Semantic subtyping for the π-calculus

Giuseppe Castagna
École Normale Supérieure
castagna@di.ens.fr

Rocco De Nicola
University of Firenze
denicola@dsi.unifi.it

Daniele Varacca
École Normale Supérieure
varacca@di.ens.fr

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 that consequently is sound and complete with respect to the semantics. The type system we consider includes read-only and write-only channel types, product types, recursive types, as well as unions, intersections, and negations of types which are interpreted as the corresponding set-theoretic operations. We prove the decidability of the subtyping relation and formally describe the subtyping algorithm.*

In order to fully exploit the expressiveness of the new type system (which subsumes several existing ones), we endow the π-calculus with structured channels where communication is subjected to pattern matching that performs dynamic typecase. These features pave the way toward the integration of functional and concurrent features within the same framework, obtained by combining π-calculus and ℂDuce, a functional programming language with semantic subtyping.

1 Introduction and motivations

Traditionally types have played a very limited role in concurrency; they have been used essentially for specifying the nature of the exchanged values. The picture has changed after the introduction of formalisms dealing with systems of mobile processes. In addition to the classical use of types, e.g. for static detection of run time errors, for enhancing programs readability, for memory management, for abstracting from implementation details, etc., types have emerged as an important tool for specifying interfaces and interactions, for controlling process mobility and resource usage, for improving efficiency of verification algorithms, etc.

In this paper we shall concentrate on type systems for a concurrent language in which values can be exchanged between concurrent agents via communication channels that can be dynamically generated. The language we shall consider is a variant of the asynchronous π-calculus [2] extended with structured channels and where communication is subjected to pattern matching.

There exists an extensive literature about typing and subtyping for the π-calculus. However, all the papers we are aware of rely on subtyping relations or on type equivalences that are based on conditions and restrictions that have little semantic justifications. The choices depend on the formalizations and are assumed in order to simplify the system or simply to have it working.

In our view, all syntactic formalizations of typing relations miss a clean semantic intuition of types. Consider for example the type system defined by Hennessy and Riely [7], which is one of the most advanced type systems for variants of π-calculus that includes read-only and write-only channels as well as union and intersection types. In the systems the following equality holds:

$$ch^{+}(\mathtt{int}) \vee ch^{+}(\mathtt{bool}) = ch^{+}(\mathtt{int} \vee \mathtt{bool}) \qquad (1)$$

where $ch^{+}(t)$ is the type of channels from which we can only read values of type t and $\vee$ denotes union. We would like to understand the precise semantic intuition that underlays an equation such as (1). To do this, we first have to provide a semantic account of channels.

Channels as boxes. 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 characterized by the set of the things the box can contain. Thus a channel of type $ch^{+}(t)$ is a box in which we must expect to find something of type t and, similarly, a channel of type $ch^{-}(t)$ is a box in which we can put only something of type t. But if one takes this standing, the equality above 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 of a piece of charcoal, but we know it is always one of the two. For example if we use the box two times, the second time we will know what it contains. A box of the second type, instead, can always give us both candies and charcoal. Our intuition suggests that the two types above are quite different because they characterize two different kinds of objects.

The role of the language. So why did Hennessy and Riely require (1)? The point is that if in the language under consideration there is no syntactic construction that can tell apart a $ch^{+}(\mathtt{int})$ channel from a $ch^{+}(\mathtt{bool})$ channel, then it is not possible to operationally observe any difference between the types in (1). On the contrary, if it is possible to test whether on a channel c we are receiving a channel of type $ch^+(\texttt{int})$ or a channel of type $ch^+(\texttt{bool})$, then a rule such as (1) would give rise to an unsound system because it would allow to have in c a channel of type $ch^+(\texttt{int} \vee \texttt{bool})$ which makes the test on c crash (since the possibility that the argument is a $ch^+(\texttt{int} \vee \texttt{bool})$ box is not contemplated). Thus, in case we can check the type of received channels, the right relation, supported by our semantic intuition, would be

$$ch^+(\texttt{int}) \vee ch^+(\texttt{bool}) \subsetneq ch^+(\texttt{int} \vee \texttt{bool}) \qquad (2)$$

because we can always safely use a box that contains only integers—or one that contains booleans—where a box that can contain both is expected (this is just the usual covariance of the input type), while, as we just argued, the converse does not hold.

Since the language considered by Hennessy and Riely is not expressive enough to distinguish channels according to the types they transport, then it is sound to impose equations such as (1) or alike, and this dramatically simplifies the definition of the subtyping algorithm (cf. Section 2.5). However the restriction brought by (1) is only justified by a weakness of the language; it is not grounded on any semantic basis.

Semantic subtyping. The aim of this work is to define a very expressive type system for π-calculus whose definition is based on a clear semantic interpretation. The type system will allows us to specify read-only and write-only channels, but will also permit intersections, unions, and negations of types. Afterwards, we will also add product, functional and (a limited form of) recursive types.

The basic idea is simple, even though is technically hard to implement: to characterize a type system (at least from an operational point of view) we do not need a full theory of type equivalence, the definition of the subtyping relation suffices. Therefore if we want to ground the type system with a semantic intuition it "suffices" to define the subtyping relation semantically. We shall give a set theoretic interpretation of the types of our system and define the subtyping relation as the inclusion of interpretations. In other terms if $[\![.]\!]$ is an interpretation function from types to sets, then we define $s \leq t$ if and only if $[\![s]\!] \subseteq [\![t]\!]$. The characterization of intersection, union, and negation types then comes for free, while for the interpretation of channel types it is possible to rely on the semantic intuition of "channel as boxes".

We have also seen that there is a tight relation between the types and the language they are used for, therefore we will also define a variant of π-calculus that exploits the full power of our new types, and in particular that permits dynamically testing the type of values received on a channel. We will implement the dynamic test by endowing input actions with patterns, and allowing synchronization 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 function types that permit manipulations of channels via higher-order functions that can be transmitted over channels.

Advantages of a semantic approach. The main advantage of using a semantic approach is that the types have a natural set theoretic interpretation: types can be thought of as the sets of all their values, and union intersection and negation types are understood in terms of the corresponding set theoretic operations. This property turns out to be very helpful not only to apprehend the meaning of the types but also to reason on them. Thus, 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, 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 [1] also demonstrated the practical impact of the semantic approach: not only subtyping results are easier to understand for a programmer, but also the compiler/interpreter can return much more precise and meaningful error messages. So 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 value provides information to 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 paper [4].

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, recursive types, and complete boolean combinations of types which are interpreted as the corresponding set-theoretic operations. Two strictly related contributions are the definition of a set-theoretic denotational model for the types above and the interpretation of channel types as set of boxes. We also show how to extend the π-calculus in order to fully exploit the expressiveness of the type system, and in particular with input actions with pattern matching *à la* ℂDuce. 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 yielding a calculus with dynamic type dispatch, overloading,channelled communications and where both functions and channels have first class citizenship.

Finally, we think that the most important, although the most debatable and controversial, contribution of this work is that it shows the effectiveness of the semantic approach of subtyping: we believe that the generality of our subtyping relation, and the definition of the subtyping algorithm summarized by Proposition 2.10 could have been very hardly achieved via a syntactic and proof theoretic approach.

Related work. The first work on subtyping for π was done by Pierce and Sangiorgi [9] and successively extended in several other works [11, 5, 12].

The work closest to ours, at least for the expressiveness of the types, is the already cited work of Hennessy and Riely [7]. For what concerns π-types, our work subsumes their system in the sense that it defines a richer subtyping relation; this can be checked by noting that their type $\mathsf{rw}\langle s,t\rangle$ corresponds to the intersection $ch^{+}(s)\wedge ch^{-}(t)$ of our formalism.

Brown *et al.* [3] enrich π with XML-like values that are deconstructed by pattern matching. The patterns they use are quite different from the one we introduce here as they work exclusively on the structure of the matched values but not on their types. Furthermore they also have patterns to match the interleaving of values, that we do not consider. On the other hand they do not consider types, which are the main motivation of our work.

For what concerns the technical issues of semantic subtyping our starting point is the work developed by Frisch *et al.* for functional programming languages [6], that led to the design of ℂDuce [1].

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

2 Types and patterns

For the sake of the presentation we would like to introduce our system gradually. Therefore, we shall start with a relatively simple system with just base types, channels and boolean combinators. In a second moment, we shall add the product type constructor and recursive types. Finally, we will consider 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, booleans, etc...), the input or the output channel type constructor, or by applying a *boolean combinator*, i.e., union, intersection, and negation:

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

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

Although types $ch(t)$ are just syntactic sugar, they will play a crucial role in the rest of the paper. In particular, we shall see that the types of the form $ch(t)$ are all and the only types that are not base types and that denote a singleton. We shall use them quite often because they are the most precise type of channels (see, e.g., the typing rule (chan) in Section 2.1).

Even if in this section we are mainly interested in types, it is necessary to make a digression and consider values; it is impossible to formalise the semantics of the former without considering the latter. In our approach channels are values, that is physical boxes where one can insert and withdraw objects of a given type. Our intuition, somehow departing from the usual intuition about channels, is that there is not such a thing as a read-only or write-only box: each box is associated to a type t and one can always write and read objects of that type into and from such a box. Thus the type of $ch^{+}(t)$ can be considered just a constraint that tells that a variable of that type will be bound only to boxes from which one can read objects of type t; and therefore to all the boxes that have type $s\leq t$. Notice that if we know that a message has type $ch^{+}(t)$ this *does not* mean that we cannot write into it, we simply know that we do not have any information about what can be written in it: for instance this message could be a box of type $ch(\mathbf{0})$ therefore a box in which nobody can write anything. Thus, we must 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 channel of type $ch(\mathbf{1})$. Therefore we better avoid reading from it, unless we are ready to get anything. In case we are ready to get anything, our type system will guarantee that we *can* read on a channel with type $ch^{-}(t)$ because we have $ch^{-}(t)\leq ch^{+}(\mathbf{1})$.

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

Now we are able to define more formally the semantics of our types. Our leading intuition is that a type represents the set of values of that type. This allows us to define the subtyping relation simply as set inclusion. The basic types should be interpreted on a set of basic values (integers, booleans). 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 contain objects of t:

$$[\![ch(t)]\!] = \big\{c \mid c \text{ is a box for objects of type } t\big\} . \qquad (3)$$

Remember that one of the main reasons for giving a semantics to types is to define a subtyping relation. Intuitively this relation is insensitive to the number boxes associated to a given type (as long as there is one). In this sense it is not unreasonable to say that a box of type $ch(t)$ "is" the set of values that it can contain, viz. the (denotation of the) type t: [1]

$$[\![ch(t)]\!] = \Big\{[\![t]\!]\Big\} \qquad (4)$$

Or, to state it otherwise from the viewpoint of types all the boxes are indistinguishable since they can all be identified with the set of all the values they can contain.[2]

Starting from the above interpretation of $ch(t)$, it is now rather straightforward to provide a semantics for $ch^+(t)$ and $ch^-(t)$. The former denotes the set of all boxes from which one can take an object of type t, thus it denotes "the" box containing objects of type t but also all the boxes containing objects of type $s \leq t$ (by subsumption these objects are also of type t). The latter denotes the set of all boxes in which one can put objects of type t, therefore all the boxes that can contain objects of type $s \geq t$ (once more, by subsumption, it can be deduced that an object of smaller type can be fitted where an object of larger type is expected). Formally, we have

$$[\![ch^+(t)]\!] = \Big\{[\![t']\!] \mid t' \leq t\Big\} , \qquad [\![ch^-(t)]\!] = \Big\{[\![t']\!] \mid t' \geq t\Big\} .$$

We have that this semantics induces invariance of channel types, 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 side of the equality have the same semantics. Our proposed interpretation has other interesting features that we shall describe later. In the meanwhile, the reader who wants to familiarise with our semantics can try to use the above definitions to verify that the difference $ch^+(t) \setminus ch^-(t)$ and the difference $ch^+(t) \setminus ch(t)$ have the same interpretation, and that the same holds for $ch^+(\mathbf{1})$ and $ch^-(\mathbf{0})$.[3]

We want now to build a model for the types, that is a set $\mathscr{D}$ such that the denotation of every type is a subset of $\mathscr{D}$. Note that the semantics of channel types seems to require that subsets of $\mathscr{D}$ "be" elements of $\mathscr{D}$, which would lead us to

$$\mathscr{P}(\mathscr{D}) \subseteq \mathscr{D} .$$

This is not possible for cardinality reasons. We devote the next section to solve this problem.

2.2 Extensionality and models

We remind the reader that the main reason why we need a semantic interpretation of types is for defining a subtyping relation. Indeed, we do not introduce an interpretation of types in order to state *what types are*, but rather to define *how types are related*. Therefore, we do not need to require that the interpretation of, say, $[\![ch^+(t)]\!]$ is equal to $\{[\![t']\!] \mid t' \leq t\}$, it suffices that our interpretation function induces the same containment relation as the one obtained when types are interpreted as described above, and therefore that we have: $[\![ch^+(s)]\!] \subseteq [\![ch^-(t)]\!]$ if and only if $\{[\![t']\!] \mid t' \leq t\} \subseteq \{[\![s']\!] \mid s \geq s\}$.

In other terms, we do consider correct every interpretation function that behaves *like* the interpretation of the previous section with respect to containment. This is formally stated by the definition of *model* below.

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

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

A pre-model thus simply requires that the interpretation of the type combinators is set-theoretic.

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

- $\mathscr{E}[\![b]\!] = [\![b]\!]$;
- $\mathscr{E}[\![\mathbf{1}]\!] = \mathbb{B} + \mathscr{P}(\mathscr{D})$, $\mathscr{E}[\![\mathbf{0}]\!] = \varnothing$;
- $\mathscr{E}[\![\neg t]\!] = \mathscr{E}[\![\mathbf{1}]\!] \setminus \mathscr{E}[\![t]\!]$;
- $\mathscr{E}[\![t_1 \vee t_2]\!] = \mathscr{E}[\![t_1]\!] \cup \mathscr{E}[\![t_2]\!]$, $\mathscr{E}[\![t_1 \wedge t_2]\!] = \mathscr{E}[\![t_1]\!] \cap \mathscr{E}[\![t_2]\!]$;
- $\mathscr{E}[\![ch^+(t)]\!] = \{[\![t']\!] \mid [\![t]\!]' \subseteq [\![t]\!]\}$;
- $\mathscr{E}[\![ch^-(t)]\!] = \{[\![t']\!] \mid [\![t']\!] \supseteq [\![t]\!]\}$.

[1] We can also have boxes of type $ch(\mathbf{0})$ that cannot contain any object. They can still be passed around as tokens.

[2] More prosaically, since every box is associated to a unique type, then no box can belong to two distinct $ch()$ types. This implies that the intersection of distinct $ch()$ types is always empty and therefore (3) and (4) induce exactly the same subtyping relation, which is all that matters. Thus the reader can consider the interpretation in (4) as a more compact and convenient way to represent (3).

[3] We find it useful to think $[\![ch^+(t)]\!]$ and $[\![ch^-(t)]\!]$ respectively as the downward and upward cones starting from t. The difference $s \setminus t$ is defined as $s \wedge \neg t$.

Function $\mathscr{E}[\![\,]\!]$ behaves exactly as $[\![\,]\!]$ apart from the top-level channel constructors which are interpreted according to the semantics outlined in Section 2.1. Therefore, we shall consider an interpretation function "acceptable" if it induces the same containment relation as its extensional interpretation. That is, we say that a pre-model is a *model* if for every type t_1, t_2, we have $[\![t_1]\!] \subseteq [\![t_2]\!]$ if and only if $\mathscr{E}[\![t_1]\!] \subseteq \mathscr{E}[\![t_2]\!]$.

Notice that the boolean combinators allow us to define subtyping in terms of emptiness: $[\![t_1]\!] \subseteq [\![t_2]\!]$ if and only if $[\![t_1 \wedge \neg t_2]\!] = \varnothing$ and the same for the extensional interpretation. This justifies the following definition of model[4]

Definition 2.3 (Model) *A pre-model $(\mathscr{D}, [\![\,]\!])$ is a* model *if for every type t, $[\![t]\!] = \varnothing \iff \mathscr{E}[\![t]\!] = \varnothing$.*

2.3 A specific model

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. We shall sketch here the construction of such model, while omitting the proofs and technical details that can be found in the appendix.

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

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

Then we set

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

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

Now suppose we have a pre-model $\mathscr{D}_n$ for $\mathscr{T}_n$, with corresponding pre-order $\leq_n$ and equivalence $=_n$. We call $\widetilde{\mathscr{T}_n}$ the set of equivalence classes $\mathscr{T}_n/_{=_n}$. Then we let $\mathscr{D}_{n+1}$ to be such that

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

with the following interpretation of channel types:

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

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

Proposition 2.4 *Let $t, t' \in \mathscr{T}_n$ and $k, h \geq n$, then $t \leq_k t'$ if and only if $t \leq_h t'$.*

Hinging on this observation we define pre-order between types as follows.

Definition 2.5 (Order) *Let $t, t' \in \mathscr{T}_n$, then $t \leq_\infty t'$ if and only if $t \leq_n t'$.*

Due to Proposition 2.4, 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 pre-order 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 pre-orders, i.e. $\leq_\infty$.

Proposition 2.6 *Let $t, t' \in \mathscr{T}$, then $t \leq t'$ if and only if $t \leq_\infty t'$.*

It is now easy to show the following.

Theorem 2.7 *The pre-model $(\mathscr{D}, [\![\,]\!])$ is a model.*

2.4 Examples of type (in)equalities

We use the equal symbol on types to denote equality of denotations: $s = t \overset{def}{\iff} [\![s]\!] = [\![t]\!]$. We list here some interesting equations and inequations between types that can be easily derived from the set-theoretic interpretation of types.

$$ch(t) \leq ch^-(\mathbf{0}) = ch^+(\mathbf{1}) \tag{5}$$

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

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

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

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

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

Union, as we observed in the introduction, behaves differently.

$$\begin{aligned} ch^+(t_1) \vee ch^+(t_2) &\leq ch^+(t_1 \vee t_2) \\ ch^-(t_1) \vee ch^-(t_2) &\leq ch^-(t_1 \wedge t_2) \end{aligned}$$

[4] Indeed, $[\![t_1]\!] \subseteq [\![t_2]\!] \iff [\![t_1]\!] \setminus [\![t_2]\!] = \varnothing \iff [\![t_1 \wedge \neg t_2]\!] = \varnothing \iff \mathscr{E}[\![t_1 \wedge \neg t_2]\!] = \varnothing \iff \mathscr{E}[\![t_1]\!] \setminus \mathscr{E}[\![t_2]\!] = \varnothing \iff \mathscr{E}[\![t_1]\!] \subseteq \mathscr{E}[\![t_2]\!]$.

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

$$ch^{+}(t_1) \wedge ch^{-}(t_2) = \mathbf{0} \tag{8}$$

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

2.5 Decidability of Subtyping

We can now use the semantic characterisation of the types to derive a decision algorithm for the subtyping relation. We do it in two steps: first, we show how to express the problem of subtyping two types into a set of subtyping problems on types of smaller heights, then we prove the decidability of these smaller problems and deduce the decidability of subtyping. The techniques of this section give a good idea of the advantages of having a set-theoretic definition of subtyping.

Simplification of the subtyping

First of all note that the subtyping problem is equivalent deciding the emptiness of a type.

$$s \leq t \iff s \wedge \neg t = \mathbf{0} \tag{9}$$

which can be derived as follows:

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

Thanks to the semantic interpretation we can directly apply set-theoretic equivalences to types (in the rest of the paper we will do it without explicitly passing via the interpretation function) and deduce that every type can be represented as the union of addenda of uniform sort. Since a union is empty only if all the addenda are empty, then in order to decide the emptiness of a type—and in virtue of (9) decide subtyping—it suffices to be able to decide whether

$$(\bigwedge_{i\in P} b_i) \wedge (\bigwedge_{j\in N} \neg b_j) \quad \text{and} \quad (\bigwedge_{i\in P} ch^{\mathrm{v}_i}(t_i)) \wedge (\bigwedge_{j\in N} \neg ch^{\mathrm{v}_j}(t_j))$$

are equivalent to **0**. The decision of emptiness of the left-hand side depends on the basic types that are used. For what concerns the right-hand side, the algorithm must decompose this problem into simpler subproblems. More precisely, it must decompose the problem of subtyping boolean combinations of channel types into a set of problems of subtyping types that form (are strict sub-occurrences) these channel types. This can be done by using some general algebraic equivalences combined with the properties of the semantic interpretation. This turns out to be quite complicated, in particular since it involves the use of *atoms*

Definition 2.8 (Atom) *An* atom *is minimal nonempty type.*

Atoms thus are types "just above" the empty type. In what follows the reader can think of atoms as types whose denotation is a singleton set (even though this is not accurate, see for instance the interpretation of channels in the value model of Section 3.2).

The problem is to decide when the following holds:

$$(\bigwedge_{i\in P} ch^{\mathrm{v}_i}(t_i)) \wedge (\bigwedge_{j\in N} \neg ch^{\mathrm{v}_j}(t_j)) = \mathbf{0}\ .$$

Using set-theoretical manipulations, this is equivalent to check that

$$(\bigwedge_{i\in P} ch^{\mathrm{v}_i}(t_i)) \leq (\bigvee_{j\in N} ch^{\mathrm{v}_j}(t_j)) \tag{10}$$

Because of equations (6) and (7), we can push the intersection on the left-hand side inside the constructors and reduce (10) to the case

$$ch^{+}(t_1) \wedge ch^{-}(t_2) \leq \bigvee_{h\in H} ch^{+}(t_3^h) \vee \bigvee_{k\in K} ch^{-}(t_4^k) \tag{11}$$

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 (11) by eliminating:

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

Then the key property for decomposing the problem (11) into simpler subproblems is given by the following theorem:

Theorem 2.9 (channel inclusion) *Suppose $t_1, t_2, t_3^h, t_4^k \in \mathscr{T}$, $k \in K$, $h \in H$. Suppose moreover that the following conditions hold:*

c1. for all distinct $h, h' \in H$, $t_3^h \not\leq t_3^{h'}$;

c2. for all distinct $k, k' \in K$, $t_4^k \not\leq t_4^{k'}$;

c3. for all $h \in H$ $t_2 \leq t_3^h$;

c4. for all $k \in K$ $t_4^k \leq t_1$.

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

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

if and only if one of the follow conditions holds

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

The four hypotheses c1–c4 simply state that the right-hand side of the inequation was simplified according to the rules stated 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 t_3^h and, respectively, one of the t_4^h contains $ch^+(t_1) \wedge ch^-(t_2)$. Finally the fourth and more involved condition (CA) says that, every time we add atoms to t_2 so that we are no longer below any t_3^h, then we must end up above some of the t_4^k.

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

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

It is easy to see that

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

since, for example, the type $ch(t_2 \vee \mathtt{err}_1)$ is a subtype of the left-hand side, but not of the right-hand side. However if $\mathtt{err}_1 = \mathtt{err}_2$, the subtyping relation holds, because of condition (CA). Indeed in that case the indexing set H of Theorem 2.9 is a singleton. The only $I \subseteq \mathscr{P}(H)$ such that $\bigcup I = \varnothing$ is $\{\varnothing\}$. The type e_I is $t_1 \wedge \neg t_3$. The only atom in it is $\mathtt{err}_1$, and it is true that $t_4 \wedge \neg t_2 \leq \mathtt{err}_1$.

Finally note that, as announced, Theorem 2.9 decomposes the subtyping problem of (11) into a finite set of subtyping problems on types of smaller heights (we must simplify the inequation RHS by verifying the inequalities of conditions c1–c4, and possibly perform the $|H| + |K| + 1$ checks for LE, R1 and R1) *and* into the verification of condition (CA). Now it is clear that the inequation (11) is decidable—and so is the subtyping relation—if and only if (CA) is decidable (modulo the decidability of subtyping on the base types of course).

Decidability

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

Proposition 2.10 *If we replace condition (CA) with*

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

then Theorem 2.9 still holds.

Therefore it suffices to check the condition just for the e_I that are finite (that is that are equal to a finite union of atoms), which can be done by an algorithm provided that we are able to:

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

Surprisingly this is possible, and since this is at the core of the subtyping algorithm we think it is worth to show *in extenso* why this is so. To prove our claim we proceed by induction on the height of the types. We strengthen the statement by requiring that all atoms of a finite type t have the same height, or lower, of t. We assume that at height 0, this is the case. It is a reasonable assumption: for example it is the case if we have for base types the type of all integers plus all constant types. Consider a type t of height $n+1$ and assume that for lower heights we can decide whether a type is finite and, if it is the case, list all its atoms. By Theorem 2.9, 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. We assumed that this is possible for basic atoms, therefore we still should check for the atoms of the form $ch(s)$, since these are the only atoms contained in non-base types. For how many s we can have that $ch(s) \leq t$? If we put t in normal form, we obtain the disjunction of terms of the form

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

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 I 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 $n+1$.

We have thus

Lemma 2.11 (Main) *There is an algorithm that decides whether a type t is finite and if it is the case, outputs all its atoms.*

Corollary 2.12 (Decidability) *If it is possible to decide (i) if a base type is finite and in that case list all its atoms and (ii) if it is a subtype of another base type, then the subtyping relation is decidable.*

Finally, once decidability is established note that the first half of the section formally described the subtyping algorithm to check whether $s \leq t$, which can be summarised as follows:

1. put $s \wedge \neg t$ in disjunctive normal form;
2. check emptiness for base types; if it does not holds, then return false else
3. Simplify the summand on channels so that it has the same form as (11) and satisfies the conditions c1–c4 of Theorem 2.9
4. Check the conditions LE, R1, R2, and CA, and return whether one of them is satisfied.

We do not discuss here the complexity of this algorithm, nor the possibility of finding more efficient ways of doing it. We leave it for future work.

2.6 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.

In order to obtain it the simplest way is to add to the π-calculus a process that dynamically tests whether the message M is of type t or not. Quite informally, this would correspond to adding the following process

$$[M:t]P$$

where M denotes a message (that is either a *value* or a *variable*) and whose behaviour intuitively is as follows:

$$\begin{array}{lcll} [v:t]P & \longrightarrow & P & \text{if the value } v \text{ is of type } t \\ [v:t]P & \longrightarrow & \mathbf{0} & \text{if the value } v \text{ is not of type } t \end{array}$$

In this work we want to introduce a more ambitious extension of π-calculus that will subsume the one above. So instead of adding an explicit type-case process as the above we embed type-cases directly in the communications by endowing input actions with $\mathbb{C}$Duce patterns. The reason why we rather do that is that the semantic subtyping framework nicely fits patterns since the semantics of patterns can be defined independently from the language or calculus they are going to be used in, but just relying on the notion of model, as we show next.

Definition 2.13 (Pre-patterns) *Given a type algebra $\mathscr{T}$, and a set of variables $\mathbb{V}$, a pre-pattern p on $(\mathbb{V},\mathscr{T})$ is a possibly infinite term p generated by the following grammar*

$$\begin{array}{lcll} p & ::= & x & \text{capture, } x \in \mathbb{V} \\ & | & t & \text{type constraint, } t \in \mathscr{T} \\ & | & p_1 \wedge p_2 & \text{conjunction} \\ & | & p_1 | p_2 & \text{alternative} \\ & | & (x := n) & \text{constant, } n \in \mathbb{B} \text{ with } [\![b_n]\!] = \{n\} \end{array}$$

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

Definition 2.14 (Patterns) *Given a type algebra $\mathscr{T}$, and a set of variables $\mathbb{V}$, a pre-pattern p on $(\mathbb{V},\mathscr{T})$ belongs to the set of (well-formed) patterns $\mathbb{P}$ on $(\mathbb{V},\mathscr{T})$ if and only if it satisfies the following condition: for every subterm $p_1 \wedge p_2$ of p we have $Var(p_1) \cap Var(p_2) = \varnothing$, and for every subterm $p_1|p_2$ of p we have $Var(p_1) = Var(p_2)$.*

These patterns and their semantics are borrowed from [6]: the reader can refer to [6, 1] for a detailed description. When a pattern is matched against an element of the domain it returns either a substitution for the free variables of the pattern, or a failure, denoted by Ω:

Definition 2.15 (Semantics of pattern matching) *Given $d \in \mathscr{D}$ and $p \in \mathbb{P}$ the matching of d with p, denoted by d/p, is the element of $\mathscr{D}^{Var(p)} \cup \{\Omega\}$ defined by induction on structure of p as follows:*

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

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:

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

In short a variable pattern always matches and captures the matched element with the variable; a type pattern matches only the elements that belong to the interpretation of the type but does not capture them; a conjunction pattern matches only if both pattern (which must use different sets of variables) match and returns the concatenation of the two substitutions (denoted by $\otimes$); the alternative pattern tries to match the first pattern and if it fails, it tries the second one, while the constant pattern always succeed by returning the constant substitution.

One of the remarkable properties 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 $\llparenthesis p \rrparenthesis$. In other terms, for every (well-formed pattern), there exists a unique types $\llparenthesis p \rrparenthesis$ such that $[\![\llparenthesis p \rrparenthesis]\!] = \{d \in \mathit{Dom} \mid d/p \neq \Omega\}$. Not only, but this type can be calculated. Similarly, consider a pattern p and a type $t \leq \llparenthesis p \rrparenthesis$, 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 2.16 *There is an algorithm mapping every pattern p to a type $\llparenthesis p \rrparenthesis$ such that $[\![\llparenthesis p \rrparenthesis]\!] = \{d \in \mathscr{D} \mid d/p \neq \Omega\}$.*

Theorem 2.17 *There is an algorithm mapping every pair (t,p), where p is a pattern and t a type such that $t \leq \llparenthesis p \rrparenthesis$, to a type environment $(t/p) \in \mathscr{T}^{\mathit{Var}(p)}$ such that $[\![(t/p)(x)]\!] = \{(d/p)(x) \mid d \in [\![t]\!]\}$.*

The proofs can be found in [6], but to give an idea here they are the inductive definitions of $\llparenthesis p \rrparenthesis$ and (t/p):

$$\begin{array}{lcl}
[\![\llparenthesis t \rrparenthesis]\!] & = & [\![t]\!] \\
[\![\llparenthesis (x := n) \rrparenthesis]\!] & = & [\![\llparenthesis x \rrparenthesis]\!] = \mathscr{D} \\
[\![\llparenthesis p_1 \wedge p_2 \rrparenthesis]\!] & = & [\![\llparenthesis p_1 \rrparenthesis]\!] \cap [\![\llparenthesis p_2 \rrparenthesis]\!] \\
[\![\llparenthesis p_1 | p_2 \rrparenthesis]\!] & = & [\![\llparenthesis p_1 \rrparenthesis]\!] \cup [\![\llparenthesis p_2 \rrparenthesis]\!]
\end{array}$$

$$\begin{array}{lcll}
(t'/x)(x) & = & t' & \\
(t'/p_1|p_2)(x) & = & ((t' \wedge \llparenthesis p_1 \rrparenthesis)/p_1)(x) \vee & \\
 & & ((t' \wedge \neg \llparenthesis p_1 \rrparenthesis)/p_2)(x) & \\
(t'/p_1 \wedge p_2)(x) & = & (t' \wedge \llparenthesis p_2 \rrparenthesis / p_1)(x) & \text{if } x \in \mathit{Var}(p_1) \\
(t'/p_1 \wedge p_2)(x) & = & (t' \wedge \llparenthesis p_1 \rrparenthesis / p_2)(x) & \text{if } x \in \mathit{Var}(p_2) \\
(t'/(x := n))(x) & = & b_n & \text{if } t' \neq \mathbf{0} \\
(t'/(x := n))(x) & = & \mathbf{0} & \text{if } t' = \mathbf{0}
\end{array}$$

3 The semantic-π calculus

3.1 Syntax

The syntax of our calculus is very similar to that of the asynchronous π-calculus, a variant of the π-calculus where message emission is non-blocking. We have chosen this calculus as our starting point, because of its simplicity and expressivity. It is generally considered as the calculus representing the essence of name passing with no redundant operation. The only operators of asynchronous π-calculus are the empty process, non-blocking output, blocking input prefix, parallel composition and replication, the exchanged values of the calculus are just names. The variant we consider is very similar to the basic calculus, we only permit patterned input prefix and guarded choice between different patterns on the same input channel.

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

where I is a possibly empty finite set of indexes and t ranges over the types defined in Section 2.1.

The constructs for processes we adopt are mostly standard. As customary we use the convention that the empty sum corresponds to the inert process, usually denoted by 0. We only want to comment on the presence of the simplified form of summation we have adapted: guarded sum of inputs on a single channel with possibly different patterns. A long standing debate is going on in the concurrency community about the usefulness of summation operators that permit choosing between different continuations. Choice operators are indeed very useful for specifying nondeterministic behaviours, but give rise to problem when considering implementation issues. Different kinds of choices have to be considered: *external choice* that leaves the decision about the continuation to the external environment (usually depending on the channel used by the environment to communicate) and *internal choice* that is performed by the process regardless of external interactions. The former type of choice is difficult to implement in presence of distribution (consider modelling $P \parallel Q + R$), thus often only guarded choices are considered; internal nondeterminism pops up as soon as two of input prefix use the same channel. Thanks to patterns we can offer an externally controllable choice that can be easily implemented by relying on pattern matchings; the received message, not the used channel, will determine continuation. Internal choice can be modelled by specifying processes that perform input on the same channel according to the same pattern.

The other, more important, difference with standard asynchronous π-calculus is that typed channels 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 typed channels also "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

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

We use $\mathscr{V}$ to the denote the set of all values. Every value is associated to a type: a constant is associated to a basic type b_c and a channel value 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

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

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

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

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

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

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

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

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.

Figure 2: Context and congruence closure

Figure 1 (actually with an empty Γ) where in the (subs) subsumption rule the $\leq$ is the subtyping relation induced by the model of Section 2.3.

3.2 Semantics

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

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

This interpretation satisfies the conditions of model of Section 2.2 and furthermore it generates the same subtyping relation as $\leq$.

Proposition 3.1 *Let $[\![t]\!]_{\mathscr{V}} = \{v \ | \ \Gamma \vdash v : t\}$. Then $(\mathscr{V}, [\![\,]\!]_{\mathscr{V}})$ is a model and $s \leq t \iff [\![s]\!]_{\mathscr{V}} \subseteq [\![t]\!]_{\mathscr{V}}$.*

The first point of the proposition states that a value v is also an element of a model of the types whose domain is $\mathscr{V}$, therefore Definition 2.15 applies for d being a value. We can thus use this 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 the substitution s to the 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 these is non-deterministically chosen and and the corresponding process executed after that the pattern variables are replaced by the captured values.

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

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

– communication is subjected to pattern matching
– communication can happen only along values (i.e. boxes)

First of all note that these two points are not restrictive. Every asynchronous π-calculus process is also a process of our calculus and with the same reduction semantics: it suffices to consider all free and restricted variables (thus excluding those that are bound in an input actions, which according to our viewpoint are "real" variables) to be typed channels of type $ch^+(\mathbf{1})$ (or $ch^-(\mathbf{0})$ since they both denote the set of all channels). So we do not lose any generality with respect to the π-calculus.

The use of pattern matching is what makes 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 now we have this distinction between variables and typed channels it is then reasonable to require the communication to be performed only if we have a physical channel that can be used as a support for it, and thus forbid synchronisation if the channel still is 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 $\Gamma \vdash x : ch^-(\texttt{bool})$. Then according to the typing rules of our system (see later on) the process

$$\overline{x}\texttt{true} \,\|\, x(y).\overline{x}(y\texttt{+}y)$$

is well typed, in the environment but ends up in a run time error since it tries to sum `true` with `true`:

$$\overline{x}\texttt{true} \,\|\, x(y).\overline{x}(y\texttt{+}y) \quad \longrightarrow \quad \overline{x}(\texttt{true+true})$$

This reduction cannot happen in our calculus, because we can never instantiate a variable of type $\mathbf{0}$.

3.3 Typing

In Figure 1 we summarise typing rules that ensure 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 types. The rules for replication and parallel composition are standard. In the rule for output we check that the message is compatible with the type of the channel. The rule for restriction is slightly different since we do not need to store in the type environment the type of the channel[5].

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 2.17) and check whether under this environment the summand process

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

Messages

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

Processes

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

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

Figure 1: Typing rules

P_i. This is all it is needed to have a sound type system. However the input construct is like a typecase/matching expression, so it seems reasonable to perform a check that pattern are exhaustive and there is no useless case[6] This is precisely what the two side conditions of (input) do:

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

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

Of course we could have used a different type system and/or reduction semantics to define more refined policies (best match, first match) that can remove all remaining nondeterminism. We did not do it since as we hint in the next section they can be easily encoded thanks to the expressive power of our patterns/types.

As usual the basic result is the subject reduction, which is preceded by a substitution lemma.

Lemma 3.2 (Substitution)
– *If* $\Gamma, t/p \vdash M' : t'$ *and* $\Gamma \vdash v : t$, *then* $\Gamma \vdash M'[v/p] : t'$.
– *If* $\Gamma, t/p \vdash P$ *and* $\Gamma \vdash v : t$ *then* $\Gamma \vdash P[v/p]$.

Lemma 3.3 (Congruence)
If $\Gamma \vdash P$ *and* $P \equiv Q$, *then* $\Gamma \vdash Q$.

Theorem 3.4 (Subject reduction)
If $\Gamma \vdash P$ *and* $P \to P'$ *then* $\Gamma \vdash P'$.

It is far from being obvious that the decidability of the subtyping relation implies the decidability of the typing relation (only semidecidability is straightforward). The typing algorithm can be derived in a standard way by eliminating the subsumption rule and embedding the subtyping checks into the elimination rules. Though, it is not so evident that the type system satisfies the minimum type property, and this is because of the (input) rule, which in its algorithmic version requires us to compute the least type of the form $ch^{+}(s)$ that is an upper bound of a given type t. Now observe that our type algebra is *not* a complete lattice (since least upper and greatest lower bounds exist only for finite sets). Nevertheless such a type exists and is unique (which gives gives the minimum typing property) and furthermore it can be effectively calculated.

[6]While in functional programming these check are necessary for soundness since an expression non-complying to them may yield a type-error, in process algebra non-compliance would just block synchronisation.

Theorem 3.5 (Upper bound channel) *For every type t there exists a least type $ch^{+}(s)$ that is an upper bound of t and an algorithms that computes it.*

3.4 Examples

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

$$\sum_{i=1..n} \alpha(p_i).P_i \tag{12}$$

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

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

Then the process

$$\sum_{\{i \mid \llparenthesis q_i \rrparenthesis \wedge t \neq \mathbf{0}\}} \alpha(q_i).P_i \tag{13}$$

behaves exactly as the above with the only difference that summand selection is deterministic and obeys a first matching discipline. Indeed, every pattern accepts only the values that are not accepted by the preceding patterns. Note that by applying a first match policy some of the summand could no longer have any chance to be selected (this happens if $\llparenthesis p_i \rrparenthesis \wedge t \leq \bigvee_{j<i} \llparenthesis p_j \rrparenthesis$), and therefore they must not be included in (13) since then it would not be well typed (there would be redundant summands), which explains the set used to index the sum.

Best match policy. It is possible to rewrite the process in (12) so that it satisfies a best matching policy. Of course this is possible only if for every possible choice in (12) there always exist a best-matching pattern[7]. If this is the case then with the following definition for q_i's

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

the process (13) is well-typed and implements the best matching policy for (12), since the difference in the definition of q_i makes the pattern fail on every value for which there exists a more precise pattern that can capture it.

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

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

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

As we discussed in the Introduction this distinction 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. In that case we would need to write P as

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

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

4 Extensions

4.1 Polyadic version

The first extension we consider consists in adding the product to our type constructors. This will require the extension of the notion of pattern, but most importantly, it will affect the definition of subtyping. The new syntax for the types is as follows

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

The definition of height is extended by:

– $\hbar(t_1 \times t_2) = \max(\hbar(t_1), \hbar(t_2))$.

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

Messages are extended by

$$\textit{Messages} \quad M \quad ::= \quad \dots \quad \mid \quad (M,M) \quad \text{pair}$$

The patterns are extended by

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

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

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

4.2 Recursive types

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

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

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

Why is that? The reason is that without this restriction is not possible to find a model. To see why, we observe that we could have a recursive type t such that

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

for some nonempty base type b. If we have a model, either $t = b$ or $t \neq b$. Does $t = b$? Suppose it does, 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 nesting of the channel constructor there are, and we can still define a function $\hbar(t)$ as the maximum number of channel constructors which appear on a branch of t. (The König's lemma guarantees that $\hbar(t)$ is finite).

This stratification also allows us to construct the model using the same ideas presented in Section 2.

The traditional example of the use of a recursive type is "self application", that is a channel that can carry itself. It is regrettable that our semantics prevents us from defining recursive types involving channel constructor, but we can still type self application by using, for instance, the type $ch(\mathbf{1})$: a channel that can carry everything, can clearly carry itself.

4.3 The extended model

The addition of product types requires minimal modifications to the definition of pre-model, that is, that the interpretations of different type constructors are pairwise disjoint, namely for all s, t, and t'

$$[\![ch^{\mathsf{v}}(s)]\!] \cap \mathbb{B} = \varnothing \quad [\![t \times t']\!] \cap \mathbb{B} = \varnothing \quad [\![ch^{\mathsf{v}}(s)]\!] \cap [\![t \times t']\!] = \varnothing$$

The extensional interpretation must be extended in order to take into account the intuitive semantics of product types, that is, the product of the interpretations:

Definition 4.1 *Let* $(\mathscr{D}, [\![\,]\!])$ *be a pre-model. The* extensional *interpretation of the types is the function* $\mathscr{E}[\![\,]\!] : T \to \mathscr{P}(\mathbb{B} + \mathscr{D} \times \mathscr{D} + \mathscr{P}(\mathscr{D}))$, *defined as follows:*

- $\mathscr{E}[\![b]\!] = [\![b]\!]$;
- $\mathscr{E}[\![\mathbf{1}]\!] = \mathbb{B} + \mathscr{D} \times \mathscr{D} + \mathscr{P}(\mathscr{D})$, $\quad \mathscr{E}[\![\mathbf{0}]\!] = \varnothing$;
- $\mathscr{E}[\![\neg t]\!] = \mathscr{E}[\![\mathbf{1}]\!] \setminus \mathscr{E}[\![t]\!]$;
- $\mathscr{E}[\![t_1 \vee t_2]\!] = \mathscr{E}[\![t_1]\!] \cup \mathscr{E}[\![t_2]\!]$, $\mathscr{E}[\![t_1 \wedge t_2]\!] = \mathscr{E}[\![t_1]\!] \cap \mathscr{E}[\![t_2]\!]$;
- $\mathscr{E}[\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!]$;
- $\mathscr{E}[\![ch^{+}(t)]\!] = \{[\![t']\!] \mid [\![t]\!]' \subseteq [\![t]\!]\}$;
- $\mathscr{E}[\![ch^{-}(t)]\!] = \{[\![t']\!] \mid [\![t']\!] \supseteq [\![t]\!]\}$.

The definition of model is quite modular since it is invariant to the definition of the extensional interpretation. Thus we have the same definition as the one in Section 2.2:

Definition 4.2 *A pre-model* $(\mathscr{D}, [\![\,]\!])$ *is a* model *if for every type* t, $[\![t]\!] = \varnothing \iff \mathscr{E}[\![t]\!] = \varnothing$.

Having a definition of model allows us to define the pattern matching, which is the same as in Section 2.6 with the extra clause

$$\begin{array}{lcll} (d_1,d_2)/(p_1,p_2) & = & d_1/p_1 \otimes d_2/p_2 & \\ d/(p_1,p_2) & = & \Omega & \text{if } d \notin \mathscr{D}_p \end{array}$$

where $d_1/p_1 \otimes d_2/p_2$ is as defined in Section 2.6 with the extra clause

$\gamma(x) \;=\; (\gamma_1(x), \gamma_2(x)) \quad \text{if } x \in Dom(\gamma_1) \cap Dom(\gamma_2)$.

Theorems 2.16 and 2.17 still hold in the new framework.

Using the same techniques employed in Section 2.3, with some more complication to take care of product and recursive types, we obtain.

Theorem 4.3 *There exists a model for the type algebra with products.*

The decision algorithm of Section 2.5 can be extended to product types. We have to decide when a type of the form

$$(t \times s) \wedge \bigwedge_{i \in I} \neg(t_i \times s_i)$$

is empty. The algorithm works by recurring on the lexicographic order whose first component is the height (as before) and whose second component is the structure of product types. It is enough to be able to solve the problem for simpler types. This can be done by observing (see [6]) that the above type is equivalent to

$$\bigvee_{J \subseteq I} [(t \setminus \bigvee_{i \in J} t_i) \times (s \setminus \bigvee_{j \in I \setminus J} s_i)]$$

and that a union is empty if and only if all its addenda are empty.

That is not enough: we must also be able to decide whether a type is finite and, if it is the case, list all its atoms. That can be done again by observing that a union is finite if and only if all its addenda are, and that a product is finite if and only if both component are, or if one is empty. Moreover the atoms of a product are exactly the product of the atoms of the components.

The decidability of the subtyping for recursive types cannot work by induction in the same way. If an algorithm exists, it will have to be some coinductive procedure on the line of the one in [6]. We leave this investigation to future work.

4.4 Function types

In this section we want to briefly discuss the integration of semantic-π with the functional programming language $\mathbb{C}$Duce. A presentation of $\mathbb{C}$Duce is out of the scope of this work, and to understand the details of this section we invite to refer to [1, 6]. But the reader that managed to arrive so far will not have much difficulty to understand it, since $\mathbb{C}$Duce can be thought, quite roughly, as a sort of semantic-π where channel types and read and write actions are respectively replaced by arrows, functions and applications, and where pattern matching is explicitly done by a matching expression which allows the programmer to define overloaded functions with late binding.

Weak extension

A first a naive way to perform this integration is to add to semantic-π all $\mathbb{C}$Duce types as base types, and all $\mathbb{C}$Duce expressions to messages. It just suffices to add to the reduction rules the following rule

$$\frac{e \to e'}{\overline{c}^t e \longrightarrow \overline{c}^t e'} \tag{14}$$

and use $\mathbb{C}$Duce typing, algorithms and semantics whenever the semantic-π system needs them, and that's all. No other modification is necessary since the two systems are stratified, thus the definitions of model, subtyping etc., do not need to be changed.

As naive as it is this extension already allows us to define a "$\mathbb{C}$Duce server":

$$fun^{s \to t}(x).arg^{s}(y).\overline{result}^{t}(x(y))$$

which waits on *fun* and *arg* respectively for a function and its argument and returns the value of the application on *result*. A more liberal server that accepts the function and its argument on a channel *compute* in whatever order they are given is:

$$\begin{array}{rl} & compute^{((s\to t)\times s)\vee(s\times(s\to t))}(x,y\wedge s).\overline{result}^{t}(x(y)) \\ + & compute^{((s\to t)\times s)\vee(s\times(s\to t))}(x\wedge s,y).\overline{result}^{t}(y(x)) \end{array}$$

However the resulting system is mildly interesting since the two systems are weakly interacting: they do not share the same type constructors, so that π and $\mathbb{C}$Duce products are incompatible (for instance, the products in the type of *compute* channel must be π products, since otherwise we could not have deconstructed them in the input actions[8]), and they are stratified, so while channels can pass around $\mathbb{C}$Duce functions, these cannot work on channels.

Strong extension: $\mathbb{C}\pi$-calculus

Much a stronger interaction can be obtained by adding the arrow type constructor:

$$\textit{Types} \quad t \quad ::= \quad \dots \quad | \quad t\to t$$

This addition is more fruitful when it is done to the polyadic semantic-π, since then it unifies $\mathbb{C}$Duce and π product types.

Furthermore, not only $\mathbb{C}$Duce expressions are to be added to messages (as before) but also to channels, since now $\mathbb{C}$Duce expressions can calculate channels. This requires that besides (14) we must also add the following reduction rules:

$$\frac{e\longrightarrow e'}{\overline{e}M\longrightarrow \overline{e'}M} \qquad \frac{e\longrightarrow e'}{e(p).P\longrightarrow e'(p).P} \tag{15}$$

Contrary to the weak extension, here the definition of model and the model itself must be modified. Since we took a lot of care in giving the definitions of Section 2 so that they could be smoothly extended with arrow types this is not so difficult, the definition of model requiring that the extensional interpretation of $s\to t$ is the set of all sets of pairs such that if the first component is in the interpretation of s then the second is in the interpretation of t, namely

$$\mathscr{E}[\![s\to t]\!] = \mathscr{P}(\,([\![s]\!]\times[\![t]\!]^{\mathsf{c}})^{\mathsf{c}})$$

By merging the technique of [6] with the one we developed in Section 2.3 it is then possible to exhibit a model, essentially of the form

$$\mathscr{D} = \mathbb{B}+\mathscr{D}\times\mathscr{D}+\mathscr{P}_f(\mathscr{D}\times\mathscr{D})+\widetilde{\mathscr{T}}$$

where the reader can easily recognise which component comes from where. Although the way to proceed is quite smooth, the details are quite involved (essentially we have to redo the technical machinery of [6] and enrich it with the work done here) and here we prefer to omit them. However the complexity of the details is not the only reason that make us refrain to pushing our presentation further. Several other reasons advise us to do so:

- First and foremost, while we conjecture that the subtyping relation is decidable also for this extension, we were not able to prove it. This is a very important drawback since computations perform dynamic type-cases undecidability would mean that the operational semantics cannot be implemented.
- We would like to be able to give a type respecting encoding of $\mathbb{C}$Duce in process calculus part, similar to the Milner-Turner encoding of the simply typed λ-calculus in π [8, 10]. However all our tries so far have failed: it seems that the expressive power of typing sums is not enough to mimic that of $\mathbb{C}$Duce overloaded functions.
- In order to fully exploit the intertwining of the two type systems we should provide the languages with constructions to interact. For instance we could add to the $\mathbb{C}$Duce base types the type `thread` and modify the type-system of Figure 1 by replacing $\Gamma\vdash P:\texttt{thread}$ for every judgment $\Gamma\vdash P$. Then we should probably add to $\mathbb{C}$Duce a `spawn` function, and the possibility to communicate on channels, also we should enrich with synchronisation events and the primitives to handle that. All of this would lead us to deal with the design of concurrent functional languages, which is not the purpose of the work.

The whole point of this section was to show that the semantic subtyping technique constitutes a common structure on which it is possible to build and integrate functional and concurrent type systems. This can constitute the very starting point of a new promising research on functional concurrent and distributed languages, surely not its final point. To support our claim we want to show that even with the bare extension we described here—without any linguistic addition—it is possible to achieve a good degree of interaction between functional and concurrent structures, by describing a very naive example, which summarises the extensions that, at a different degree of detail, we presented in this section.

First we can use recursive and product types to define the type of associative lists, which associate a string key with a channel and where we use `nil` to denote both the empty list and its singleton type (we use sans_serif for recursion type variables):

$$\mathsf{a\_list} = ((\texttt{string}\times ch(\texttt{int}))\times \mathsf{a\_list})\vee\texttt{nil}$$

Associative lists can be searched with recursive patterns. For instance if we match an associative list with the following recursive pattern p:

$$p = (("\texttt{key1}",x),p)|(\mathbf{1},p)|(x:=\texttt{nil})$$

then x is bound to the list of all the channels that are associated with the key "`key1`" (strictly speaking, that have the

[8]If they were $\mathbb{C}$Duce products then the server would have been programmed as
]indent$compute^{((s\to t),s)\vee(s,(s\to t))}(x).$
$\quad\overline{result}^{t}(\texttt{match}\;x\;\texttt{with}\;(\mathbf{1},s)\to(\pi_1(x))\pi_2(y)$
$\qquad\qquad\qquad\qquad\quad|\;\mathbf{1}\quad\to(\pi_2(x))\pi_1(y))$

singleton type `key1`), while the following one

$$p = ((\texttt{"key1"}, x), \mathbf{1}) | (\mathbf{1}, p) | (x := \texttt{nil})$$

captures just the first channel associated with and then stops.

So we can use patterns to "calculate" channels. But when such a calculation is more complex (e.g. parametric in the key string), then it is better to delegate such a calculation to a function such as:

$$\begin{array}{l}
\texttt{fun}\ assoc(s : \texttt{string}\ , l : \mathsf{a\_list}) : ch(\texttt{int}) = \\
\quad \texttt{match}\ l\ \texttt{with}\ \texttt{nil}\ \rightarrow \texttt{fail} \\
\qquad\qquad | \ ((k, c), t)\ \rightarrow \texttt{if}\ k = s\ \texttt{then}\ c\ \texttt{else}\ assoc(s, t)
\end{array}$$

which can then be communicated by a process as a message on the channel *announce* below to dispatch all the notes of an examination:

$$\begin{array}{l}
announce^{\mathsf{m\_list} \times \mathsf{a\_list} \times (\texttt{string} \times \texttt{a\_list} \rightarrow ch(\texttt{int}))}(marks, mails, getch). \\
\quad (\nu c^{\mathsf{m\_list}})\ \overline{c}(marks)\ \ | \\
\qquad !(\quad c(\ ((n,m), rest)\ ).\ (\ \overline{getch(n, mails)}(m)\ \ |\ \ \overline{c}(rest)\ ) \\
\qquad\quad +\ c(\ \texttt{nil}).0\ )
\end{array}$$

where $\mathsf{m\_list} = ((\texttt{string} \times \texttt{int}) \times \mathsf{m\_list}) \vee \texttt{nil}$. The channel *announce* waits for an associative list of marks, an associative list of channels, and a dispatch function that calculates a channel. The process creates a private channel c to iterate on the list of marks (since it must communicate on channels, then in the absence of a `spawn` we cannot use a function to perform such an iteration) and use the function received on *announce* (bound to *getch*) to calculate the channel $getch(n, mails)$ on which to write the mark, and iterating the process with the rest of the list.

When the list of marks is empty (the pattern `nil` matches it), the process becomes inert.

So for instance if the following process synchronises

$$\begin{array}{l}
\overline{announce}(\ (\ (\texttt{"Alice"}, 6), ((\texttt{"Bob"}, 8), \texttt{nil}))\ , \\
\qquad\qquad (\ (\texttt{"Bob"}, c_B), ((\texttt{"Alice"}, c_A), \texttt{nil}))\ , \\
\qquad\qquad assoc\ \ )
\end{array}$$

it will produce the process: $\overline{c_A}(6)\ \ |\ \ \overline{c_B}(8)$.

Acknowledgements

We are very grateful to Alain Frisch for interesting discussions, and Daniele wants to thank him for making available parts of his forthcoming PhD thesis. Giuseppe and Daniele acknowledge the support of the European FET contract *MyThS*, IST-2001-32617. Rocco acknowledges the support of FET contract *Mikado*, IST-2001-32222, and would like to thank ENS for a visiting professorship grant.

References

[1] 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.

[2] 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.

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

[4] G. Castagna and A. Frisch. A gentle introduction to semantic subtyping. In *Second workshop on Programmable Structured Documents*, Hakone, Japan, 2004. Invited paper. Available at `www.cduce.org/papers`.

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

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

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

[8] R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

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

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

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

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

For referees' convenience only. To be omitted in final version.

A Type algorithm

The type algorithm is obtained from the typing rules in a standard way, namely by deleting the subsumption rule and embedding the checking of the subtyping relation in the elimination rules. The only difficult point is the definition of $\mathscr{C}()$, that is the least upper bound of s which is of the form $ch^+(t)$. The decidability of $\mathscr{C}()$ is given by Theorem 3.5. The algorithmic rules are summarised in Figure 3.

B Proofs

B.1 Characterising inclusion (Theorem 2.9 and Proposition 2.10)

In this section we first prove Theorem 2.9 and then strengthen the result as in Proposition 2.10.

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 of equal than an atom. It is easy to prove that an atomic boolean algebra is equivalent to a subset of the powerset of its atoms.

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

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

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

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

and

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

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

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

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

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

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

The most difficult case occurs when

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

The way of thinking the inclusion is the following. (From now on it will be easier to think of D as a subset of the powerset of its atoms; therefore we will use $\subseteq$ rather than $\leq$ and "contained" rather than "smaller".) Consider an element in $ch^+(d_1) \cap ch^-(d_2)$. If it 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, $d_4^k \setminus d_2 \subseteq \{x\}$[9]. 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, $d_4^k \setminus d_2 \subseteq \{x_1, x_2\}$.

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

Therefore the condition we look for is: for every $\mathscr{X}$ such that $\bigcap \mathscr{X} = \varnothing$, for every choice of $x_I \in e_I, I \in \mathscr{X}$ there must be a d_4^k such that $d_4^k \setminus d_2 \subseteq \{x_I \mid I \in \mathscr{X}\}$.

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

We can strengthen the result by as stated in Proposition 2.10. Consider the case where some of the e_I are infinite. Since there are only finitely many d_4^k, the condition is

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

Messages

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

Processes

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

$$\begin{array}{c} t \le \bigvee_{i\in I} \lbag p_i \rbag \\ \lbag p_i \rbag \wedge t \neq \mathbf{0} \end{array}\ \frac{\Gamma \vdash \alpha : s \quad \mathscr{C}(s) = ch^+(t) \quad \Gamma, t/p_i \vdash P_i}{\Gamma \vdash \sum_{i\in I} \alpha(p_i).P_i}\ \text{(input)} \qquad \frac{\Gamma \vdash M : t \quad \Gamma \vdash \alpha : s \quad s \le ch^-(t)}{\Gamma \vdash \overline{\alpha} M}\ \text{(output)}$$

Figure 3: Algorithmic rules

satisfied if and only if for at least two (in fact infinitely many) different choices x_I and x'_I we have that the same d_4^k satisfies $d_4^k \setminus d_2 \subseteq \{x_I \mid I \in \mathscr{X}\}$, and $d_4^k \setminus d_2 \subseteq \{x'_I \mid I \in \mathscr{X}\}$. Therefore we must have $d_4^k \setminus d_2 \subseteq \{x_I \mid I \in \mathscr{X}\ \&\ e_I \text{ finite }\}$. (We could improve further by considering only those e_I whose cardinality is not greater than the number of d_4^k - we don't need this for our purposes).

The condition we will use in our proof is: for every $\mathscr{X}$ such that $\bigcap \mathscr{X} = \varnothing$, for every choice of $x_I \in e_I$, $I \in \mathscr{X}$, e_I finite, there must be a d_4^k such that $d_4^k \setminus d_2 \subseteq \{x_I \mid I \in \mathscr{X}\}$.

B.2 Proof of Proposition 2.4

We prove the theorem for the simplest type system, with only basic types. The case with product is very similar; while recursive types (and possibly arrow types) require some refined techniques.

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

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

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

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

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

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

with the semantics of the channel types being

$$\begin{array}{rcl} [\![ch^+(t)]\!]_{n+1} & = & \{[t']_{=_n} \mid t' \le_n t\} \\ [\![ch^-(t)]\!]_{n+1} & = & \{[t']_{=_n} \mid t \le_n t'\} \\ [\![\mathbf{k+1}]\!]_{n+1} & = & \{[\mathbf{k}]_{=_n}\} \end{array}$$

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

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

Therefore it is enough to check emptiness for intersections of literals of on 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 (6) and (7) of Section 2 we can simplify the above expression to

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

To prove Proposition 2.4, we now prove by induction the following statement: let $t \in \mathscr{T}_n$, then

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

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

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

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

Which can be true only if there are two different $\mathbf{k} \in P$ or if the only $\mathbf{k}$ in P is also in N. It is important here that

$\mathbb{N}$ is infinite, so no finite union of singleton can cover it. Therefore the condition above is equivalent to

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

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

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

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

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

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

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

$$[\![ch^+(t_1)]\!]_{n+1} \cap [\![ch^-(t_2)]\!]_{n+1}$$
$$\subseteq \bigcup_{h\in H} [\![ch^+(t_3^h)]\!]_{n+1} \cup \bigcup_{k\in K} [\![ch^-(t_4^k)]\!]_{n+1}$$

if and only if

$$[\![ch^+(t_1)]\!]_{n+2} \cap [\![ch^-(t_2)]\!]_{n+2}$$
$$\subseteq \bigcup_{h\in H} [\![ch^+(t_3^h)]\!]_{n+2} \cup \bigcup_{k\in K} [\![ch^-(t_4^k)]\!]_{n+2}$$

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

- $t_2 \not\leq_n t_1$ or
- $\exists h \in H$ such that $t_1 \leq_n t_3^h$ or
- $\exists k \in K$ such that $t_4^k \leq_n t_2$ or
- the complicated condition involving $\leq_n$ and atoms

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

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

are equivalent to

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

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

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

we have to check that the condition

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

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

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

Lemma B.1 *Suppose that for every $t \in \mathscr{T}_n$*

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

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

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

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

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

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

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

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

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

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

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

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

We finally observe that adding the $\mathbf{k}$ to our types is not restrictive, as $\mathbf{k} =_{k+1} ch(\mathbf{0})^{k+1}$

B.3 Proof of Proposition 2.6

Remember we defined

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

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

Moreover we put $[\![\mathbf{k}+\mathbf{1}]\!] = \{[\mathbf{k}]_{=_\infty}\}$.

This pre-model defines a new preorder between types that we denote by $\leq$. We then have (Proposition 2.6):

- Let $t, t' \in T$ then $t \leq t'$ if and only if $t \leq_\infty t'$.

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

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

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

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

The second statement, and the all inductive step are proven as in the proof of Proposition 2.4.

B.4 Proof of Theorem 2.7

Theorem 2.7 states that the pre-model $(\mathscr{D}, [\![\,]\!])$ is a model.

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 2.6, we have that $\langle [\![\mathscr{T}]\!], \subseteq\rangle$ is isomorphic to $\langle \widetilde{\mathscr{T}}, \leq\rangle$. Up to this isomorphism, $\mathscr{E}[\![\,]\!]$ coincides with $[\![\,]\!]$. □

B.5 Proof of Theorem 3.1

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

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

Point 1 is a simple application of the subsumption rule. For 2 suppose that exist t such that $v : t$ and $v \neg t$. The only rule to deduce a negative type for a value is the subsumption rule. Therefore it must be the case that $t \leq \neg t$. But then $t = \mathbf{0}$ impossible since the empty type is not inhabited. If instead there exists t such that $\not\vdash v : t$ and $\not\vdash v : \neg t$; if $v = c^s$ then $ch(s)$ is not smaller than t nor than $\neg t$, impossible since $ch(s)$ is atomic. The same can be deduced from the atomicity of b_n for $v = n$ ($[\![b_n]\!] = \{n\}$ see Definition 2.13). Therefore $(\mathscr{V}, [\![\,]\!]_{\mathscr{V}})$ is a premodel.

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

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