-calculus [12], where message emission is non-blocking. It is generally considered as the calculus representing the essence of name passing with no redundant operation. We deviate from the original calculus by having patterned input prefix and guarded choice between different patterns on the same input channel.

Channels	α	$::=$	x	variable
		$\mid$	c^t	channel constant
Messages	M	$::=$	n	basic constant
		$\mid$	α	channel
Processes	P	$::=$	$\overline{\alpha}M$	output
		$\mid$	$\sum_{i\in I} \alpha(p_i).P_i$	patterned input
		$\mid$	$P_1 \| P_2$	parallel
		$\mid$	$(\nu c^t)P$	restriction
		$\mid$	$!P$	replication

In the above definition, I is a possibly empty finite set of indexes, t ranges over the types defined in Section 2.1 and p_i are patterns as defined in Section 3.1. As customary, we use the convention that the empty sum corresponds to the inert process, usually denoted by 0.

We want to comment on the presence of the simplified form of summation we have adopted: guarded sum of inputs on a single channel with possibly different patterns.

A long standing debate is going on in the concurrency community about the usefulness of summation operators that permit choosing between different continuations. Choice operators are very useful for specifying nondeterministic behaviours, but give rise to problems when considering implementation issues. Two main kinds of choice have to be considered: *external choice* that leaves the decision about the continuation to the external environment (usually having it dependent on the channel used by the environment to communicate) and *internal choice* that is performed by the process regardless of external interactions. External choice is difficult to implement in presence of distribution, (consider modelling $P \parallel Q + R$), thus often only guarded choices are considered; internal choice pops up as soon as two input prefixes use the same channel. Thanks to patterns we can offer an externally controllable choice, where the type of the received message, not the used channel, determines the continuation. Internal choice can also be modelled by specifying processes that perform input on the same channel according to the same pattern.

The other important difference with standard π-calculus is the distinction between channel variables and channel constants. Every channel constant is associated a unique type, which is the type of the messages it can carry (much like Milner's sorting discipline [12]). We make this explicit by decorating channel constant with their associated type. In what follows we will call channel constants also "boxes" to distinguish them from channel variables.

The *values* of the language are the closed messages, that is to say the constants

$$v ::= n \mid c^t \ .$$

We use $\mathscr{V}$ to the denote the set of all values. Every value has a unique atomic type: a basic constant n has an atomic basic type b_n while a channel constant c^t has the channel type $ch(t)$. So all the values can be typed by the rules (const), (chan), and (subs) of Figure 1 (actually with an empty Γ) where in the (subs) subsumption rule $\leq$ is the subtyping relation induced by the model of Section 2.3.

3.3 Semantics

Now consider the interpretation function $[\![\,]\!]_{\mathscr{V}} : \mathscr{T} \to \mathscr{P}(\mathscr{V})$ defined as follows:

$$[\![t]\!]_{\mathscr{V}} = \{v \mid \Gamma \vdash v : t\} \ .$$

It turns out that this interpretation satisfies the model conditions of Section 2.3 and furthermore it generates the same subtyping relation as $\leq$. The circle we mentioned in the Introduction is now closed.

Theorem 3.6 (Model of values)
Let $[\![t]\!]_{\mathscr{V}} = \{v \mid \Gamma \vdash v : t\}$. Then $(\mathscr{V}, [\![\,]\!]_{\mathscr{V}})$ is a model and $s \leq t \iff [\![s]\!]_{\mathscr{V}} \subseteq [\![t]\!]_{\mathscr{V}}$.

Since values are elements of a model of the types, Definition 3.3 applies for d being a value. We can thus use it to define the reduction semantics of our calculus:

$$\overline{c^t} v \parallel \sum_{i\in I} c^t(p_i).P_i \quad \longrightarrow \quad P_j[v/p_j]$$

$$\mathscr{R}[] \;::=\; [] \;\mid\; \mathscr{R}[]\|P \;\mid\; P\|\mathscr{R}[] \;\mid\; (\nu c^t)\mathscr{R}[]$$

$$P \longrightarrow Q \;\Rightarrow\; \mathscr{R}[P] \longrightarrow \mathscr{R}[Q]$$

$$P' \equiv P \longrightarrow Q \;\Rightarrow\; P' \longrightarrow Q$$

$$P\|\mathbf{0} \equiv P \qquad P\|Q \equiv Q\|P \qquad P\|(Q\|R) \equiv (P\|Q)\|R$$
$$(\nu c^t)\mathbf{0} \equiv \mathbf{0} \qquad (\nu c^t)P \equiv (\nu d^t)P\{c^t \rightsquigarrow d^t\} \qquad !P \equiv !P\|P$$
$$(\nu c_1^{t_1})(\nu c_2^{t_2})P \equiv (\nu c_2^{t_2})(\nu c_1^{t_1})P \qquad \text{for } c_1 \neq c_2$$

$$(\nu c^t)(P\|Q) \equiv P\|(\nu c^t)Q \qquad \text{for } c^t \notin \mathsf{fn}(P)$$

where $P\{c^t \rightsquigarrow d^t\}$ is obtained from P by renaming all free occurrences of the box c^t into d^t, and assumes d^t is fresh.

Figure 2. Context and congruence closure

where $P[\sigma]$ denotes the application of substitution σ to process P. The output of a *value* on the box c^t synchronises with an input on the same box only if at least one of the patterns guarding the sum matches the communicated value. If more than one pattern matches, then one of them is nondeterministically chosen and the corresponding process executed, but before its execution the pattern variables are replaced by the captured values. More refined matching policies (best match, first match) can be easily encoded.

As usual the notion of reduction must be completed with reductions in evaluation contexts and up to structural congruence, whose definitions are summarised in Figure 2.

This operational semantics is the same as that of π-calculus but the actual process behavior has been refined in two points:

- communication is subjected to pattern matching
- communication can happen only along values (boxes)

First of all note that these two points are not restrictive. Every asynchronous π-calculus process is also a process of our calculus and with the same reduction semantics: it suffices to consider all free and restricted variables (thus excluding those that are bound in an input actions, which according to our viewpoint are "real" variables) to be typed channels of some channel type (as we do not consider well-typing issues, yet). So we do not lose any generality with respect to the π-calculus. The use of pattern matching is what makes it necessary to distinguish between typed channels and variables: matching is defined only for the former as they are values, while a matching on variables must be delayed until they will be bound to a value.

Since we distinguish between variables and typed channels, it is reasonable to require that communication takes place only if we have a physical channel that can be used as a support for it; thus, we forbid synchronisation if the channel is still a variable. However there is a more technical reason to require this. Consider an environment $\Gamma = x : \mathbf{0}$. By subsumption we have $\Gamma \vdash x : ch(\mathtt{int})$ and $\Gamma \vdash x : ch^-(\mathtt{string})$. Then, according to the typing rules of our system (see later on) the process $\overline{x}\,\text{“}\mathtt{ciao}\text{”} \,\|\, x(y).\overline{x}(y{+}1)$ is well typed, in the environment Γ, but it would give rise to a run time error by attempting to increase the string "ciao" by 1:

$$\overline{x}\,\text{“}\mathtt{ciao}\text{”} \,\|\, x(y).\overline{x}(y{+}1) \quad\longrightarrow\quad \overline{x}(\text{“}\mathtt{ciao}\text{”}{+}1)$$

This reduction cannot happen in our calculus, because we can never instantiate a variable of type $\mathbf{0}$ (from a logical viewpoint, this corresponds to the classical *ex falsum quodlibet* deduction rule).

3.4 Typing

In Figure 1, we summarise typing rules that guarantee that, in well typed processes, channels communicate only values that correspond to their type.

The rules for messages do not deserve any particular comment. As customary, the system deduces only the well-formedness of processes without assigning them any types. Rules for replication and parallel composition are standard. The rule for restriction is slightly different from the usual one since we do not need to store in the type environment the type of the channel[3]. In the rule for output we check that the message is compatible with the type of the channel.

The rule for input is the most involved one. The premises of the rule first infer the type t of the message that can be transmitted over the channel α, then for each summand i they use this type to calculate the type environment of the pattern variables (the environment $((t \wedge \llparenthesis p_i \rrparenthesis)/p_i)$ of Theorem 3.5) and check whether under this environment the summand process P_i is typeable. For instance in the polyadic version (§ 4.1) if $\alpha : ch^+(s \times t)$ then in order to type $\alpha(\, x, y \wedge (\mathtt{int}|\mathtt{bool})\,)P$ the (input) rule verifies the type of P under the environment $x : s\,,\; y : t \wedge (\mathtt{int} \vee \mathtt{bool})$.

This is all it is needed to have a sound type system. However the input construct is like a typecase/matching expression, so it seems reasonable to perform a check that patterns are exhaustive and there is no useless case[4]. This is precisely what the two side conditions of (input) do:

$(t \leq \bigvee_{i\in I} \llparenthesis p_i \rrparenthesis)$ checks whether pattern matching is exhaustive, that is if for whatever value (of type t) sent on α there exists at least one pattern p_i that will accept it (the cases cover all possibilities).

$(\llparenthesis p_i \rrparenthesis \wedge t \neq \mathbf{0})$ checks that pattern matching is not redundant that is that there does not exists a pattern p_i that will fail with every value of type t (no case is useless).

[3] Strictly speaking, we do not restrict variables but constants, so it would be formally wrong to store it in Γ. For the same reason, α-conversion is handled as a structural equivalence rule.

[4] In functional programming these checks are necessary for soundness since an expression non-complying to them may yield a type-error. In process algebræ non-compliance would just block synchronisation.

Messages

$$\frac{}{\Gamma \vdash n : b_n}\ (\text{const}) \qquad \frac{}{\Gamma \vdash c^t : ch(t)}\ (\text{chan}) \qquad \frac{}{\Gamma \vdash x : \Gamma(x)}\ (\text{var}) \qquad \frac{\Gamma \vdash M : s \leq t}{\Gamma \vdash M : t}\ (\text{subs})$$

Processes

$$\frac{\Gamma \vdash P}{\Gamma \vdash (\nu c^t)P}\ (\text{new}) \qquad \frac{\Gamma \vdash P}{\Gamma \vdash !P}\ (\text{repl}) \qquad \frac{\Gamma \vdash P_1 \quad \Gamma \vdash P_2}{\Gamma \vdash P_1 \| P_2}\ (\text{para})$$

$$\frac{t \leq \bigvee_{i\in I} \lfloor p_i \rfloor \quad t \wedge \lfloor p_i \rfloor \neq \mathbf{0} \quad \Gamma \vdash \alpha : ch^+(t) \quad \Gamma, (t \wedge \lfloor p_i \rfloor)/p_i \vdash P_i}{\Gamma \vdash \sum_{i\in I} \alpha(p_i).P_i}\ (\text{input}) \qquad \frac{\Gamma \vdash M : t \quad \Gamma \vdash \alpha : ch^-(t)}{\Gamma \vdash \overline{\alpha} M}\ (\text{output})$$

Figure 1. Typing rules

As usual the basic result is the subject reduction, preceded by a substitution lemma. The proof of the theorem relies on the semantics of channel types as set of boxes.

Lemma 3.7 (Substitution)
- *If* $\Gamma, t/p \vdash M' : t'$ *and* $\Gamma \vdash v : t$, *then* $\Gamma \vdash M'[v/p] : t'$.
- *If* $\Gamma, t/p \vdash P$ *and* $\Gamma \vdash v : t$ *then* $\Gamma \vdash P[v/p]$.

Lemma 3.8 (Congruence)
If $\Gamma \vdash P$ *and* $P \equiv Q$, *then* $\Gamma \vdash Q$.

Theorem 3.9 (Subject reduction)
If $\Gamma \vdash P$ *and* $P \to P'$ *then* $\Gamma \vdash P'$.

The decidability of the subtyping relation does not directly imply decidability of the typing relation (only semi-decidability is straightforward). In similar situations, a typing algorithm can be often derived by eliminating the subsumption rule and embedding the subtyping checks into the elimination rules. However the (input) rule, in its algorithmic version, requires computing the least type of the form $ch^+(s)$ which is above a given type t, and it is not so evident that such a type exists (observe that our type algebra is *not* a complete lattice). Nevertheless, it turns out that such a type does exist (which gives us the minimum typing property) and furthermore it can be effectively computed.

Lemma 3.10 (Upper bound channel) *For every type* $t \leq ch^+(\mathbf{1})$ *there exists a least type* $ch^+(s)$ *that is an upper bound of* t *and an algorithm that computes it.*

Proof:(hint) Consider $t \wedge \neg ch^+(s)$. We have to find the least s such that this intersection is empty. Put the intersection in the form of unions of terms like (10). Now for each addendum take the corresponding t_1 (we refer to the form in (10)) and if $\neg ch(s)$ occurs in the addendum, use Theorem 2.6 (actually Proposition 2.7) to take away from t_1 the maximum number of atoms such that the intersection is still empty. The wanted s is the union of all these types.

Theorem 3.11 *The typing relation is decidable.*

3.5 An example

We present here an example of a $\mathbb{C}\pi$ process. Consider the following situation. A web server is waiting on some channel α. The client wants the server to perform some computation on some values it will send to the server. The server is able to perform two different kinds of computation, on values of type t_1 (say arithmetic operations), or on values of type t_2 (say list sorting). At the beginning of each session, the client can decide which operation it wants the server to perform, by sending a channel to the server, along which the communication can happen. The server checks the type of the channel, and provides the corresponding service.

$$P := \alpha(x \wedge ch^+(t_1)).!x(y).P_1 + \alpha(x \wedge ch^+(t_2)).!x(y).P_2$$

In the above process the channel α has type $ch^+(ch^+(t_1) \vee ch^+(t_2))$. Note that, $ch^+(t_1) \vee ch^+(t_2) \neq ch^+(t_1 \vee t_2)$. This means that the channel the server received on α will communicate *either* always values of type t_1 *or* always values of type t_2, and not interleaving sequences of the two as would do $ch^+(t_1 \vee t_2)$.

As we discussed in the Introduction, this distinction would not be present if the equation $ch^+(t_1) \vee ch^+(t_2) = ch^+(t_1 \vee t_2)$ held true. In that case we would need to write P as

$$P' := \alpha(x).!(x(y \wedge t_1).P_1 + x(y \wedge t_2).P_2)$$

which is a less efficient server, as it performs pattern matching every time it receives a value.

4 Extensions

4.1 Polyadic version

The first extension we propose consists in adding product to our type constructors. This requires extending the notion of pattern, but, most importantly, it affects the definition of subtyping. The new syntax for types is the following

$$\begin{array}{rrcl} \textit{Types} & t & ::= & b \mid ch^+(t) \mid ch^-(t) \mid t \times t \\ & & \mid & \mathbf{0} \mid \mathbf{1} \mid \neg t \mid t \vee t \mid t \wedge t \end{array}$$

Messages are extended by

$$\textit{Messages} \quad M \quad ::= \quad \ldots \quad | \quad (M,M) \quad \text{pair}$$

The patterns are extended by

$$\textit{Patterns} \quad p \quad ::= \quad \ldots \quad | \quad (p_1,p_2) \quad \text{pair}$$

with the condition that the for every subterm (p_1,p_2) of p we have $Var(p_1) \cap Var(p_2) = \varnothing$.

A semantic model can be built, in analogy with Section 2.2. The corresponding subtyping relation is also decidable, as well as the typing relation.

Note that besides the extension above we do not need to add anything else since for instance projections can be encoded by pattern matching. By using product types, together with the recursive types we show next, we can also encode more structured data, like lists or XML documents.

4.2 Recursive types

Another important addition to our type systems is that of recursive types. This is for example necessary to define the type of lists.

So far, types could be represented as finite labelled trees. Recursive types are obtained by allowing infinite trees, without changing the syntax. As in the type system of $\mathbb{C}$Duce we require such trees to be regular and with the property that every infinite branch contains infinitely many nodes labelled by the product constructor.

Moreover we require that every branch can contain only finitely many nodes labelled with a channel constructor. If we were to define recursive types with equations, this would amount to forbidding the recursive variable being defined to be used inside a channel constructor (such as $x = ch(x) \vee \mathtt{int}$). However a recursive type can appear inside a channel constructor provided that the number of occurrences of channel constructors is finite (such as in $ch(\mathit{intlist})$ where $\mathit{intlist} = (\mathtt{int} \times \mathit{intlist}) \vee ch(\mathbf{0})$).

The reason for this is that, without this restriction, there is no model. To see why, we observe that we could have a recursive type t such that

$$t = b \vee (ch(t) \wedge ch(b))$$

for some nonempty base type b. If we have a model, either $t = b$ or $t \neq b$. Suppose $t = b$, then $ch(t) \wedge ch(b) = ch(b)$ and $b = t = b \vee ch(b)$. The latter implies $ch(b) \leq b$ which is not true when b is a base type. Therefore it must be $t \neq b$. According to our semantics this implies $ch(t) \wedge ch(b) = \mathbf{0}$, because they are two distinct atoms. Thus $t = b \vee \mathbf{0} = b$, contradiction.

Types are therefore stratified according to how many nestings of the channel constructor there are and this stratification allows us to construct the model using the same ideas presented in Section 2.

One traditional example of the use of a recursive type is "self application", that is a channel that can carry itself. In our type system, we can still type self application by using, for instance, the type $ch(\mathbf{1})$: a channel that can carry everything, can clearly carry itself.

4.3 Local calculus

This restriction on recursive types can removed, by moving to a *local* version of the calculus [10], where only the output capability of a channel can be communicated. This makes the type $ch^+(t)$ useless. Without this type, the example of Section 4.2 cannot be constructed, and indeed it is possible construct a model of the types with full recursion.

The absence of input channel types makes also the decision algorithm considerably simpler, as condition (CA) is invoked only when channel types of both polarities are present. In particular the subtyping of channel types can be reduced to the following condition: $ch^-(t) \leq \bigvee_{i \in I} ch^-(t_i)$ if and only if there exists $i \in I$ such that $t_i \leq t$.

The details of this construction are part of ongoing work. We preferred to deal here with the full type system, for several reasons. First of all, we wanted to study the most general calculus – we chose the asynchronous version for the sake of exposition, but that is clearly not restrictive from the point of view of types. Secondly, without input channel types, the model of the types is very similar to the one of $\mathbb{C}$Duce, while the model we present here is new. Finally we believe the complex subtyping algorithm and the paradox on recursion to be interesting results on their own.

5 Conclusion

We have presented a novel approach to defining subtyping relations for the π-calculus, and discussed its merits and limitations. To exploit our type system, we have defined a variant of the π-calculus with pattern matching on input. We would like to be able to give a type respecting encoding of $\mathbb{C}$Duce into $\mathbb{C}\pi$, similar to the Milner-Turner encoding of the simply typed λ-calculus in π [12]. However, the standard translation of arrow types into channel types does not respect equality, and a more subtle approach is needed.

Acknowledgements. We are very grateful to Alain Frisch and Mariangiola Dezani for interesting discussions. We acknowledge the support of the European FET contracts *MyThS*, IST-2001-32617 and *Mikado*, IST-2001-32222, of the ACI *Masses de données* "Transformation languages for XML: logic and applications", of the EPSRC grant GR/T04724/01 and of ENS for a visiting professorship grant for Rocco.

References

[1] L. Acciai and M. Boreale. XPi: a typed process calculus for XML messaging. Unpublished manuscript, 2004.

[2] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-friendly general purpose language. In *ICFP '03, 8th ACM International Conference on Functional Programming*, pages 51–63, Uppsala, Sweden, 2003. ACM Press.

[3] G. Boudol. Asynchrony and the π-calculus. Research Report 1702, INRIA, http://www.inria.fr/rrrt/rr-1702.html. Also available from http://www-sop.inria.fr/mimosa/personnel/Gerard.Boudol.html, 1992.

[4] A. Brown, C. Laneve, and G. Meredith. πduce: a process calculus with native XML datatypes. Unpublished, 2004.

[5] R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE Transaction on Software Engineering*, 24(5):315–330, 1998.

[6] Alain Frisch. *Théorie, conception et réalisation d'un langage de programmation fonctionnel adapté à XML*. PhD thesis, Université Paris 7, December 2004.

[7] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic Subtyping. In *Proceedings, Seventeenth Annual IEEE Symposium on Logic in Computer Science*, pages 137–146. IEEE Computer Society Press, 2002.

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

[9] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *Proc. ECOOP 91*, volume 512 of *LNCS*, pages 133–147. Springer, 1991.

[10] Massimo Merro. *Locality in the pi-calculus and applications to distributed objects*. PhD thesis, Ecole des Mines de Paris, Nice, France, 2000.

[11] B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5), 1996.

[12] D. Sangiorgi and D. Walker. *The π-calculus*. Cambridge University Press, 2002.

[13] P. Sewell. Global/local subtyping and capability inference for a distributed π-calculus. In *Proceedings of 25th ICALP*, volume 1443 of *LNCS*, pages 695–706, 1998.

[14] N. Yoshida and M. Hennessy. Subtyping and locality in distributed higher order processes. In *Proceedings of 10th CONCUR*, volume 1664 of *LNCS*, pages 557–572, 1999.

A Type algorithm

The type algorithm is obtained from the typing rules in a standard way, namely by deleting the subsumption rule and embedding the checking of the subtyping relation in the elimination rules. This requires the use of the least type t such that $t = ch^+(t')$ for some t', and $s \leq t$. Such a t is denoted by $\mathscr{C}(s)$. The existence and the decidability of $\mathscr{C}(s)$ is given by Lemma 3.10, the decidability of (t/p) is given by Theorem 3.5. The algorithmic rules are summarised in Figure 3.

B Proofs

B.1 Characterising inclusion (Theorem 2.6 and Proposition 2.7)

In this section we first prove Theorem 2.6 and then strengthen the result as in Proposition 2.7.

We recall that in a boolean algebra, an *atom* is a minimal nonzero element. A boolean algebra is *atomic* if every nonzero element is greater or equal than an atom. It is easy to prove that an atomic boolean algebra is equivalent to a subset of the powerset of its atoms.

Let $(D, \wedge, \vee, \mathbf{0}, \mathbf{1})$ be an atomic boolean algebra where, as customary, $d' \leq d$ if and only if $d' \vee d = d$. For every $d \in D$ we denote $\downarrow d$ (that is, the set of all elements smaller than or equal to d) as $ch^+(d)$ and $\uparrow d$ (that is, the set of all elements larger than or equal to d) as $ch^-(d)$. We want to give an equivalent characterisation of the equation

$$\bigcap_{i\in I} ch^+(d_1^i) \cap \bigcap_{j\in J} ch^-(d_2^j) \subseteq \bigcup_{h\in H} ch^+(d_3^h) \cup \bigcup_{k\in K} ch^-(d_4^k)$$

that does not use the "operators" $ch^+(), ch^-()$. Notice that

$$\bigcap_{i\in I} ch^+(d_1^i) = ch^+(\bigwedge_{i\in I} d_1^i)$$

and

$$\bigcap_{j\in J} ch^-(d_2^j) = ch^-(\bigvee_{j\in J} d_2^j)\ .$$

Also if there exist h, h' such that $d_3^{h'} \leq d_3^h$ we can ignore $d_3^{h'}$ as $ch^+(d_3^{h'}) \subseteq ch^+(d_3^h)$. Dually for the d_4^k. Therefore we can concentrate on the case

$$ch^+(d_1) \cap ch^-(d_2) \subseteq \bigcup_{h\in H} ch^+(d_3^h) \cup \bigcup_{k\in K} ch^-(d_4^k)$$

where no two d_3^h are comparable, and no d_4^k are comparable.

The first case in which the inclusion holds is when $ch^+(d_1) \cap ch^-(d_2) = \varnothing$, which happens exactly when $d_2 \not\leq d_1$. If $d_2 \leq d_1$, without loss of generality we can also assume that $d_3^h \geq d_2$ for all $h \in H$ and that $d_4^k \leq d_1$ for all $k \in K$. This is because if $d_3^{\bar{h}} \not\geq d_2$ for some $\bar{h}$ then no element of $ch^-(d_2)$

Messages

$$\frac{}{\Gamma \vdash n : b_n}\ \text{(const)} \qquad \frac{}{\Gamma \vdash c^t : ch(t)}\ \text{(chan)} \qquad \frac{}{\Gamma \vdash x : \Gamma(x)}\ \text{(var)}$$

Processes

$$\frac{\Gamma \vdash P}{\Gamma \vdash (\nu c^t)P}\ \text{(new)} \qquad \frac{\Gamma \vdash P}{\Gamma \vdash !P}\ \text{(repl)} \qquad \frac{\Gamma \vdash P_1 \quad \Gamma \vdash P_2}{\Gamma \vdash P_1 \| P_2}\ \text{(para)}$$

$$\frac{\Gamma \vdash \alpha : s \quad \mathscr{C}(s) = ch^+(t) \quad \Gamma, t/p_i \vdash P_i}{\Gamma \vdash \sum_{i\in I} \alpha(p_i).P_i}\ \text{(input)} \qquad \frac{\Gamma \vdash M : t \quad \Gamma \vdash \alpha : s \quad s \leq ch^-(t)}{\Gamma \vdash \overline{\alpha} M}\ \text{(output)}$$

$t \leq \bigvee_{i\in I} \lfloor p_i \rfloor \quad \lfloor p_i \rfloor \wedge t \neq \mathbf{0}$

Figure 3. Algorithmic rules

can be in $ch^+(d_3^{\bar{h}})$. We can thus ignore such sets to test for the inclusion, and similarly for the d_4^k's.

The inclusion surely holds if for some $\bar{h}$ we have $d_1 \leq d_3^{\bar{h}}$, or if for some $\bar{k}$ we have $d_2 \geq d_4^{\bar{k}}$, since then, for instance in the former case, $ch^+(d_1)$ is contained in $ch^+(d_3^{\bar{h}})$ and so is its intersection with $ch^-(d_2)$.

The most difficult case occurs when

- $d_2 \leq d_1$;
- for all $h \in H$, $d_3^h \geq d_2$;
- for all $k \in K$, $d_4^k \leq d_1$;
- for all $h \in H$, $d_3^h \not\geq d_1$;
- for all $k \in K$, $d_4^k \not\leq d_2$.

The way of thinking the inclusion is the following. (From now on it will be easier to think of D as a subset of the powerset of its atoms; therefore we will sometimes say "contained" rather than "smaller", and so on.) Consider a d in $ch^+(d_1) \cap ch^-(d_2)$. If d is not below any of the d_3^h then it must be above one of the d_4^k. Suppose there is an element x of d_1 which is in no d_3^h (more precisely, suppose that there is an atom $\overline{d}$ such that $\overline{d} \leq d_1$ and for all h, $\overline{d} \not\leq d_3^h$; to stress that it is an atom denote $\overline{d}$ by $\{x\}$). Then $d_2 \vee \{x\}$ is not contained in any of the d_3^h, and it must contain one of the d_4^k. This implies that for such d_4^k, $d_4^k \setminus d_2 \leq \{x\}$[5]. Consider now two elements x_1, x_2 in d_1 such that if x_1 belongs to d_3^h then x_2 does not belong to d_3^h. Then $d_2 \vee \{x_1, x_2\}$ is not contained in any of the d_3^h, and it must contain one of the d_4^k. This implies that for such d_4^k, $d_4^k \setminus d_2 \leq \{x_1, x_2\}$.

More generally: for every $I \subseteq H$ consider the the set e_I defined as $d_1 \wedge \bigwedge_{h\in I} d_3^h \setminus \bigvee_{h\notin I} d_3^h$. The set e_I contains those elements of d_1 which belong precisely to the d_3^h for $h \in I$. Because all d_3^h are incomparable, the e_I are nonempty and pairwise disjoint. Consider a subset $\mathscr{X}$ of $\mathscr{P}(H)$ satisfying the property $\bigcap \mathscr{X} = \varnothing$. For every $I \in \mathscr{X}$, choose an element x_I in e_I. We have that $d_2 \vee \{x_I \mid I \in \mathscr{X}\}$ is not contained in any of the d_3^h. Reasoning as above we then have that there is a d_4^k such that $d_4^k \setminus d_2 \leq \{x_I \mid I \in \mathscr{X}\}$.

[5] it is in fact equal as $d_4^k \not\leq d_2$.

This proves the necessity of the condition (CA): for every $\mathscr{X}$ such that $\bigcap \mathscr{X} = \varnothing$, for every choice of $x_I \in e_I$, $I \in \mathscr{X}$ there must be a d_4^k such that $d_4^k \setminus d_2 \leq \{x_I \mid I \in \mathscr{X}\}$.

We argued that the condition (CA) is necessary. It is also sufficient: if the condition holds, every set d included in d_1, containing d_2, and which is not contained in any of the d_3^h, must contain a set of the form $d_2 \vee \{x_I \mid I \in \mathscr{X}\}$: just pick one witness of noncontainment for every d_3^h. Thus d contains one of the d_4^k.

We can strengthen the result as stated in Proposition 2.7. Consider the case where some of the e_I are infinite. Since there are only finitely many d_4^k, the condition is satisfied if and only if for at least two (in fact infinitely many) different choices x_I and x'_I we have that the same d_4^k satisfies $d_4^k \setminus d_2 \leq \{x_I \mid I \in \mathscr{X}\}$, and $d_4^k \setminus d_2 \leq \{x'_I \mid I \in \mathscr{X}\}$. Therefore we must have $d_4^k \setminus d_2 \subseteq \{x_I \mid I \in \mathscr{X} \ \&\ e_I \text{ finite}\}$. (We could improve further by considering only those e_I whose cardinality is not greater than the number of d_4^k - we do not need this for our purposes.)

This proves that condition (CA) is equivalent to condition (CA*): for every $\mathscr{X}$ such that $\bigcap \mathscr{X} = \varnothing$, for every choice of $x_I \in e_I$, $I \in \mathscr{X}$, e_I finite, there must be a d_4^k such that $d_4^k \setminus d_2 \leq \{x_I \mid I \in \mathscr{X}\}$.

B.2 The existence of a model

We shall construct here a model for the simplest of our type systems. This amounts to build a pre-model and then show that it satisfies Definition 2.3.

Types are stratified according to the height of the nesting of the channel constructor. We define the height function $\hbar(t)$ as follows:

- $\hbar(b) = \hbar(\mathbf{0}) = \hbar(\mathbf{1}) = 0$;
- $\hbar(ch(t)) = \hbar(ch^+(t)) = \hbar(ch^-(t)) = \hbar(t) + 1$;
- $\hbar(t_1 \vee t_2) = \hbar(t_1 \wedge t_2) = \max(\hbar(t_1), \hbar(t_2))$;
- $\hbar(\neg t) = \hbar(t)$.

Then we set

$$\mathscr{T}_n \stackrel{def}{=} \{t \mid \hbar(t) \leq n\}\ .$$

Our pre-model for the types is built in steps. We start by providing a model for types of height 0, that is types in $\mathscr{T}_0$. Note that we must define the semantics only for type

constructors, because the interpretation of the combinators is determined by the definition of pre-model. The only constructors of height 0 are the basic types, for these we assume existence of a universe of interpretation $\mathbb{B}$. We also assume that every basic type b has an interpretation $\mathscr{B}[\![b]\!] \subseteq \mathbb{B}$. Therefore we set $\mathscr{D}_0 = \mathbb{B}$, with the semantics defined by $[\![b]\!]_0 = \mathscr{B}[\![b]\!]$ and interpret boolean combinators by using the corresponding set-theoretic combinators, according to Definition 2.1. Using this pre-model we define a subtyping relation over $\mathscr{T}_0$ as $t \leq_0 t'$ if and only if $[\![t]\!]_0 \subseteq [\![t']\!]_0$. We shall denote by $=_0$ the corresponding equivalence.

Now suppose we have a pre-model $\mathscr{D}_n$ for $\mathscr{T}_n$, with corresponding preorder $\leq_n$ and equivalence $=_n$. We call $\widetilde{\mathscr{T}_n}$ the set of equivalence classes $\mathscr{T}_n/=_n$. Then, $\mathscr{D}_{n+1}$ is defined as follows:

$$\mathscr{D}_{n+1} = \mathbb{B} + \widetilde{\mathscr{T}_n} .$$

with the following interpretation of channel types:

- $[\![ch^+(t)]\!]_{n+1} = \{[t']_{=_n} \mid t' \leq_n t\}$;
- $[\![ch^-(t)]\!]_{n+1} = \{[t']_{=_n} \mid t \leq_n t'\}$.

In principle each of these pre-models defines a different preorder between types. However, all such preorders coincide in the following sense:

Proposition B.1 *Let $t,t' \in \mathscr{T}_n$ and $k,h \geq n$, then $t \leq_k t'$ if and only if $t \leq_h t'$.*

Proof. To carry out the proof we use an interesting fact: every singleton of our pre-models is denoted by some type. (Assuming this is true for base types, which we can safely assume.)

We also need a technicality: we add to our types of height 0 the types $\mathbb{k}$ for all positive natural number k: they are used at level 0 as a witness of channel types. At level 0 we only know that there are infinitely many different channel types. The pre-model at level 0 is exactly formed by the basic types plus the positive natural numbers to modelling the $\mathbb{k}$.

Therefore $\mathscr{D}_0 := \mathbb{B} + \mathbb{N}^+$ with

$$[\![\mathbb{k}]\!]_0 = \{k\} .$$

Now suppose we have a model $\mathscr{D}_n$ for $\mathscr{T}_n$, with corresponding preorder $\leq_n$ and equivalence $=_n$. We call $\widetilde{\mathscr{T}_n}$ the set of equivalence classes $T_n/=_n$. Then we set

$$\mathscr{D}_{n+1} = \mathbb{B} + \widetilde{\mathscr{T}_n}$$

with the semantics of the channel types being

$$\begin{aligned} [\![ch^+(t)]\!]_{n+1} &= \{[t']_{=_n} \mid t' \leq_n t\}; \\ [\![ch^-(t)]\!]_{n+1} &= \{[t']_{=_n} \mid t \leq_n t'\}; \\ [\![\mathbb{k} + \mathbb{1}]\!]_{n+1} &= \{[\mathbb{k}]_{=_n}\}. \end{aligned}$$

Note that the semantics of $\mathbb{1}$ coincides with the semantics of $ch(\mathbf{0})$, and in general the semantics of $\mathbb{k} + \mathbb{1}$ coincides with the semantics of $ch(\mathbb{k})$. Therefore in the semantics at levels greater than 0 we can substitute $\mathbb{k}$ with the appropriate channel type.

When is a type t empty? Given a type t we put it in disjunctive normal form. Clearly t is empty if and only if all summands are empty. If a summand contains literals of both basic types and channel types it is easy to decide emptiness: if it contains two positive literals of different kinds, then it is empty. If the positive literals are all of one kind, it is empty if and only if it is empty when removing the negative literals of the other kind. Finally the intersection of only negative literals is empty if the two kinds separately cover their own universe of interpretation. (That is if the union of all negated basic types is $\mathbb{B}$ and similarly for the channel types.)

Therefore it is enough to check emptiness for intersections of literals of one kind only. For base types:

$$\bigwedge_{b \in P} b \wedge \bigwedge_{b \in N} \neg b .$$

For channel types:

$$\bigwedge_{i \in I} ch^+(t_1^i) \wedge \bigwedge_{j \in J} ch^-(t_2^j) \wedge \bigwedge_{h \in H} \neg ch^+(t_3^h) \wedge \bigwedge_{k \in K} \neg ch^-(t_4^k) .$$

Using equations (5) and (6) of Section 2 we can simplify the above expression to

$$ch^+(t_1) \wedge ch^-(t_2) \wedge \bigwedge_{h \in H} \neg ch^+(t_3^h) \wedge \bigwedge_{k \in K} \neg ch^-(t_4^k) .$$

To prove Proposition B.1, we now prove by induction the following statement: let $t \in \mathscr{T}_n$, then

- $t =_n \mathbf{0}$ if and only if $t =_{n+1} \mathbf{0}$;
- $|t|_n = h$ if and only if $|t|_n = h$;

where $|t|$ denotes the cardinality of t.

We start by the case $n = 0$. The "algorithm" for checking emptiness works in the same way for basic types. The only difference occurs for the types $\mathbb{k}$. The condition to check at level 0 is the following

$$\mathbb{N} \cap \bigcap_{\mathbb{k} \in P} [\![\mathbb{k}]\!]_0 \subseteq \bigcup_{\mathbb{k} \in N} [\![\mathbb{k}]\!]_0$$

which can be true only if there are two different $\mathbb{k} \in P$ or if the only $\mathbb{k}$ in P is also in N. It is important here that $\mathbb{N}$ is infinite, so no finite union of singletons can cover it. Therefore the condition above is equivalent to

$$\widetilde{\mathscr{T}_0} \cap \bigcap_{\mathbb{k} \in P} [\![\mathbb{k}]\!]_1 \subseteq \bigcup_{\mathbb{k} \in N} [\![\mathbb{k}]\!]_1$$

and therefore $t =_0 \mathbf{0}$ if and only if $t =_1 \mathbf{0}$. As for the cardinality: the proof is more general and it is the same as the inductive step case that we will show next.

For the inductive step suppose that we know that for every type $t \in \mathscr{T}_n$ we have

- $t =_n \mathbf{0}$ if and only if $t =_{n+1} \mathbf{0}$;
- $|t|_n = h$ if and only if $|t|_{n+1} = h$.

Now take a type $t \in \mathscr{T}_{n+1}$, we want to prove that

- $t =_{n+1} \mathbf{0}$ if and only if $t =_{n+2} \mathbf{0}$;
- $|t|_{n+1} = h$ if and only if $|t|_{n+2} = h$.

Again the "algorithm" for checking the emptiness of basic types does not change. In the case of channel types we have to check that

$$[\![ch^+(t_1)]\!]_{n+1} \cap [\![ch^-(t_2)]\!]_{n+1}$$

$$\subseteq \bigcup_{h\in H} [\![ch^+(t_3^h)]\!]_{n+1} \cup \bigcup_{k\in K} [\![ch^-(t_4^k)]\!]_{n+1}$$

if and only if

$$[\![ch^+(t_1)]\!]_{n+2} \cap [\![ch^-(t_2)]\!]_{n+2}$$

$$\subseteq \bigcup_{h\in H} [\![ch^+(t_3^h)]\!]_{n+2} \cup \bigcup_{k\in K} [\![ch^-(t_4^k)]\!]_{n+2} .$$

As argued in the previous section, the first condition is equivalent to:

LE. $t_2 \not\leq_n t_1$ or

R1. $\exists h \in H$ such that $t_1 \leq_n t_3^h$ or

R2. $\exists k \in K$ such that $t_4^k \leq_n t_2$ or

CA$*$ the complicated condition involving $\leq_n$ and atoms.

The induction hypothesis gives us easily the equivalence of the first three conditions at levels n and $n+1$. For the condition (CA$*$) note first that

- $t_2 \leq_n t_1$
- for all $h \in H$, $d_3^h \geq_n d_2$
- for all $k \in K$, $d_4^k \leq_n d_1$
- for all $h \in H$, $d_3^h \not\geq_n d_1$
- for all $k \in K$, $d_4^k \not\leq_n d_2$

are equivalent to

- $t_2 \leq_{n+1} d_1$
- for all $h \in H$, $d_3^h \geq_{n+1} d_2$
- for all $k \in K$, $d_4^k \leq_{n+1} d_1$
- for all $h \in H$, $d_3^h \not\geq_{n+1} d_1$
- for all $k \in K$, $d_4^k \not\leq_{n+1} d_2$

because of the induction hypothesis. For every $I \subseteq H$ define t_I as

$$t_1 \wedge \bigwedge_{h\in I} t_3^h \wedge \neg \bigvee_{h\notin I} t_3^h .$$

we have to check that the condition (CA$*$):

> for every $\mathscr{X}$, for every $a_I \in Atom_n$, $a_I \leq_n t_I$, $I \in \mathscr{X}$, $|t_I|_n$ finite, there must be a t_4^k such that $t_4^k \wedge \neg d_2 \leq_n \bigvee_{I\in\mathscr{X}} a_I$.

is equivalent to the same condition where we replace all the n with $n+1$.

Recall that since all singletons are denoted, atoms are exactly the singleton types. We need a lemma.

Lemma B.2 *Suppose that for every $t \in \mathscr{T}_n$*

- *$t =_n \mathbf{0}$ if and only if $t =_{n+1} \mathbf{0}$;*
- *$|t|_n = h$ if and only if $|t|_{n+1} = h$.*

Pick $t \in \mathscr{T}_n$, consider an atom $a \in \mathscr{T}_{n+1}$ such that there is no atom $a' \in \mathscr{T}_n$ with $a =_{n+1} a'$. If $a \leq_{n+1} t$ then $|t|_{n+1}$ and $|t|_n$ are both infinite.

Proof:suppose $|t|_n = h$ with h finite. Since every singleton is denoted, $t =_n a_1 \vee \ldots \vee a_h$ for disjoint n-atoms a_i. Then the same equality is true at level $n+1$. We thus deduce $a' \leq_{n+1} a_1 \vee \ldots \vee a_h$ from which we derive that $a' =_{n+1} a_i$ for some i. Contradiction. □

We are now going to check the equivalence of the conditions.

Suppose it is true for the $n+1$ case. Then pick a choice of n-atoms a_I. By the induction hypothesis they are $n+1$ atoms. Suppose $|t_I|_n$ is finite. By the induction hypothesis $|t_I|_{n+1}$ is finite, then there must be a t_4^k such that $t_4^k \wedge \neg d_2 \leq_{n+1} \bigvee_{I\in\mathscr{X}} a_I$. Which implies $t_4^k \wedge \neg d_2 \leq_n \bigvee_{I\in\mathscr{X}} a_I$.

Conversely suppose it is true for n. Pick a choice of $n+1$-atoms a_I. Suppose one of these a_I is not equivalent to an n-atom. Then by lemma B.2, $|t_I|_n = |t_I|_{n+1}$ is infinite. So we can assume that a_I is a n-atom. Then there must be a t_4^k such that $t_4^k \wedge \neg d_2 \leq_n \bigvee_{I\in\mathscr{X}} a_I$. Which implies $t_4^k \wedge \neg d_2 \leq_{n+1} \bigvee_{I\in\mathscr{X}} a_I$.

We have now to prove the condition on the cardinality. We start by observing that all the atoms we have described above (when we proved that every singleton is denoted) are atoms independently of the level. They are atoms because of their shape. We now prove the following

- $|t|_{n+1} = h$ implies $|t|_{n+2} = h$;
- $|t|_{n+1} \geq h$ implies $|t|_{n+2} \geq h$.

from which we can conclude $|t|_{n+1} = h$ if and only if $|t|_{n+2} = h$.

Suppose $|t|_{n+1} = h$. Then $t =_{n+1} a_1 \vee \ldots \vee a_h$ for some disjoint atoms. Thus $t =_{n+2} a_1 \vee \ldots \vee a_h$, and since the a_i are still atoms (and they are still disjoint), $|t|_{n+2} = h$.

Suppose $|t|_{n+1} \geq h$, then $t \geq_{n+1} a_1 \vee \ldots \vee a_h$ for some disjoint atoms. Thus $t \geq_{n+2} a_1 \vee \ldots \vee a_h$, and since the a_i are still atoms (and they are still disjoint), $|t|_{n+2} \geq h$. □

We finally observe that adding the $\mathbb{k}$ to our types is not restrictive, as $\mathbb{k} =_k ch(\mathbf{0})^k$.

Hinging on Proposition B.1, we define preorder between types as follows.

Definition B.3 (Order) *Let $t, t' \in \mathscr{T}_n$, then $t \leq_\infty t'$ if and only if $t \leq_n t'$.*

Due to Proposition B.1, this relation is well defined and induces an equivalence $=_\infty$ on the set of types T. Let $\widetilde{\mathscr{T}}$ be $\mathscr{T}/_{=_\infty}$, we are finally able to produce a unique pre-model $\mathscr{D}$ defined as:

$$\mathscr{D} = \mathbb{B} + \widetilde{\mathscr{T}} .$$

Where

- $[\![ch^+(t)]\!] = \{[t']_{=_\infty} \mid t' \leq_\infty t\}$;
- $[\![ch^-(t)]\!] = \{[t']_{=_\infty} \mid t \leq_\infty t'\}$.

This pre-model defines a new preorder between types that we denote by $\leq$. However, the following proposition proves that $\leq$ is not new but it is the limit of the previous preorders, i.e. $\leq_\infty$.

Proposition B.4 *Let $t, t' \in \mathscr{T}$, then $t \leq t'$ if and only if $t \leq_\infty t'$.*

Proof:We prove it by induction on the height of the types. That is we prove by induction on n that if $t \in \mathscr{T}_n$, then

- $t = \mathbf{0}$ if and only if $t =_\infty \mathbf{0}$;
- $|t| = h$ if and only if $|t|_\infty = h$.

Note that to check emptiness of a type in $\mathscr{T}_{n+1}$ we only invoke types in $\mathscr{T}_n$.

The condition at level 0 only requires that the types $\mathbb{k}$ be interpreted into distinct singletons contained in $\widetilde{\mathscr{T}}$, which is the case.

The second statement, and the whole inductive step are proven as in the proof of Proposition B.1. □

It is now easy to show the following.

Theorem B.5 *The pre-model $(\mathscr{D}, [\![\,]\!])$ is a model.*

Proof:Consider the extensional interpretation $\mathscr{E}[\![\,]\!]$ of types as in Definition 2.2. We have to check that $[\![t]\!] = \varnothing \iff \mathscr{E}[\![t]\!] = \varnothing$. Note that in fact the range of $\mathscr{E}[\![\,]\!]$ is $\mathscr{P}(\mathbb{B} + [\![\mathscr{T}]\!])$. By proposition B.4, we have that $\langle [\![\mathscr{T}]\!], \subseteq \rangle$ is isomorphic to $\langle \widetilde{\mathscr{T}}, \leq \rangle$. Up to this isomorphism, $\mathscr{E}[\![\,]\!]$ coincides with $[\![\,]\!]$. □

B.3 Proof of decidability of finiteness

Given our model of types, we show that we can

1. decide whether a type is finite
2. if it is the case, list all its atoms

To prove our claim we proceed by induction on the height of the types. We strengthen the statement by requiring that all atoms of a finite type t have the same height, or lower, of t. We assume that at height 0, this is the case. It is a reasonable assumption: for example it is the case if we have for base types the type of all integers plus all constant types. Consider a type t of height $n+1$ and assume that for lower heights we can decide whether a type is finite and, if it is the case, list all its atoms. By Theorem 2.6, this guarantees that we can also decide emptiness of all types of height $n+1$. We ask ourselves which atoms can be proved to belong to t. If we put t in normal form, we obtain the disjunction of terms of the form

$$r := ch^+(t_1) \wedge ch^-(t_2) \wedge \bigwedge_i \neg ch^+(t_3^i) \wedge \bigwedge_j \neg ch^-(t_4^j) .$$

(We exclude base types, because they have been considered at height 0, and "mixed types", which can be reduced to one of the "pure" cases.) Only atoms of the form $ch(s)$, can be contained in non-base types. For how many s we can have that $ch(s) \leq t$? A union is finite if and only if all its summands are, thus t is finite if and only if all the r's are finite. When is r finite? First of all it is finite when it is empty, which we can test it by induction hypothesis.

Otherwise if r is not empty, then r is finite if and only if $ch^+(t_1) \wedge ch^-(t_2)$ is finite, which happens exactly when $t_2 \leq t_1$ and $t_1 \wedge \neg t_2$ is finite. For the "if" part, note that $ch(s)$ belongs to $ch^+(t_1) \wedge ch^-(t_2)$, if and only if $s = t_2 \vee s'$ for some $s' \leq t_1 \wedge \neg t_2$. Since $t_1 \wedge \neg t_2$ is finite and of smaller height, then by induction hypothesis we can list all its atoms, thus all the corresponding s''s, thus all the corresponding $ch(t_2 \vee s')$ that are all the possible candidates of atoms of r. By induction hypothesis we also have that all the s' have at most height n.

For the "only if" part it suffices to prove that if $ch^+(t_1) \wedge ch^-(t_2)$ is infinite, then the whole of r is infinite. Assume that for no i, $t_1 \leq t_3^i$ and for no j, $t_4^j \leq t_2$ (otherwise r is empty). We have to find infinitely many s such that $t_2 \leq s \leq t_1$, $s \not\leq t_3^i$ for all i and $t_4^j \not\leq s$ for all j. Pick atoms $a_3^i \leq t_1 \wedge \neg t_3^i$ and $a_4^j \leq t_4^j \wedge \neg t_2$. Note that no a_3^i can coincide with any a_4^j, because they are taken from disjoint sets. Then for any type s' such that $t_2 \leq s' \leq t_1$, the type $s := (s' \vee \bigvee_i a_3^i) \wedge \neg \bigvee_j a_4^j$ belongs to r. It is possible that for two different s' the corresponding s coincide. However such "equivalence classes" of s' are finite. Since there are infinitely many s', there are infinitely many s, so r is infinite.

In summary, for every r that forms t we check whether $t_2 \leq t_1$ and $t_1 \wedge \neg t_2$ is finite, and at the end we find either

that t is infinite (if one of the r is) or that it is finite. In the latter case we have a finite list of candidates to be the atoms of t (namely all $ch(s)$ for s included in the the various $t_1 \wedge \neg t_2$) and to list all the atoms of t we just to check for each candidate its inclusion in t. Which we can do, since they are at most of height $n+1$.

B.4 Proof of Theorem 3.6

We first show that $(\mathscr{V}, [\![]\!]_{\mathscr{V}})$ is a pre-model. Inspecting the typing rules, it is easy to show that for every value v and every types t_1, t_2

1. $\Gamma \vdash v : \mathbf{1}$;
2. $\Gamma \vdash v : t_1$ if and only if $\Gamma \not\vdash v : \neg t_1$;
3. $\Gamma \vdash v : t_1 \wedge t_2$ if and only if $\Gamma \vdash v : t_1$ and $\Gamma \vdash v : t_2$.

Point 1 is a simple application of the subsumption rule. For 2 suppose that exist t such that $v : t$ and $v \neg t$. The only rule to deduce a negative type for a value is the subsumption rule. Therefore there must be a type s, such that $v : s$, $s \leq t$ and $s \leq \neg t$. But then $s = \mathbf{0}$, impossible since the empty type is not inhabited. Suppose instead there exists t such that $\not\vdash v : t$ and $\not\vdash v : \neg t$; if $v = c^s$ then $ch(s)$ is not smaller than t nor than $\neg t$, impossible since $ch(s)$ is atomic. The same can be deduced from the atomicity of b_n for $v = n$ ($[\![b_n]\!] = \{n\}$ see Definition 3.1). Therefore $(\mathscr{V}, [\![]\!]_{\mathscr{V}})$ is a pre-model.

The subsumption rules tells us that $s \leq t \implies [\![s]\!]_{\mathscr{V}} \subseteq [\![t]\!]_{\mathscr{V}}$. For the other direction, if $s \not\leq t$, there is an atom a in $s \setminus t$. For every atom a there is a value v such that $\Gamma \vdash v : a$ (either a constant or a channel). By subsumption $\Gamma \vdash v : s$ and $\Gamma \vdash v : \neg t$, which implies $\Gamma \not\vdash v : t$. Thus $[\![s]\!]_{\mathscr{V}} \not\subseteq [\![t]\!]_{\mathscr{V}}$.

To prove that it is a model we have to check that $[\![t]\!] = \varnothing \iff \mathscr{E}[\![t]\!] = \varnothing$. Again the range of $\mathscr{E}[\![]\!]$ is $\mathscr{P}(\mathbb{B} + [\![\mathscr{T}]\!]_{\mathscr{V}})$. By the observation above, we have that $\langle [\![\mathscr{T}]\!]_{\mathscr{V}}, \subseteq \rangle$ is isomorphic to $\langle \widetilde{\mathscr{T}}, \leq \rangle$. Up to this isomorphism, $\mathscr{E}[\![]\!]$ coincides with $[\![]\!]_{\mathscr{V}}$. □

C More examples

First match policy. We show how is possible to impose a first match policy in a input sum: consider the following process

$$\sum_{i=1..n} \alpha(p_i).P_i \tag{11}$$

and let $ch^+(t)$ be the least type of this form that can be deduced for α (this can be calculated by using the set-theoretic properties of the interpretation and it is at the basis of the algorithmic typing rule for input actions). Define q_i as follows:

$$q_{i+1} = \begin{cases} p_1 & \text{if } i = 0 \\ p_i \wedge \neg \lfloor q_i \rfloor & \text{if } 1 \leq i \leq n \end{cases}$$

Then the process

$$\sum_{\{i \mid \lfloor q_i \rfloor \wedge t \neq \mathbf{0}\}} \alpha(q_i).P_i \tag{12}$$

behaves exactly as the above with the only difference that summand selection is deterministic and obeys a first matching discipline. Indeed, every pattern accepts only the values that are not accepted by the preceding patterns. Note that by applying a first match policy some of the summand could no longer have any chance to be selected (this happens if $\lfloor p_i \rfloor \wedge t \leq \bigvee_{j<i} \lfloor p_j \rfloor$), and therefore they must not be included in (12) since then it would not be well typed (there would be redundant summands), which explains the set used to index the sum.

Best match policy. It is possible to rewrite the process in (11) so that it satisfies a best matching policy. Of course this is possible only if for every possible choice in (11) there always exist a best-matching pattern[6]. If this is the case then with the following definition for q_i's

$$q_i = p_i \wedge (\lfloor p_i \rfloor \setminus \bigvee_{\{j \mid \lfloor p_i \rfloor \wedge t \not\leq \lfloor p_j \rfloor \wedge t\}} \lfloor p_j \rfloor)$$

the process (12) is well-typed and implements the best matching policy for (11), since the difference in the definition of q_i makes the pattern fail on every value for which there exists a more precise pattern that can capture it.

[6] More precisely it is necessary that for every $h, k \in I$ if $\lfloor p_h \rfloor \wedge \lfloor p_k \rfloor \wedge t \neq \mathbf{0}$ then there exists a unique $j \in I$ such that $\lfloor p_j \rfloor \wedge t = \lfloor p_h \rfloor \wedge \lfloor p_k \rfloor \wedge t$.