Semantic subtyping for the pi-calculus

Giuseppe Castagna [a] Rocco De Nicola [b] Daniele Varacca [a]

[a] *CNRS - PPS, Université Paris 7, France*

[b] *Dipartimento di Sistemi e Informatica - Università di Firenze, Italy*

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 of types are interpreted as the corresponding set-theoretic operations. Subtyping is defined as inclusion of the interpretations. We prove 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.

Dedicated to the 60th birthday of Mario Coppo, Mariangiola Dezani-Ciancaglini, and Simona Ronchi della Rocca

1 Introduction and motivations

In this article 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 [Bou92, HT91], in which communication is subjected to pattern matching.

There exists a well established literature on typing and subtyping for the π-calculus (e.g. [PS96, Sew98, YH99, SW02]). 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 [HR02], 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. In that system the following equality is used to *define* the union type:

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

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 the values that have that type, 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.

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. This is a particular interpretation (see Section 5.5 for alternative intuitions), but if one takes this stand, then equality (1) does not seem to be justified. Consider the types $ch^{+}(\mathtt{candy}) \vee ch^{+}(\mathtt{coal})$ and $ch^{+}(\mathtt{candy} \vee \mathtt{coal})$. Both represent boxes. If we have a box of the first type, then we expect to find in it either a candy or 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 `candy` from a `coal` *and then branch*, that is, if it is not possible to branch to different pieces of code for messages of different types (e.g. a typecase, an exception trapping, an overloaded function, ...), then it is not possible to operationally observe any difference between the types in (1). Hennessy and Riely do not have such a construction, therefore (1) is sound.

On the contrary, suppose we are sent a channel c of type $ch^{+}(\texttt{candy}) \vee ch^{+}(\texttt{coal})$ If it is possible to test whether c is of type $ch^{+}(\texttt{candy})$ or of type $ch^{+}(\texttt{coal})$, then we can continue assuming that on c we will receive messages of only one of the two types. In this case a rule such as (1) would be unsound, because it would make it possible to receive on c both `candy` and `coal` and this could make the code crash.

We define a variant of the π-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 π-calculus, and with a restricted form of recursive types.

Advantages of a semantic approach. The main advantage of using a semantic approach is that 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 ℂDuce [BCF03] 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 to 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 article [CF05].

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 ℂDuce systems, to yield a calculus with dynamic type dispatch, overloading, channelled communications and where both functions and channels have first class citizenship. A step in that direction has already been taken with the work in [CDV06].

Related work. The first work on subtyping for π was done by Pierce and Sangiorgi [PS96] and successively extended in several other works [Sew98, NFPV00, YH99].

The work closest to ours, at least for the expressiveness of the types, is the already cited work of Hennessy and Riely [HR02]. 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.

The works of Acciai and Boreale [AB05] and of Carpineti *et al.* [CLP06], define languages similar to ours, with XDuce-like pattern matching. However their type systems are less rich than ours and, most importantly, their subtyping relations are 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 [FCB02, Fri04], that led to the design of ℂDuce [BCF03].

Plan of the article: In Section 2 we describe the types, their semantics, and subtyping relation whose decidability is shown in Section 3. In Section 4 we introduce ℂπ, a variant of π-calculus tailored on the previous types, and show examples of its usage. In Section 5 we discuss possible extensions of ℂπ while similarities with different paradigms are outlined in the conclusion, Section 6. In order to lighten the presentation, we postpone the proofs of all properties stated in the article to the appendixes.

2 Types and subtyping

We shall present in detail a relatively simple system with just base types, channels, and boolean combinators. In Section 5, we will then sketch how to add the product type constructor, recursive types, and functional 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:

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

Combinators are self-explaining, with **0** being the empty type and **1** the type of all values. The "set difference" combinator $s \backslash t$ will be used as a shorthand for $s \wedge \neg t$. 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 $\mathscr{T}$.

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 with 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 later, we have $ch^{-}(t) \leq ch^{+}(\mathbf{1})$.

2.2 Semantics of types

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

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

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:

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

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 $ch^+(t)$ denotes 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 thus 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

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

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 $ch^-(t) \wedge ch^+(t)$, rather than a type constructor.

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 again in the presence of a circle. 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 (Pre-model) *Let $\mathscr{D}$,and $\mathbb{B}$ be sets such that $\mathbb{B} \subseteq \mathscr{D}$, and let $[\![\ ]\!]$ be a function from $\mathscr{T}$ to $\mathscr{P}(\mathscr{D})$. The pair $(\mathscr{D}, [\![\ ]\!])$ is said to be 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). The symbol $+$ will denote disjoint union of sets.

Definition 2.2 (Extensional interpretation) *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 pre-orders 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:

Definition 2.3 (Model) *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 § 5.2), 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. The long and technical proof can be found in Appendix A.2.

Finally, given a model for the types, we define

$$s \leq t \overset{\text{def}}{\Longleftrightarrow} [\![s]\!] \subseteq [\![t]\!]\ , \qquad\qquad s = t \overset{\text{def}}{\Longleftrightarrow} [\![s]\!] = [\![t]\!]\ .$$

2.4 Examples of type (in)equalities and graphical representation

We list here some interesting equations and inequations between types that can be easily derived from the set-theoretic interpretation of types. A first simple example of equality and inequality is

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

which states that every channel c of whatever type $ch(t)$ can always be safely used in a process that does not write on c (since it has also type $ch^-(\mathbf{0})$) and that does not care about what c returns (since it has type $ch^+(\mathbf{1})$).

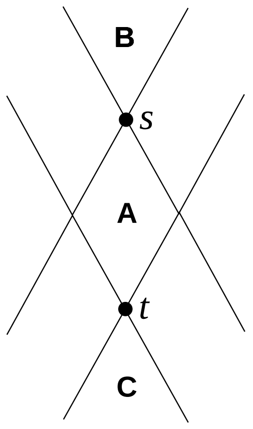

Fig. 1. Channel types

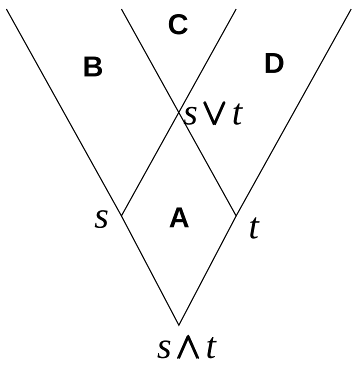

Fig. 2. Some equations

Besides these fiddling relations, far more interesting relations can be deduced and, quite remarkably, in many cases this can be done graphically. Consider the definitions in (3): they tell us that the interpretation of $ch^{+}(t)$ is the set of the interpretations of all types smaller than or equal to t. As such, it can be represented by the downward cone starting from t. Similarly, the upward cone starting from t represents $ch^{-}(t)$. This illustrated in Figure 1 where the upward cone B represents $ch^{-}(s)$ and the downward cone C represents $ch^{+}(t)$. As the reader can easily verify, this representation immediatly gives covariance of input types and contravariance of output types.

If we now pass to Figure 2 we see that $ch^{-}(s)$ is the upward cone B+C and $ch^{-}(t)$ is the upward cone C+D. Their intersection is the cone C, that is the upward cone starting from the least upper bound of s and t which yields the following equation

$$ch^{-}(s) \wedge ch^{-}(t) = ch^{-}(s \vee t) \ . \tag{5}$$

This states that if on a channel we can write values of type s and values of type t, this means that we can write on it values of type $s \vee t$. Dually, by turning Figure 2 upside down it is easy to check the following equation:

$$ch^{+}(s) \wedge ch^{+}(t) = ch^{+}(s \wedge t) \tag{6}$$

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

Similarly, note that the union of $ch^{-}(s)$ and $ch^{-}(t)$ is given by B+C+D and that this is strictly contained in the upward cone starting from $s \wedge t$, since the latter also contains the region A, whence the strictness of the following containment:

$$ch^{-}(s) \vee ch^{-}(t) \lneq ch^{-}(s \wedge t) \ . \tag{7}$$

Actually, the difference of the two types in the above inequality is the region A which represents $ch^{+}(s \vee t) \wedge ch^{-}(s \wedge t)$, from which we deduce

$$ch^{-}(s \wedge t) = ch^{-}(s) \vee ch^{-}(t) \vee (ch^{+}(s \vee t) \wedge ch^{-}(s \wedge t)) \ .$$

By turning Figure 2 upside down again we can check the dual of equation (7):

$$ch^+(s) \vee ch^+(t) \lneq ch^+(s \vee t) \tag{8}$$

As a final example consider the type $ch^+(s) \wedge ch^-(t)$, that is the type of a channel on which we can write values of type t and from which we expect to read values of type t. We have

$$ch^+(s) \wedge ch^-(t) = \mathbf{0} \tag{9}$$

if and only if $t \not\leq s$, i.e. we should expect to read at least what we can write. Once more this can be checked graphically on Figure 1, but in order to show the role of our definitions, let us formally deduce this last equation. By definition, (9) holds if and only if $[\![ch^+(s) \wedge ch^-(t)]\!] = [\![\mathbf{0}]\!]$. By definition of model and the antisymmetry of $\subseteq$ this holds if and only if $\mathscr{E}(ch^+(s) \wedge ch^-(t)) = \mathscr{E}(\mathbf{0})$. By definition of $\mathscr{E}()$ this holds if and only if $\{[\![s]\!]' | [\![s]\!]' \subseteq [\![s]\!]\} \cap \{[\![t]\!]' | [\![t]\!] \subseteq [\![t']\!]\} = \varnothing$. By the reflexivity and transitivity of $\subseteq$ this holds if and only if $[\![t]\!] \not\subseteq [\![s]\!]$, that is, by definition of subtyping if and only if $t \not\leq s$.

3 Decidability of subtyping

For practical applications, it is essential that subtyping relations are decidable. The subtyping relation defined in Section 2 is indeed decidable. The decision procedure is however a bit involved. As we show in details later in this section, we can always reduce the problem of deciding the subtyping between two types to deciding an inclusion of the following form:

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

While in some cases it is easy to decide the inclusion above (for instance, when $t_2 \not\leq t_1$ since then the left-hand side is empty), in general, this requires checking whether a type is *atomic*, that is whether its only proper subtype is the empty type (for sake of simplicity the reader can think of the atomic types as the singletons of the type system[1]). To have an idea of why we have to push the check at the level of atomic types let us once more resort to the graphical representation. Consider the equation (10) above with only two types s and t with $t \lneq s$ (note the strictness of inclusion, which implies that $s \backslash t$ is not empty), and try to check whether:

$$ch^+(s) \wedge ch^-(t) \leq ch^-(s) \vee ch^+(t)\ .$$

The situation is represented in Figure 1 where the region A represents the left-hand side of the inequality, while the region B+C is the right hand side. So to check the subtyping above we have to check whether A is contained in B+C. At first sight these two regions look completely disjoint, but observe that they have at least two

[1] Nevertheless, notice that according to their definition, atomic types may be neither singletons nor finite. For instance, $ch(\mathbf{0})$ is atomic, but in the model defined by equation (2)—more precisely, in the model of values of Theorem 4.5—it is the set of all the synchronisation channels; these are just token identifiers on a countable alphabet, thus the type is countable as well.

points in common, marked in bold in the figure (they are respectively the types $ch(s)$ and $ch(t)$). Now, the containment holds if the region A does not contain any other type besides these two. This holds true if and only if there is no other type between s and t, that is if and only if $s\backslash t$ is an atomic type.

Let us now present the technical details of the decision procedure (proofs can be found in the appendix). First of all we need to define the notions of finite and atomic types.

Definition 3.1 (Atomic and finite types) *An* atom *is a minimal non-empty type. A type is* finite *if it is equivalent to a finite union of atoms.*

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

$$s \leq t \iff s \wedge \neg t = \mathbf{0} \tag{11}$$

which can be derived as follows:

$$s \leq t \iff [\![s]\!] \subseteq [\![t]\!] \iff [\![s]\!] \cap \complement[\![t]\!] = \varnothing \iff [\![s \wedge \neg t]\!] = [\![\mathbf{0}]\!] \iff s \wedge \neg t = \mathbf{0}\ .$$

Thanks to the semantic interpretation we can directly apply set-theoretic equivalences to types (in the rest of the article 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 (11) 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 "$-$" and we grouped literals according to whether they are negated or not). 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—in the case in point De Morgan's laws—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

$$(\bigwedge_{i\in P} ch^{\nu_i}(t_i)) \leq (\bigvee_{j\in N} ch^{\nu_j}(t_j)) \ . \tag{12}$$

Because of equations (5) and (6), we can push the intersection on the left-hand side inside the constructors and reduce (12) to the equation (10) we met in the previous section, and that we recall below:

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

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 types 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 $ch^-(t_2) \cap ch^+(t_3^h) = \mathbf{0}$, so it does not change the inequation);
(4) all contravariant channels on a t_4^k that is not smaller than or equal to t_1 (since then $ch^+(t_1) \cap ch^-(t_4^k) = \mathbf{0}$).

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

Theorem 3.2 *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 $k \in K$, $t_4^k \leq t_1$.

Then

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

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 choice of atoms $a_h \leq t_1 \backslash t_3^h$, with $h \in H$, there exists $k \in K$ such that $t_4^k \leq t_2 \vee \bigvee_{h\in H} a_h$.

The four hypotheses c1–c4 simply state that the right-hand side of the inequation was simplified according to the rules (1–4) 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[2] condition (CA) says that, every time we add to t_2 atoms of t_1 so that we are no longer below any t_3^h then we must end up above some of the t_4^k.

We have already shown at the beginning of this Section an example of the sensitivity of the subtyping relation to atoms. To obtain another, more concrete example of this fact, suppose there are three atoms $\mathtt{err}_1, \mathtt{err}_2, \mathtt{exc}$ and consider the case where $t_2 = \mathtt{int}$, $t_1 = t_2 \vee \mathtt{err}_1 \vee \mathtt{err}_2 \vee \mathtt{exc}$, $t_3 = t_2 \vee \mathtt{exc}$, $t_4 = t_2 \vee \mathtt{err}_1 \vee \mathtt{err}_2$. 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 3.2 is a singleton. The only atom in $t_1 \backslash t_3$ is $\mathtt{err}_1$, and it is true that $t_4 \leq t_2 \vee \mathtt{err}_1$.

As announced, Theorem 3.2 decomposes the subtyping problem of (10) into a finite set of subtyping problems on simpler types (we must simplify the right hand side of inequation (10) 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 a universal quantification on possibly infinite sets $t_1 \backslash t_3^h$, 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 3.3 *If we replace condition (CA) with*

CA. Let $H_f \subseteq H$ be the set of those indices h for which $t_1 \backslash t_3^h$ is finite. For every choice of atoms $a^h \leq t_1 \backslash t_3^h$, with $h \in H_f$, there exists $k \in K$ such that $t_4^k \leq t_2 \vee \bigvee_{h \in H_f} a_h$.*

then Theorem 3.2 still holds.

Therefore it suffices to check the condition just for the $t_1 \backslash t_3^h$ that are finite. This can be done effectively provided that we are able to:

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

We will assume that this is possible for base types and prove that this implies that it is possible for all types.

Lemma 3.4 *There is an algorithm that decides whether a type t is finite and if it is the case, outputs all its atoms.*

Theorem 3.5 *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.

[2] The original condition (CA) as it can be found in [CDV05] was even more involved. We renew our gratituted to the anonymous referee who suggested a major simplification.

4 The $\mathbb{C}\pi$ calculus

We shall present a variant of the π-calculus, that exploits the type system of Section 2. We will present its syntax, semantics, and typing rules, and prove the decidability of the typing relation.

4.1 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 type-case process (e.g. $[M : t]P$ which reduces to P or $\mathbf{0}$ according 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 4.1 (Patterns) *Given a type algebra $\mathscr{T}$, and a set of variables $\mathbb{V}$, a* pattern *p on $(\mathbb{V}, \mathscr{T})$ is a term generated by the following grammar*

Patterns	$p ::=$	x	capture, $x \in \mathbb{V}$
	$\mid$	t	type constraint, $t \in \mathscr{T}$
	$\mid$	$p \wedge p$	conjunction
	$\mid$	$p \mid p$	alternative

such that 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 $Var(p_1) = Var(p_2)$ (where $Var(p)$ denotes the set of variables of $\mathbb{V}$ occurring in p).

Patterns are rather basic: they can test if a value is of a given type, capture it, and combine these tests via conjunctions and disjunctions. So for instance $x \wedge t$ is the pattern that captures a value in x if it is of type t. As a matter of fact, the patterns above lack the main capability peculiar of general patterns that is to deconstruct values. The reason is that here we consider a minimal type system in which the only type constructors are for channel types, and their values are not “constructed” from simpler values (e.g. pairs of values for product constructor) but are constants. So here patterns act more as a placeholder and they are interesting in view of the extension of our language with recursive types (Section 5.2) product types (Section 5.1) or other type constructors.

Following [FCB02, BCF03] we define the semantics of patterns directly on models. A pattern is matched against an element of the domain $\mathscr{D}$ of a model of the types and the matching returns either a substitution for the free variables of the pattern, or a failure, denoted by Ω:

Definition 4.2 *Given a model $[\![\,]\!] : \mathscr{T} \to \mathscr{D}$, an element $d \in \mathscr{D}$, and a pattern p, the matching of d with p, noted by d/p, is the element of $\mathscr{D}^{Var(p)} \cup \{\Omega\}$ defined as follows:*

$$\begin{array}{llll} d/t = \{\} & \textit{if } d \in [\![t]\!] & d/p_1 \wedge p_2 = d/p_1 \otimes d/p_2 & \\ d/t = \Omega & \textit{if } d \in [\![\neg t]\!] & d/p_1|p_2 \;\; = d/p_1 & \textit{if } d/p_1 \neq \Omega \\ d/x = \{x \mapsto d\} & & d/p_1|p_2 \;\; = d/p_2 & \textit{if } d/p_1 = \Omega \end{array}$$

where $\gamma_1 \otimes \gamma_2$ is Ω when $\gamma_1 = \Omega$ or $\gamma_2 = \Omega$ and the union of the two otherwise.

A quite useful 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 $\wr p \int$. In other terms, for every (well-formed) pattern p, there exists a unique type $\wr p \int$ such that $[\![\wr p \int]\!] = \{d \in Dom \mid d/p \neq \Omega\}$. Not only, but this type can be calculated. Similarly, consider a pattern p and a type $t \leq \wr p \int$, 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 4.3 *There is an algorithm mapping every pattern p to a type $\wr p \int$ such that $[\![\wr p \int]\!] = \{d \in \mathscr{D} \mid d/p \neq \Omega\}$.*

Theorem 4.4 *There is an algorithm mapping every pair (t,p), where p is a pattern and t a type such that $t \leq \wr p \int$, to a type environment $(t/p) \in \mathscr{T}^{Var(p)}$ such that $[\![(t/p)(x)]\!] = \{(d/p)(x) \mid d \in [\![t]\!]\}$.*

For such basic patterns the proofs of the properties above are really straightforward. What is remarkable is that these properties hold for polyadic $\mathbb{C}\pi$ with recursive types, as well (Section 5.2).

4.2 The language

The syntax of our calculus is very similar to that of the asynchronous π-calculus a variant of the π-calculus for which message emission is non-blocking. The latter is generally considered as the calculus representing the essence of name passing with no redundant operation. The variant we consider is very similar to the original calculus, but we permit patterned input prefix and guarded choice between different patterns on the same input channel.

$$\begin{array}{llll@{\qquad}llll} \textit{Channels} & \alpha ::= & x & \text{variables} & \textit{Processes } P ::= & & \overline{\alpha}M & \text{output} \\ & \mid & c^t & \text{constant} & & \mid & \sum_{i\in I} \alpha(p_i).P_i & \text{patterned input} \\ & & & & & \mid & P_1 \| P_2 & \text{parallel} \\ \textit{Messages } M ::= & & n & \text{constant} & & \mid & (\nu c^t)P & \text{restriction} \\ & \mid & \alpha & \text{channel} & & \mid & !P & \text{replication} \end{array}$$

where 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 given in Definition 4.1. As customary we use the convention that the empty sum corresponds to the inert process, 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. Choice operators are very useful for specifying nondeterministic behaviours,

Messages

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

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\ (\text{var}) \qquad \frac{\Gamma \vdash M : s \leq t}{\Gamma \vdash M : t}\ (\text{subs})$$

Processes

$$\frac{\Gamma \vdash P}{\Gamma \vdash (\nu c^t)P}\ (\text{new}) \qquad \frac{\Gamma \vdash P}{\Gamma \vdash !P}\ (\text{repl}) \qquad \frac{\Gamma \vdash M : t \quad \Gamma \vdash \alpha : ch^-(t)}{\Gamma \vdash \overline{\alpha} M}\ (\text{output})$$

$$t {\leq} \bigvee_{i\in I} \{\!\!\{ p_i \}\!\!\}\ \frac{\Gamma \vdash \alpha : ch^+(t) \qquad \Gamma, t/p_i \vdash P_i}{\Gamma \vdash \sum_{i\in I} \alpha(p_i).P_i}\ (\text{input}) \qquad \frac{\Gamma \vdash P_1 \quad \Gamma \vdash P_2}{\Gamma \vdash P_1 \| P_2}\ (\text{para})$$

Fig. 3. Typing rules

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. 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 asynchronous π-calculus is that we distinguish between channel variables and channel constants and that the latter are decorated by the type of messages they communicate. This corresponds to our intuition that every box is intimately associated to the type of the objects it can contain. In what follows we will call channel constants also “typed channels”, “boxes”, or “channel values” to distinguish them from channel variables.

The *values* of the language are the closed messages, that is to say the typed channels and the constants: $\quad v ::= n \mid c^t$.

We use $\mathscr{V}$ to denote the set of all values. Every value is associated to a type: every constant n is associated to an atomic basic type b_n (we also assume that every atomic basic type b_n has its corresponding basic value n), while every channel value is associated with the channel type that transport messages of the type indicated in the index. So all the values can be typed by the rules (const), (chan), and (subs) of Figure 3 (actually with an empty Γ) where in the (subs) subsumption rule the $\leq$ is the subtyping relation induced by the model built to prove Theorem 2.4 (see Appendix A.2).

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

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

$$P\|0 \equiv P \qquad P\|Q \equiv Q\|P \qquad P\|(Q\|R) \equiv (P\|Q)\|R$$

$$(\nu c^t)0 \equiv 0 \qquad (\nu c^t)P \equiv (\nu d^t)P\{c^t \rightsquigarrow d^t\} \qquad !P \equiv !P\|P$$

$$(\nu c_1^{t_1})(\nu c_2^{t_2})P \equiv (\nu c_2^{t_2})(\nu c_1^{t_1})P \qquad \text{for } c_1 \neq c_2$$

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

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.

Fig. 4. Context and congruence closure

4.3 Semantics

Let $\mathsf{M} = (\mathscr{D}_{\mathsf{M}}, [\![\;]\!]_{\mathsf{M}})$ be any model (that is, it satisfies Definition 2.3). M induces a subtyping relation $\leq_{\mathsf{M}}$ defined as $s \leq_{\mathsf{M}} t \overset{\text{def}}{\iff} [\![s]\!]_{\mathsf{M}} \subseteq [\![t]\!]_{\mathsf{M}}$. Consider the typing rules for Message in Figure 3, use for the subsumption rule (subs) the $\leq_{\mathsf{M}}$ relation, and denote by $\Gamma \vdash_{\mathsf{M}} M : t$ the corresponding typing relation.

Now consider this new interpretation function $[\![\;]\!]_{\mathscr{V}} : \mathscr{T} \to \mathscr{P}(\mathscr{V})$ defined as $[\![t]\!]_{\mathscr{V}} = \{v \mid \Gamma \vdash_{\mathsf{M}} v : t\}$. It turns out that this interpretation, whatever the model is, satisfies the model conditions of Section 2.3 and furthermore it generates the same subtyping relation as $\leq_{\mathsf{M}}$. The circle we mentioned in the Introduction is now closed.

Theorem 4.5 (Model of values) *Let $(\mathscr{D}, [\![\;]\!])$ be a model and $\leq$ and $\Gamma \vdash M : t$ be, respectively, the subtyping and typing relations it induces. 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 4.2 applies for d being a value. We can thus use it to define the reduction semantics of our calculus:

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

where $P[\sigma]$ denotes the application of substitution σ to process P. The asynchronous 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 non-deterministically 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 by a proper use of type combinators in patterns. 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 4.

This operational semantics is the same as that of π-calculus but the actual process behavior has been refined in two points: (i) communication is subjected to pattern matching and (ii) communication can happen only along values (boxes).

The use of pattern matching is what makes it necessary to distinguish between typed channels and variables: matching is defined only for the formers 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 $\Gamma \vdash x : ch(\texttt{int})$ and $\Gamma \vdash x : ch^{-}(\texttt{string})$. Then, according to the typing rules of our system (see later on) the process $\bar{x}\,\texttt{ciao} \,\|\, x(y).\bar{x}(y \div y)$ is well typed, in the environment Γ, but it would give rise to a run time error by attempting to divide the string $\texttt{ciao}$ by itself:

$$\bar{x}\,\texttt{ciao} \,\|\, x(y).\bar{x}(y \div y) \quad \longrightarrow \quad \bar{x}(\texttt{ciao} \div \texttt{ciao})$$

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 falso quodlibet* deduction rule).

4.4 Typing

In Figure 3, 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 good-formation of processes without assigning them any type. The rules for replication and parallel composition are standard. The rule for restriction is slightly different 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/p_i) of Theorem 4.4) and check whether under this environment the summand process P_i is typeable. This is all that 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 (i) patterns are exhaustive and (ii) there is no useless case [4]. The first check is performed by the side condition of the

[3] Strictly speaking, we do not restrict variables but values, 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

(input) rule: $t \leq \bigvee_{i\in I} \wr p_i \int$ 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). For the second condition one could naively think to add a second side condition such as $\wr p_i \int \wedge t \neq \mathbf{0}$ for all $i \in I$ (we did this naivety in [CDV05]), which should check that the pattern matching is not redundant, by verifying that there does not exists a pattern p_i that will fail with every value of type t (no case is useless). However such a check is meaningful only if t is the *best* possible type we can deduce for the messages arriving on α. In a system with subsumption this condition can be always satisfied by considering a larger t (e.g., $t = \bigvee_{i\in I} \wr p_i \int$), thus, without ensuring that all cases of the pattern matching are useful. Therefore we postpone the verification of this property till the definition of the typing algorithm (Section 4.5) when this "best" type will be available.

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, and can be found in Appendix B.2

Lemma 4.6 (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 4.7 (Congruence) *If* $\Gamma \vdash P$ *and* $P \equiv Q$, *then* $\Gamma \vdash Q$.

Theorem 4.8 (Subject reduction) *If* $\Gamma \vdash P$ *and* $P \to P'$, *then* $\Gamma \vdash P'$.

4.5 Typing algorithm

The decidability of the subtyping relation does not directly imply decidability of the typing relation (only semi-decidability is straightforward). 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, in our case the (output) rule. As it is often the case, the typing algorithm also requires to compute a least upper bound of some given form. In particular, the algorithmic version of the (input) rules requires us to compute 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 4.9 (Upper bound channel) *For every type $s \leq ch^{+}(\mathbf{1})$ there exists a least type t such that $ch^{+}(t)$ is an upper bound of s. We denote such type by $\mathscr{C}(s)$.*

The algorithmic rules are then defined as in Figure 5. Soundness and completeness of these rules with respect to those in Figure 3 are completely straightforward:

non-complying to them may yield a type-error. In process algebræ non-compliance would just block synchronisation.

Messages

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

Processes

$$\frac{\Gamma \vdash P}{\Gamma \vdash (\nu c^t)P}\;\text{(new)} \qquad \frac{\Gamma \vdash P}{\Gamma \vdash !P}\;\text{(repl)} \qquad \frac{\Gamma \vdash M : t \quad \Gamma \vdash \alpha : s \quad s \leq ch^-(t)}{\Gamma \vdash \overline{\alpha}M}\;\text{(output)}$$

$$\mathscr{C}(s)\leq\bigvee_{i\in I}\{\!\!\{p_i\}\!\!\}\;\frac{\Gamma \vdash \alpha : s \quad \Gamma, \mathscr{C}(s)/p_i \vdash P_i}{\Gamma \vdash \sum_{i\in I}\alpha(p_i).P_i}\;\text{(input)} \qquad \frac{\Gamma \vdash P_1 \quad \Gamma \vdash P_2}{\Gamma \vdash P_1 \| P_2}\;\text{(para)}$$

Fig. 5. Algorithmic rules

soundness is obtained by a trivial application of the subsumption rule, while completeness can be easily deduced thanks to the fact that no type is inferred for processes (only good formation is checked), by using the fact that the type $\mathscr{C}(s)$ in the algorithmic (input) rule is always smaller than or equal to the type used by the corresponding rule in Figure 3. Lemma 4.9 and the decidability of $(\mathscr{C}(s)/p)$ (given by Theorem 4.4) immediatly yield the following result.

Theorem 4.10 *The typing relation is decidable.*

Finally, recall that in Section 4.4 we hinted that we cannot statically check that all the branches of a pattern match are useful until we do not deduce the minimum type of the message that a channel can transport. Note that the algorithmic rules deduce for a channel its minimum type, and if this minimum type is, say, s, then by definition $\mathscr{C}(s)$ is the minimum type of the messages that the channel trasports. Therefore in order to check the usefulness of every branch it suffices to add to both the (input) rules in Figure 3 and 5 the side condition $\forall i \in I, \{\!\!\{p_i\}\!\!\} \wedge \mathscr{C}(s) \neq \mathbf{0}$, and all the previous results carry along.

4.6 An example

We present here an example of a $\mathbb{C}\pi$ process. Consider the following situation. A web server is waiting on a channel α. The client wants the server to perform some computation on 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 operations 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.

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

where we used the $\mathbb{C}$Duce convention for patterns according to which $x : t$ is syntactic sugar for $x \wedge t$ In the above process the channel α has type $ch^+(ch^+(t_1) \vee ch^+(t_2))$. Note that, as explained in Section 2.4 (equation (8)), $ch^+(t_1) \vee ch^+(t_2) \neq ch^+(t_1 \vee t_2)$. 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 interleaved sequences of the two, as $ch^+(t_1 \vee t_2)$ would do.

As we discussed in the Introduction, this distinction is not present in analogous versions of process calculi where the axiom $ch^+(t_1) \vee ch^+(t_2) = ch^+(t_1 \vee t_2)$ is present. If such an axion were added to our theory, then we would program P defensively, as if α had the (morally larger) type $ch^+(ch^+(t_1 \vee t_2))$

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

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

5 Extensions and variations

5.1 Polyadic version

The first extension we propose consists in adding product to our type constructors. This is pretty straightforward. It requires adding $t ::= t \times t$ to the productions of types, $M ::= (M, M)$ to the productions of messages, and $p ::= (p_1, p_2)$ to the productions of patterns with the condition that for every subterm (p_1, p_2) of a pattern we have $Var(p_1) \cap Var(p_2) = \varnothing$.

The extensional interpretation becomes $\mathscr{E}(\,) : \mathscr{T} \to \mathscr{P}(\mathbb{B} + \mathscr{D}^2 + [\![\mathscr{T}]\!])$ and requires $\mathscr{E}(t_1 \times t_2) = [\![t_1]\!] \times [\![t_2]\!]$. This completely characterises the subtyping relation. A semantic model can be built, in analogy with Section 2.2. The subtyping relation is still decidable, as well as the typing relation.

The extensions described above suffice to obtain the polyadic calculus. In particular projections can be encoded by pattern matching. By using product types, together with the partially recursive types we show next, we can also encode more structured data, like lists or XML documents.

5.2 Partially recursive types

The types introduced so far can be represented as finite labelled trees. Recursive types are obtained without changing the syntax, by allowing trees to be infinite. As in the type system of $\mathbb{C}$Duce we require such trees to be regular (so as they are finitely representable) and with the property that every infinite branch contains infinitely many nodes labelled by the product constructor (so as to avoid meaningless recursive definitions such as $t = t \wedge t$).

Moreover we require that every branch can contain only finitely many nodes labelled with channel constructor. This amounts to require that the number of nested channel constructors is always bound. Or equivalently, if we were to define recursive types with equations, this amounts to forbid the recursive variable being defined to be used inside a channel constructor (such as $x = ch(x) \vee \texttt{int}$).

The reason for this is that, without this restriction, it is not possible to find a model. To see why, 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 levels of nesting of the channel constructor there are and this stratification allows us to construct the model using the same ideas as presented in Section 2. There are two main usages for arbitrary nested recursion of channels: one is to type "self application", that is a channel that can carry itself; the other is for the definition of typed encodings. In our type system, we can already type self application by using, for instance, the type $ch(\mathbf{1})$: a channel that can carry everything, can clearly carry itself. Alternatively we can recover fully recursive types if we restrict to a *local* version of $\mathbb{C}\pi$ (see Section 5.4 below) which is also enough for encoding functional languages [CDV06].

Furthermore, note that recursion is still allowed with other type constructors, and a recursive type can appear inside a channel constructor provided that the number of occurrences of channel constructors is finite. For instance we are allowed to define the type $ch(IBlist)$, where $IBlist$ is the type of heterogeneous lists of booleans and integers, defined as

$$IBlist = ((\texttt{int} \vee \texttt{bool}) \times IBlist) \vee ch(\mathbf{0})$$

(we use $ch(\mathbf{0})$ as the type of the empty list). Formally we have:

Definition 5.1 (Types) *A type t is a possibly infinite regular tree generated by the following productions*

$$\textit{Types} \qquad t \;::=\; b \mid ch^{+}(t) \mid ch^{-}(t) \mid t \times t \mid \mathbf{0} \mid \mathbf{1} \mid \neg t \mid t \vee t \mid t \wedge t$$

and such that on every infinite branch it has infinitely many occurrences of the product constructor and finitely many occurrences of the channel constructors.

With such recursive types it becomes interesting to use recursive patterns. If we relax the condition defined in Section 5.1 for pair patterns and introduce a "constant pattern" as a case base for recursive pattern, then we can express the powerful patterns of $\mathbb{C}$Duce.

Definition 5.2 (Patterns) *A pattern p is a possibly infinite regular tree generated by the following productions*

$$\textit{Patterns} \qquad p \;::=\; x \mid t \mid (p,p) \mid (x := n) \mid p \wedge p \mid p|p$$

where x denotes a variable, t a type, and n a basic value. Additionally we require that on every infinite branch of p there are infinitely many occurrences of the pair pattern, that for every subterm $p_1 \wedge p_2$ of p $Var(p_1) \cap Var(p_2) = \varnothing$, and that for every subterm $p_1|p_2$ of p $Var(p_1) = Var(p_2)$. Their semantics is defined as follows

$$\begin{array}{rcllrcll}
d/t & = & \{\} & \textit{if } d \in [\![t]\!] & d/p_1 \wedge p_2 & = & d/p_1 \otimes d/p_2 & \\
d/t & = & \Omega & \textit{if } d \in [\![\neg t]\!] & d/(p_1,p_2) & = & d/p_1 \otimes d/p_2 & \\
d/x & = & \{x \mapsto d\} & & d/p_1|p_2 & = & d/p_1 & \textit{if } d/p_1 \neq \Omega \\
d/(x := d) & = & \{x \mapsto d\} & & d/p_1|p_2 & = & d/p_2 & \textit{if } d/p_1 = \Omega
\end{array}$$

where $\gamma_1 \otimes \gamma_2$ *is* Ω *when* $\gamma_1 = \Omega$ *or* $\gamma_2 = \Omega$ *and otherwise is the element* $\gamma \in \mathscr{D}^{Dom(\gamma_1)\cup Dom(\gamma_2)}$ *such that:*

$$\gamma(x) = \begin{cases} \gamma_1(x) & \textit{if } x \in Dom(\gamma_1)\backslash Dom(\gamma_2), \\ \gamma_2(x) & \textit{if } x \in Dom(\gamma_2)\backslash Dom(\gamma_1), \\ (\gamma_1(x), \gamma_2(x)) & \textit{if } x \in Dom(\gamma_2) \cap Dom(\gamma_1). \end{cases}$$

Let us give an example of recursive pattern that uses a constant pattern $(x := n)$. If we match a value of the type *IBlist* defined above, against the recursively defined pattern $p = (x : \texttt{int}, p)|(\_, p)|(x := nil^{\mathbf{0}})$, then we capture in x the list of all integers occurring in the matched value. More in details, the pattern is composed of three alternative subpatterns, each subpattern being applied only if the preceding ones fail. The first subpattern matches if the head of the list is of type `int`. In that case it captures the head in x and recursively applies the pattern to the tail. If the head is not of type `int`, then the second patterns skips it, and recursively applies the pattern to the tail. The constant pattern is applied only if the previous two patterns failed, that is if the matched value is not a pair (head,tail). This means that the value is the empty list, and therefore we associate $nil^{\mathbf{0}}$ to x. The third case of the the definition of γ states that for the whole pattern, x is associated to the list—actually the pair (head,tail)—of the values captured by x in each pair subpattern. Both Theorems 4.3 and 4.4 hold also for this extension (the proofs are similar to those found in [FCB02]) and the algorithm of the latter deduces for x the type $t = (\texttt{int} \times t) \vee ch(\mathbf{0})$, that is the type of the lists of integers.

This kind of recursive types and patterns are enough to encode XML data types and manipulate them *à la* ℂDuce. The reader can refer to [BCF03] for more details.

5.3 *Arrow types*

We can extend the type system further by adding function types, so that processes could send ℂDuce expressions as messages. To construct the model, we need to combine the techniques used for ℂDuce with the ones presented in this work.

However, we still cannot get full recursive types, due to the limitation described above. Moreover, we do not know whether the subtyping relation for this system is decidable. The techniques used for the simple system cannot be extended here, because we do not know how to decide whether an arrow type denotes a finite set.

5.4 *The local calculus*

We do not investigate in detail the last two extensions proposed above, because, although theoretically challenging, they do not have much practical interest. In the applications, we may not want to have the full power of the π-calculus. In particular it has been observed [Mer00] that the *input capability*, the ability to use in input a received channel, is difficult to implement. In practice it is convenient to restrict to the so-called *local* variant of the π-calculus [Mer00], where the input capability is not allowed.

In our case this restriction has other important consequences:

- the covariant channel type $ch^+(t)$ is no longer necessary. The example of Section 5.2 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 different polarity 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 $t_i \leq t$.
- it is possible to define a type-respecting encoding of ℂDuce into ℂπ, similar to the Milner-Turner encoding of the simply typed λ-calculus in π (see for instance [SW02]). This makes explicit arrow types not necessary. However the standard translation of arrow types into channel types does not respect equality, therefore to devise a type-respecting encoding a more subtle approach was needed.

The contribution described above was carried out by the first and the third authors, together with Mariangiola Dezani [CDV06].

5.5 *Alternative models*

Hitherto, the whole discussion is based on the intuition that channels always have both input and output capabilities, intuition that we materialised with the definition of the model given in Appendix A.2. However, this is just a *particular* model based on a *particular* intuition. As a matter of fact, the semantic subtyping approach provides two degrees of freedom in the definition of a model and, thus, of a subtyping relation:

(1) We can give different definitions of the extensional interpretation (i.e., Definition 2.2).
(2) Once the extensional definition is set, there may exist different models, that is, different premodels that satisfy Definition 2.3 for the given $\mathscr{E}$.

Both knobs can be turned to tune the subtyping relation, but between them the one that really matters is the first one.

The extensional interpretation is the one that devises the characteristics of the subtyping relation: from our experience, different models induce slight variations to the subtyping relation, if any at all. For instance, in the definition of ℂDuce the chosen extensional interpretation admits models that induce different subtyping relations [FCB02]. These models, however are rather difficult to find and differ only in the degree of sharing in recursive types [Fri04]. For this reason, we believe that, once the extensional interpretation is defined, the existence of a model matters much more than its definition. Moreover in our case we conjecture that all models for the extensional interpretation of Definition 2.2 induce the same subtyping relation. This explains why we focused on the extensional interpretation and relegated the definition of the model to Appendix A.2.

On the contrary it can be very interesting to study alternative definitions of the extensional interpretation, since they correspond to different intuitive semantics and induce substantially different subtyping relations. The reason why we chose our current definition for the extensional interpretation is that it allows us to mix and compare channels of different polarities. This interpretation pushed the approach to its limits, as the issues with recursion and atomic types clearly show. But it is possible to consider different interpretations, in order to either recover existing subtyping relations, or make the subtyping relation more robust with respect to some features. As an example, let us briefly hint at four alternative definitions of the extensional interpretation.

(1) We can define the extensional interpretation so that it reflects an intuitive model in which not only read-and-write channels but also read-only channels and write-only channels are present. Here we would interpret $ch^+(t)$ as the set of all read-only and read-and-write channels for a type s smaller than or equal to t (and similarly for $ch^-(t)$). Although $ch(t)$ would still be the intersection of $ch^+(t)$ and $ch^-(t)$, this would substantially change the subtyping relation (there no longer is a type of all channels, channels of different polarities are less comparable, etc.) yielding a subtyping relation closer to the one defined by Pierce and Sangiorgi [PS96].

(2) We can define an extensional interpretation sensitive to the identity of individual channels, that is, an interpretation in which the read-and-write channel type no longer is atomic. We would then obtain a subtyping relation which would be compatible with a language in which pattern matching can also test the name of a channel.

(3) We can draw inspiration from the models of ℂDuce and interpret $ch^+(t)$ as the set of (the interpretations of) functions of type $\texttt{unit} \to t$, $ch^-(t)$ as the set of (the interpretations of) functions of type $t \to \texttt{unit}$, and $ch(t)$ as their intersection. Once more this would induce a substantially different subtyping relation. In particular, this interpretation is compatible with an unconstrained definition of recursive types: since in ℂDuce the intersection of two function spaces is never empty, then the counterexample given in Section 5.2 no longer works ($b \lneq t$ holds in all models).

(4) We can define a variant of the previous interpretation which instead of single functions uses records of functions to interpret channels. In particular we would interpret $ch^+(t)$ as the record type $\{\texttt{read: unit} \to t\}$, $ch^-(t)$

as the record type {`write:` $t \rightarrow$ `unit`}, and finally $ch(t)$ as the record type {`read:` `unit` $\rightarrow t$, `write:` $t \rightarrow$ `unit`}. This interpretation, too, is compatible with full recursion (as an aside, this is the way in which references types are encoded and implemented in the language ℂDuce, which explains why pointers are possible even if ℂDuce features fully recursive types) but keeps the interpretation of read-only, write-only, and read-and-write channel types, distinct. This interpretation should also induce a conservative extension of the Pierce and Sangiorgi's subtyping relation.

The four above are just some of the possible different interpretations for channel types. Although in this work we considered one particular interpretation, we did not do so with the purpose to fix it as the best possible interpretation, but rather with the purpose to use it to illustrate how to apply the technique of semantic subtyping to mobile processes.

6 Conclusion

Pierce and Sangiorgi's subtyping for the π-calculus, though very elegant, is structurally very poor: it essentially amounts to compare the levels of nesting of channel constructors with the same polarity. In order to obtain a much richer and expressive subtyping relation, we combine here their types with union, intersection, and negation types. This is not a new idea—at least for what concerns unions and intersections—, but the originality of our approach is that the theory is semantically justified via a set theoretic interpretation of types as sets of values, which looks as quite a reasonable interpretation. The naturalness of the interpretation is justified and supported by several technical aspects, and reinforced by the results exposed in the follow up of this work [CDV06] where, together with Mariangiola Dezani, the first and third author devised a local version of ℂπ and defined a type-preserving translation of ℂDuce into the latter.

While the interpretation is very simple, its consequences are not. We have seen that deciding subtyping requires to enumerate and check one by one the atoms that compose the types involved in the verification. Such a degree of complexity is present only in the general framework. This is acceptable since our work aims at establishing the foundational basis of subtyping for π-calculus. Of course, such a degree of complexity makes the calculus unfit for practical applications. However in a practical scenario one would rather resort to the local version of ℂπ as defined in [CDV06] and, in that case, the extra complexity of subtyping disappears, the subtyping algorithm being reduced to perform classic structural checks on syntactic types.

The fact that here we have to descend to the very structure that composes types (the world "atoms" is quite suggestive in this case) is not overly surprising. The point is that we are touching deep into the semantics of computations. This is witnessed by the fact that some characteristics (in some case, some "oddities") of ℂπ are shared by completely different paradigms for which a semantic subtyping technique was used. For instance, $\mathbb{C}$Duce function values require some special non-structural typing rule which uses negated literals. This kind of rule becomes necessary also for $\mathbb{C}\pi$ as soon as one consider its local variant [CDV06]. A much more striking correspondence happens with atoms: we have shown that in order to decide the subtyping relation in $\mathbb{C}\pi$ one must be able to decide the atomicy of the types. Quite surprisingly the same problem appears in λ-calculus (actually, in any semantic subtyping based system) as soon as we try to extend it with polymorphic types. Imagine that we embed our types (whatever they are) with type variables $X, Y, \ldots$. Then the "natural" (semantic) extension of the subtyping relation is to quantify the interpretations over all substitutions for the type variables:

$$t_1 \leq t_2 \quad \overset{\text{def}}{\iff} \quad \forall s. [\![t_1[s/X]]\!] \subseteq [\![t_2[s/X]]\!] \;. \tag{13}$$

Consider now the following inequality (taken from [HFC05]) where t is a closed type

$$(t, X) \leq (t \times \neg t) \vee (X \times t). \tag{14}$$

It is easy to see that this inequality holds if and only if t is atomic. If t is not atomic, then it has at least one non-empty proper subtype, and (13) does not hold when we substitute this subtype for X. If instead t is atomic, then for all X either $t \leq X$ or $t \leq \neg X$, whence (14). Note that this example does not use any fancy or powerful type constructor, such as arrows or channels: it only uses products and type variables. So it applies to all polymorphic extensions of semantic subtyping where, once more, deciding subtyping reduces to deciding whether some type is atomic or not.

These and other similarities are discussed in [Cas05] to which the reader can refer for deeper analysis and a discussion on perspectives.

Acknowledgements. This article, as well as the follow up in which Mariangiola Dezani was involved, were conceived in the *lingerie* of the École Normale Supérieure in Paris, on the white board near the Lavazza coffee machine. A necessary treat for an all-Italian crew. Besides providing the white board and the coffee machine (though, alas, we had to provide the coffee) the École Normale Supérieure offered Rocco a visiting professorship grant. Funding was also partially provided by the European FET Project MyThS, and by French ACI project Tralala. We are also grateful to Alain Frisch for discussions. We are extremely grateful to an anonymous referee who was able to simplify condition CA (the new condition is equivalent to the original, but much simpler and intuitive).

We want to warmly thank Stefano Berardi and Ugo de' Liguoro for having invited us to contribute to this issue in honour of Mario Coppo, Mariangiola Dezani-Ciancaglini, and Simona Ronchi della Rocca. A work about adding union and intersection types (besides negation) to process algebrae seemed to us an appropriate tribute to pay to the persons that founded this branch of type theory. It is for us an honour and a real pleasure to have this opportunity to express all the admiration, respect, and friendship we have for them.

References

[AB05] L. Acciai and M. Boreale. XPi: A typed process calculus for XML messaging. In *FMOODS*, number 3535 in LNCS, pages 47–66. Springer, 2005.

[BCF03] 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.

[Bou92] 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.

[Cas05] G. Castagna. Semantic subtyping: challenges, perspectives, and open problems. In *ICTCS 2005, Italian Conference on Theoretical Computer Science*, number 3701 in Lecture Notes in Computer Science, pages 1–20. Springer, 2005.

[CDV05] G. Castagna, R. De Nicola, and D. Varacca. Semantic subtyping for the π-calculus. In *LICS '05, 20th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 2005.

[CDV06] G. Castagna, M. Dezani Ciancaglini, and D. Varacca. Encoding ℂDuce into the ℂπ-calculus. In *CONCUR 2006, 17th. International Conference on Concurrency Theory*, number 4137 in LNCS, pages 310–326. Springer, 2006.

[CF05] G. Castagna and A. Frisch. A gentle introduction to semantic subtyping. In Proc. of *PPDP '05, the 7th ACM SIGPLAN Int. Symp. on Principles and Practice of Declarative Programming*, ACM Press (full version) and *ICALP '05, 32nd Int. Colloquium on Automata, Languages and Programming*, LNCS n. 3580, Springer (summary), Lisboa, Portugal, 2005. Joint ICALP-PPDP keynote talk.

[CLP06] S. Carpineti, C. Laneve, and L. Padovani. Piduce – a project for experimenting web services technologies. Unpublished manuscript. Available at `http://www.cs.unibo.it/PiDuce/#pt`, 2006.

[FCB02] A. Frisch, G. Castagna, and V. Benzaken. Semantic Subtyping. In *LICS '02, 17th Annual IEEE Symposium on Logic in Computer Science*, pages 137–146. IEEE Computer Society Press, 2002.

[Fri04] A. Frisch. *Théorie, conception et réalisation d'un langage de programmation fonctionnel adapté à XML*. PhD thesis, Université Paris 7, December 2004.

[HFC05] H. Hosoya, A. Frisch, and G. Castagna. Parametric polymorphism for XML. In *POPL '05, 32nd ACM Symposium on Principles of Programming Languages*. ACM Press, 2005.

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

[HT91] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *Proc. ECOOP 91*, volume 512 of *LNCS*, pages 133–147. Springer, 1991.

[Mer00] M. Merro. *Locality in the pi-calculus and applications to distributed objects*. PhD thesis, Ecole des Mines de Paris, Nice, France, 2000.

[NFPV00] Rocco De Nicola, Gian Luigi Ferrari, Rosario Pugliese, and Betti Venneri. Types for access control. *Theor. Comput. Sci.*, 240(1):215–254, 2000.

[PS96] B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5), 1996.

[Sew98] P. Sewell. Global/local subtyping and capability inference for a distributed π-calculus. In *Proc. of 25th ICALP*, volume 1443 of *LNCS*, pages 695–706, 1998.

[SW02] D. Sangiorgi and D. Walker. *The π-calculus*. Cambridge University Press, 2002.

[YH99] N. Yoshida and M. Hennessy. Subtyping and locality in distributed higher order processes. In *Proc. of 10th CONCUR*, LNCS n. 1664, pages 557–572, 1999.

A Proofs from Sections 2 and 3

A.1 Characterising inclusion (Theorem 3.2 and Proposition 3.3)

In this section we first prove Theorem 3.2 and then strengthen the result as in Proposition 3.3.

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 than or equal to 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

$$\bigcap_{i\in I} ch^+(d_1^i) = ch^+(\bigwedge_{i\in I} d_1^i) \qquad \text{and} \qquad \bigcap_{j\in J} ch^-(d_2^j) = ch^-(\bigvee_{j\in J} d_2^j)\ .$$

Also, if there exist h, h' such that $d_3^{h'} \leq d_3^h$, then 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 $d_3^{\overline{h}} \not\geq d_2$ for some $\overline{h}$ then no element of $ch^-(d_2)$ can be in $ch^+(d_3^{\overline{h}})$. 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 $\overline{h}$ we have $d_1 \leq d_3^{\overline{h}}$, or if for some $\overline{k}$ we have $d_2 \geq d_4^{\overline{k}}$, since then, for instance in the former case, $ch^+(d_1)$ is contained in $ch^+(d_3^{\overline{h}})$ 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 $h \in H$ choose an element $x_h \in d_1 \setminus d_3^h$. Clearly we have that $d_2 \vee \{x_h \mid h \in H\}$ 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 $d_4^k \setminus d_2 \leq \{x_h \mid h \in H\}$.

This proves the necessity of condition (CA): for every choice of $x_h \in d_1 \setminus d_3^h$ there must be a d_4^k such that $d_4^k \setminus d_2 \leq \{x_h \mid h \in H\}$.

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 $d_2 \vee \{x_h \mid h \in H\}$: 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 3.3. Consider the case where for some h the sets $d_1 \setminus d_3^h$ are infinite. Let $H_i \subseteq H$ be the set of such h. Pick $\overline{h} \in H_i$, and let $\overline{H} = H \setminus \{\overline{h}\}$. 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 $x'_{\overline{h}}$ and $x''_{\overline{h}}$ we have that the same d_4^k satisfies $d_4^k \setminus d_2 \leq \{x_h \mid h \in \overline{H}\} \vee \{x'_{\overline{h}}\}$, and $d_4^k \setminus d_2 \leq \{x_h \mid h \in \overline{H}\} \vee \{x''_{\overline{h}}\}$. Therefore we must have $d_4^k \setminus d_2 \subseteq \{x_h \mid h \in \overline{H}\}$. Repeating this for every index in H_i, we conclude that $d_4^k \setminus d_2 \leq \{x_h \mid h \in H \setminus H_i\}$. Noting that $H \setminus H_i = H_f$, we conclude the proof that the condition (CA) is equivalent to

[5] It is in fact $d_4^k \setminus d_2 = \{x\}$, since $d_4^k \not\leq d_2$.

condition (CA*): for every choice of $x_h \in d_1 \setminus d_3^h$, $h \in H_f$, there must be a d_4^k such that $d_4^k \setminus d_2 \leq \{x_h \mid h \in H_f\}$. (We could improve further by considering only those $d_1 \setminus d_3^h$ whose cardinality is not greater than the number of d_4^k - we do not need this for our purposes.)

A.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. To understand the definitions and the proofs in this section, it is advisable to read first Section 3 and Appendix A.1.

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

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 $\mathscr{B}[\![b]\!] \subseteq \mathbb{B}$. Finally, we need a small technicality: we add to our types of height 0 the types $\overbrace{ch(\ldots(ch}^{k}(\mathbf{0})))$, that we denote here as $\mathbb{k}$. Although at higher levels these types are just syntactic sugar, we need them at level 0 to witness the existence of infinitely many channel types. The pre-model at level 0 is exactly formed by the basic types plus the positive natural numbers to model the $\mathbb{k}$. Therefore $\mathscr{D}_0 = \mathbb{B} + \mathbb{N}^+$ with $[\![b]\!]_0 = \mathscr{B}[\![b]\!]$ and $[\![\mathbb{k}]\!]_0 = \{k\}$. The boolean combinators are interpreted 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:

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

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 A.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. For elements of $\mathbb{B}$ this was an assumption. For elements of $\widetilde{\mathscr{T}_n}$, observe that the singleton $\{[t]_{=_n}\}$ is denoted by the type $ch(t)$.

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 $\mathscr{T}_n / =_n$. Then we set $\mathscr{D}_{n+1} \overset{\text{def}}{=} \mathbb{B} + \widetilde{\mathscr{T}_n}$, 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 now the semantics of $\mathbb{1} = ch(\mathbf{0})$ is the expected one, 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 appropriately desugar the $\mathbb{k}$s, and ignore their existence.

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

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

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 $\Bbbk$. The condition to check at level 0 is the following

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

which can be true only if there are two different $\Bbbk \in P$ or if the only $\Bbbk$ 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

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

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

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

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

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 involved 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

• $t_2 \leq_n t_1$		• $t_2 \leq_{n+1} t_1$
• for all $h \in H$, $t_3^h \geq_n t_2$		• for all $h \in H$, $t_3^h \geq_{n+1} t_2$
• for all $k \in K$, $t_4^k \leq_n t_1$	are equivalent to	• for all $k \in K$, $t_4^k \leq_{n+1} t_1$
• for all $h \in H$, $t_3^h \not\geq_n t_1$		• for all $h \in H$, $t_3^h \not\geq_{n+1} t_1$
• for all $k \in K$, $t_4^k \not\leq_n t_2$		• for all $k \in K$, $t_4^k \not\leq_{n+1} t_2$

because of the induction hypothesis.

We have to check that the condition (CA*):

Let $H_{f,n}$ be the set of $h \in H$ such that $|t_1 \backslash t_3^h|_n$ finite. For every $a_h \in Atom_n$, $a_h \leq_n t_1 \backslash t_3^h$, $h \in H_{f,n}$, there must be a t_4^k such that $t_4^k \backslash t_2 \leq_n \bigvee_{h \in H_{f,n}} a_h$.

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 to prove that the condition (CA*) at level n works on exactly the same atoms as at level $n+1$:

Lemma A.2 *Suppose that for every $t \in \mathscr{T}_n$*
– $t =_n \mathbf{0}$ if and only if $t =_{n+1} \mathbf{0}$;
– $|t|_n = l$ if and only if $|t|_{n+1} = l$.
Pick $t \in \mathscr{T}_n$ and an atom $a \in \mathscr{T}_{n+1}$. If $a \leq_{n+1} t$ and $|t|_n$ is finite, then there exists an atom $a' \in \mathscr{T}_n$ with $a =_{n+1} a'$.

Proof: suppose $|t|_n = l$ with l finite. Since every singleton is denoted, $t =_n a_1 \vee \ldots \vee a_l$ for disjoint n-atoms a_i. Then the same equality is true at level $n+1$. Since $a \leq_{n+1} t$, then $a \leq_{n+1} a_1 \vee \ldots \vee a_l$ from which we derive that $a =_{n+1} a_i$ for some i. Thus $a' = a_i$ satisfies the required condition. $\square$

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_h, $h \in H_{f,n}$. By the induction hypothesis the a_h are $n+1$-atoms, too. Also, by the induction hypothesis $|t_1 \backslash t_3^h|_{n+1}$ is finite if and only if $|t_1 \backslash t_3^h|_n$ is finite. Thus $H_{f,n} = H_{f,n+1}$. Since (CA*) is true at level $n+1$, then there must be a t_4^k such that $t_4^k \backslash t_2 \leq_{n+1} \bigvee_{h \in H_{f,n+1}} a_h$. Which implies $t_4^k \backslash t_2 \leq_n \bigvee_{h \in H_{f,n}} a_h$.

Conversely suppose it is true for n. Pick a choice of $n+1$-atoms a_h, $h \in H_{f,n+1}$. If one of these a_h is not equivalent to an n-atom, then by Lemma A.2, $|t_1 \backslash t_3^h|_{n+1}$ would be infinite. Thus we can assume that all a_h are n-atoms. As above we have $H_{f,n} = H_{f,n+1}$, and since (CA*) is true at level n, there must be a t_4^k such that $t_4^k \backslash t_2 \leq_n \bigvee_{h \in H_{f,n}} a_h$. Which implies $t_4^k \backslash t_2 \leq_{n+1} \bigvee_{h \in H_{f,n+1}} a_h$.

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

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

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

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

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

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

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

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

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 A.1. □

It is now easy to show the following.

Theorem A.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 A.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 $[\![\,]\!]$. □

A.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 3.2, 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 Proofs from Section 4

B.1 *Proof of Theorem 4.5*

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 there exists 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 $v = n$. Therefore $(\mathscr{V}, [\![\,]\!]_{\mathscr{V}})$ is a pre-model.

By the subsumption rule we have that if $v : s$ and $s \leq t$ then $v : t$. Therefore $s \leq t \implies [\![s]\!]_{\mathscr{V}} \subseteq [\![t]\!]_{\mathscr{V}}$. For the other direction, if $s \not\leq t$, there is an atom a in $s \backslash t$. For every atom a there is a value v such that $\Gamma \vdash v : a$ (this is clearly true for channels, while it was an assumption for basic types). 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 $[\![\,]\!]_{\mathscr{V}}$. □

B.2 *Proof of the subject reduction*

As usual, the crucial step is the substitution lemma 4.6. We need to prove

- 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 $\Gamma \vdash P[v/p]$.

This is done by induction on the typing rules, by making use of Theorem 4.4. Then consider a well-typed premise of the reduction rule: $\Gamma \vdash \overline{c^t} v \parallel \sum_{i \in I} c^t(p_i).P_i$. This means that $\Gamma \vdash v : t$ and $\Gamma, t/p_i \vdash P_i$. Since $t \leq \bigvee_{i \in I} \wr p_i \int$, there must be a j such that $\vdash v : \wr p_j \int$. For all such j, the substitution v/p_j is defined. By the substitution lemma, for all such j we have $\Gamma \vdash P_j[v/p_j]$.

B.3 *Proof of Lemma 4.9*

Take a nonempty type $s \leq ch^+(\mathbf{1})$. This means that its disjunctive normal form contains only channel types. Consider first the case where s is composed of only one clause $s = ch^+(t_1) \wedge ch^-(t_2) \wedge \bigwedge_h \neg ch^+(t_3^h) \wedge \bigwedge_k \neg ch^-(t_4^k)$. Since s is not empty we have

- $t_2 \leq t_1$ and
- $\forall h \in H, t_1 \not\leq t_3^h$ and
- $\forall k \in K, t_4^k \not\leq t_2$ and
- there exists a choice of atoms $a_h \leq t_1 \backslash t_3^h$ for $h \in H_f$ such that for no $k \in K$, $t_4^k \leq t_2 \vee \bigvee_{h \in H_f} a_h$.

Consider now some type t and the inequation $s \leq ch^+(t)$. This is satisfied if an only if $s \wedge \neg ch^+(t) = \mathbf{0}$. We can think of $ch^+(t)$ as an extra $ch^+(t_3^h)$ added to the normal form of s. In order to have that $s \wedge \neg ch^+(t)$ is empty, we only have two possibilities. The first is that $t_1 \leq t$. Therefore the first candidate for least t is precisely t_1. But can it be smaller than this?

First, note that we must have that $t \geq t_2$, as otherwise we cannot have $s \leq ch^+(t)$. Therefore to obtain a smaller t we must remove some atoms in $t_1 \setminus t_2$. Which ones? Consider all possible choices of atoms $a_h \leq t_1 \backslash t_3^h$ for $h \in H_f$ such that for no $k \in K$, $t_4^k \leq t_2 \vee \bigvee_{h \in H_f} a_h$. As noticed, since s is not empty, there must be at least one such choice.

We claim that none of those a_h can be removed from t_1. To show this, consider a choice of atoms a_h as above with $h \in H_f$ and let $a = a_{\bar{h}}$ for some $\bar{h} \in H_f$. Consider $t = t_1 \setminus a$ and recall we can consider t as one extra t_3^h in the normal form of s. Now we must check condition (CA*) for this new clause. Let $H^\bullet = H \cup \{\bullet\}$, with $t_3^\bullet = t$. Note that $t_1 \setminus t = a$ is finite, and thus $H_f^\bullet = H_f \cup \{\bullet\}$. By putting $a = a_\bullet$, we can see the above choice of atoms as a choice of atoms a_h, with $h \in H_f^\bullet$. Indeed the atom a plays the double role of $a_{\bar{h}}$ and $a_\bullet$.

In order for (CA*) to be satisfied, we should be able to find a t_4^k such that $t_4^k \leq t_2 \vee \bigvee_{h \in H_f^\bullet} a_h = t_2 \vee \bigvee_{h \in H_f} a_h$, which it is not possible by hypothesis. Then, such atoms cannot be removed from t_1.

Now, consider an atom a that is not of this form. Reasoning in similar way as above we can show that we can take a out of t_1 if and only if for all possible choices of atoms $a_h \leq H_f$, such that for no $k \in K$, $t_4^k \leq t_2 \vee \bigvee_{h \in H_f} a_h$, there is $\bar{k}$ such that $t_4^{\bar{k}} \setminus (t_2 \vee \bigvee_{h \in H_f} a_h) = a$.

How many such atoms there are? Only finitely many, as the universal quantification above is finite. Therefore we can remove these atoms one by one. The corresponding t is such that $s \leq ch^+(t)$ and moreover we cannot remove any other atom. Finally all such atoms can be computed.

The above proves the statement for types s composed only of one clause. Consider a type s whose disjunctive normal form is $s = s_1 \vee \ldots \vee s_n$, and suppose for each s_i the type t_i is the least such that $s_i \leq ch^+(t_i)$. Then the type $t = t_1 \vee \ldots \vee t_n$ is the least such that $s \leq ch^+(t)$. Clearly it has the property. To show it is the least such, remove one atom a from it and suppose it still has the property. Therefore no s_i contains a. However a belongs to one of the t_i. Therefore, by removing a from such t_i we would obtain a smaller t_i' such that $s_i \leq ch^+(t_i')$, contradiction.