In ICTCS '05, Lecture Notes in Computer Science, ????:??-??, © Springer, 2005.

Semantic subtyping: challenges, perspectives, and open problems

Giuseppe Castagna

CNRS, École Normale Supérieure de Paris, France

Based on joint work with: Véronique Benzaken, Rocco De Nicola, Mariangiola Dezani, Alain Frisch, Haruo Hosoya, Daniele Varacca

Abstract. Semantic subtyping is a relatively new approach to define subtyping relations where types are interpreted as sets and union, intersection and negation types have the corresponding set-theoretic interpretation. In this lecture we outline the approach, give an aperçu of its expressiveness and generality by applying it to the λ-calculus with recursive and product types and to the π-calculus. We then discuss in detail the new challenges and research perspectives that the approach brings forth.

1 Introduction to the semantic subtyping

Many recent type systems rely on a subtyping relation. Its definition generally depends on the type algebra, and on its intended use. We can distinguish two main approaches for defining subtyping: the *syntactic* approach and the *semantic* one. The syntactic approach—by far the more used—consists in defining the subtyping relation by axiomatising it in a formal system (a set of inductive or coinductive rules); in the semantic approach (for instance, [AW93,Dam94]), instead, one starts with a model of the language and an interpretation of types as subsets of the model, then defines the subtyping relation as the inclusion of denoted sets, and, finally, when the relation is decidable, derives a subtyping algorithm from the semantic definition.

The semantic approach has several advantages (see [CF05] for an overview) but it is also more constraining. Finding an interpretation in which types can be interpreted as subsets of a model may be a hard task. A solution to this problem was given by Haruo Hosoya and Benjamin Pierce [HP01,Hos01,HP03], who noticed that in order to define subtyping all is needed is a set theoretic interpretation of types, not a model of the terms. In particular, they propose to interpret a type as the set of all values that have that type. So if we use $\mathscr{V}$ to denote the set of all values, then we can define the following set-theoretic interpretation for types $[\![t]\!]_{\mathscr{V}} = \{v \in \mathscr{V} \mid \vdash v : t\}$ which induces the following subtyping relation:

$$s \leq_{\mathscr{V}} t \quad \overset{def}{\iff} \quad [\![s]\!]_{\mathscr{V}} \subseteq [\![t]\!]_{\mathscr{V}} \tag{1}$$

This works for Hosoya and Pierce because the set of values they consider can be defined independently from the typing relation.[1] But in general in order to state when a value has a given type (the "$\vdash v : t$" in the previous definition) one needs the subtyping relation. This yields a circularity: we are building a model to define the subtyping relation, and the definition of this model needs the subtyping relation. This circularity is patent in both the examples we discuss below: in λ-calculus (Section 2) values are λ-abstractions and to type them (in particular, to type applications that may occur in their body) subtyping is needed; in π-calculus (Section 3) the covariance and contravariance of read-only and write-only channel types make the subtyping relation necessary to type channels.

In order to avoid this circularity and still interpret types as set of values, we resort to a bootstrapping technique. The general ideas of this technique are informally exposed in [CF05], while the technical development can be found in [FCB02,Fri04]. For the aims of this article, the process of defining semantic subtyping can be roughly summarised in the following steps:

1. Take a bunch of type *constructors* (e.g., $\rightarrow$, $\times$, $ch()$, $\ldots$) and extend the type algebra with the following *boolean combinators*: union $\vee$, intersection $\wedge$, and negation $\neg$.
2. Give a *set-theoretic model* of the type algebra, namely define a function $[\![\,]\!]_{\mathscr{D}} : \mathbf{Types} \rightarrow \mathscr{P}(\mathscr{D})$, for some domain $\mathscr{D}$ (where $\mathscr{P}(\mathscr{D})$ denotes the powerset of $\mathscr{D}$). In such a model, the combinators must be interpreted in a set-theoretic way (that is, $[\![s \wedge t]\!]_{\mathscr{D}} = [\![s]\!]_{\mathscr{D}} \cap [\![t]\!]_{\mathscr{D}}$, $[\![s \vee t]\!]_{\mathscr{D}} = [\![s]\!]_{\mathscr{D}} \cup [\![t]\!]_{\mathscr{D}}$, and $[\![\neg t]\!]_{\mathscr{D}} = \mathscr{D} \setminus [\![t]\!]_{\mathscr{D}}$), and the definition of the model must capture the essence of the type constructors.
   There might be several models, and each of them induces a specific subtyping relation on the type algebra. We only need to prove that there exists at least one model and then pick one that we call the *bootstrap model*. If its associated interpretation function is $[\![\,]\!]_{\mathscr{B}}$, then it induces the following subtyping relation:
   $$s \leq_{\mathscr{B}} t \quad \overset{def}{\Longleftrightarrow} \quad [\![s]\!]_{\mathscr{B}} \subseteq [\![t]\!]_{\mathscr{B}} \tag{2}$$
3. Now that we defined a subtyping relation for our types, find a subtyping algorithm that decides (or semi-decides) the relation. This step is not mandatory but highly advisable if we want to use our types in practise.
4. Now that we have a (hopefully) suitable subtyping relation available, we can focus on the language itself, consider its typing rules, use the new subtyping relation to type the terms of the language, and deduce $\Gamma \vdash_{\mathscr{B}} e : t$. In particular this means to use in the subsumption rule the bootstrap subtyping relation $\leq_{\mathscr{B}}$ we defined in step 2.

[1] Their values are XML documents, and they can be defined as regular trees. The typing relation, then, becomes recognition of a regular tree language.

5. The typing judgement for the language now allows us to define a *new* natural set-theoretic interpretation of types, the one based on values $[\![t]\!]_{\mathscr{V}} = \{v \in \mathscr{V} \mid \vdash_{\mathscr{B}} v : t\}$, and then define a "new" subtyping relation as in equation (1). This relation might be different from $\leq_{\mathscr{B}}$ we started from. However, if the definitions of the model, of the language, and of the typing rules have been carefully chosen, then the two subtyping relations coincide

$$s \leq_{\mathscr{B}} t \iff s \leq_{\mathscr{V}} t$$

and this closes the circularity. Then, the rest of the story is standard (reduction relation, subject reduction, type-checking algorithm, etc ...).

The accomplishment of this is process is far from being straightforward. In point 2 it may be quite difficult to capture the semantics of the type constructors (e.g., it is quite hard to define a set-theoretic semantics for arrow types); in point 3 defining a model may go from tricky to impossible (e.g., because of bizarre interactions with recursive types); point 4 may fail for the inability of devising a subtyping algorithm (cf. the subtyping algorithm for $\mathbb{C}\pi$ in [CNV05]); finally the last step is the most critical one since it may require a consistent rewriting of the language and/or of the typing rules to "close the circle" ... if possible at all. We will give examples of all these problems in the rest of this document.

In the next two sections we are going to show how to apply this process to λ-like and π-like calculi. The presentation will be sketchy and presuppose from the reader some knowledge of the λ-calculus, of the π-calculus, and of their type systems. Also, the calculi we are going to present are very simplified versions of the actual ones whose detailed descriptions can be found in [FCB02] and [CNV05], respectively.

2 Semantic λ-calculus: $\mathbb{C}$Duce

As a first example of application of the semantic subtyping 5-steps technique, let us take the λ-calculus with products.

Step 1. The first step consists in taking some type constructors, in this case products and arrows, and adding boolean combinators to them:

$$t ::= \mathbb{0} \mid \mathbb{1} \mid t \rightarrow t \mid t \times t \mid \neg t \mid t \vee t \mid t \wedge t$$

where $\mathbb{0}$ and $\mathbb{1}$ correspond, respectively, to the empty and the universal types. For more generality we consider also recursive types. Thus, our types are the regular trees generated by the grammar above and satisfying the standard contractivity condition that every infinite branch has infinitely many occurrences of the $\times$ or of the $\rightarrow$ constructors (this rules out meaningless expressions such as $t \wedge (t \wedge (t \wedge (\ldots))))$.

Step 2. The second step is, in this case, the hard one as it requires to define a set-theoretic interpretation $[\![\,]\!]_{\mathscr{D}} : \textbf{Types} \to \mathscr{P}(\mathscr{D})$. But, how can we give a set theoretic interpretation to the arrow type? The set theoretic intuition we have of $t \to s$ is that it is the set of all functions (of our language) that when applied to a value of type t either diverge or return a result of type s. If we interpret functions as binary relations on $\mathscr{D}$, then $[\![t \to s]\!]$ is the set of binary relations in which if the first projection is in (the interpretation of) t then the second projection is in (the interpretation of) s, namely $\mathscr{P}(\overline{[\![t]\!] \times \overline{[\![s]\!]}})$, where the overline denotes set complement.[2] However, setting $[\![t \to s]\!] = \mathscr{P}(\overline{[\![t]\!] \times \overline{[\![s]\!]}})$ is impossible since, for cardinality reasons, we cannot have $\mathscr{P}(\mathscr{D}^2) \subseteq \mathscr{D}$. Note though, that we do not define the interpretation $[\![\,]\!]$ in order to formally state what the syntactic types *mean* but, more simply, we define it in order to state how they are *related*. Therefore, even if the interpretation does not capture the intended semantics of types, all we need is that it captures the containment relation induced by this semantics. That is, roughly, it suffices to our aims that the interpretation function satisfies

$$[\![t_1 \to s_1]\!] \subseteq [\![t_2 \to s_2]\!] \iff \mathscr{P}(\overline{[\![t_1]\!] \times \overline{[\![s_1]\!]}}) \subseteq \mathscr{P}(\overline{[\![t_2]\!] \times \overline{[\![s_2]\!]}}) \tag{3}$$

Note that $\mathscr{P}_f(X) \subseteq \mathscr{P}_f(Y)$ if and only if $\mathscr{P}(X) \subseteq \mathscr{P}(Y)$ (where $\mathscr{P}_f$ denotes the finite powerset). Therefore, if we set $[\![t \to s]\!] = \mathscr{P}_f(\overline{[\![t]\!] \times \overline{[\![s]\!]}})$, this interpretation satisfies (3). In other words, we can use as bootstrap model $\mathscr{B}$ the least solution of the equation $X = X^2 + \mathscr{P}_f(X^2)$ and the following interpretation function[3] $[\![\,]\!]_{\mathscr{B}} : \textbf{Types} \to \mathscr{P}(\mathscr{B})$:

$$[\![\mathbb{0}]\!]_{\mathscr{B}} = \varnothing \quad [\![\mathbb{1}]\!]_{\mathscr{B}} = \mathscr{B} \qquad [\![s \vee t]\!]_{\mathscr{B}} = [\![s]\!]_{\mathscr{B}} \cup [\![t]\!]_{\mathscr{B}} \qquad [\![s \wedge t]\!]_{\mathscr{B}} = [\![s]\!]_{\mathscr{B}} \cap [\![t]\!]_{\mathscr{B}}$$
$$[\![\neg t]\!]_{\mathscr{B}} = \mathscr{B} \setminus [\![t]\!]_{\mathscr{B}} \qquad [\![s \times t]\!]_{\mathscr{B}} = [\![s]\!] \times [\![t]\!] \qquad [\![t \to s]\!]_{\mathscr{B}} = \mathscr{P}_f(\overline{[\![t]\!]_{\mathscr{B}} \times \overline{[\![s]\!]_{\mathscr{B}}}})$$

The model we have chosen can represent only finite graph functions, therefore it is not rich enough to give semantics to a λ-calculus (even the simply typed one). However since this model satisfies equation (3), it is able to express the containment relation induced by the semantic intuition we have of the type $t \to s$ (namely that it represents $\mathscr{P}(\overline{[\![t]\!] \times \overline{[\![s]\!]}})$, which is all we need.

Step 3. We can use the definition of subtyping as given by equation (2) to deduce some interesting relations: for instance, according to (2) the type $(t_1 \to$

[2] This is just one of the possible interpretations. See [CF05] for a discussion about the implications of such a choice and [Fri04] for examples of different interpretations.

[3] For the details of the definition of the interpretation in the presence of recursive types, the reader is invited to consult [Fri04] and [FCB02]. The construction is also outlined in [CF05].

$s_1) \wedge (t_2 \to s_2)$ is a subtype of $(t_1 \wedge t_2) \to (s_1 \wedge s_2)$, of $(t_1 \vee t_2) \to (s_1 \vee s_2)$, of their intersection and, in general, all these inclusions are strict.

Apart from these examples, the point of course is to devise an algorithm to decide inclusion between any pair of types. Deciding subtyping for arbitrary types is equivalent to decide whether a type is equivalent to (that is, it has the same interpretation as) $\mathbb{0}$:

$$s \leq_{\mathscr{B}} t \Leftrightarrow [\![s]\!]_{\mathscr{B}} \subseteq [\![t]\!]_{\mathscr{B}} \Leftrightarrow [\![s]\!]_{\mathscr{B}} \cap \overline{[\![t]\!]_{\mathscr{B}}} = \varnothing \Leftrightarrow [\![s \wedge \neg t]\!]_{\mathscr{B}} = \varnothing \Leftrightarrow s \wedge \neg t = \mathbb{0}.$$

By using the definition of $\mathscr{B}[\![\,]\!]$, we can show that every type is equivalent to a finite union where each summand is either of the form:

$$(\bigwedge_{s \times t \in P} s \times t) \wedge (\bigwedge_{s \times t \in N} \neg(s \times t)) \tag{4}$$

or of the form

$$(\bigwedge_{s \to t \in P} s \to t) \wedge (\bigwedge_{s \to t \in N} \neg(s \to t)) \tag{5}$$

Put $s \wedge \neg t$ in this form. Since it is a finite union, then it is equivalent to $\mathbb{0}$ if and only if each summand is so. So the decision of $s \leq_{\mathscr{B}} t$ is reduced to the problem of deciding whether the types in (4) and (5) are empty. The subtyping algorithm, then, has to work coinductively, decomposing these problems into simpler subproblems where the topmost type constructors have disappeared. In particular, in [Fri04] it is proved that the type in (4) is equivalent to $\mathbb{0}$ if and only if for every $N' \subseteq N$:

$$\left(\bigwedge_{(t \times s) \in P} t \;\wedge \bigwedge_{(t' \times s') \in N'} \neg t'\right) \simeq \mathbb{0} \;\text{ or }\; \left(\bigwedge_{(t \times s) \in P} s \;\wedge \bigwedge_{(t' \times s') \in N \setminus N'} \neg s'\right) \simeq \mathbb{0}; \tag{6}$$

while the type in (5) is equal to zero if and only if there exists some $(t' \to s') \in N$ such that for every $P' \subseteq P$:

$$\left(t' \wedge \bigwedge_{(t \to s) \in P'} \neg t\right) \simeq \mathbb{0} \;\text{ or }\; \left(\bigwedge_{(t \to s) \in P \setminus P'} s \wedge \neg s'\right) \simeq \mathbb{0}. \tag{7}$$

By applying these decompositions the algorithm can decide the subtyping relation. Its termination is ensured by the regularity and contractivity of the types.

Step 4 We have just defined a decidable subtyping relation for our types. We now want to apply it to type the terms of a language. We do not present here a complete language: the reader can find plenty of details in [CF05,FCB02]. Instead, we concentrate on the definition and the typing of the terms that are the most interesting for the development of this article, namely λ-abstractions. These in the rest of this paper will have the form $\lambda^{\wedge_{i \in I} s_i \to t_i} x.e$, that is we index them by an intersection type. This index instructs the type checker to verify that the abstraction is in the given intersection, namely, that it has all the types composing it, as implemented by the following rule:

$$\frac{\begin{array}{l} t{\equiv}(\bigwedge_{i=1..n} s_i{\to}t_i)\wedge(\bigwedge_{j=1..m}\neg(s'_j{\to}t'_j)){\neq}\mathbb{0} \\ (\forall i)\ \ \Gamma, x : s_i \vdash_{\mathscr{B}} e : t_i \end{array}}{\Gamma \vdash_{\mathscr{B}} \lambda^{\wedge_{i\in I} s_i \to t_i} x.e : t}\ \text{(abstr)}$$

To understand this rule consider, as a first approximation, the case for $m = 0$, that is, when the type t assigned to the function is exactly the index. The rule verifies that the function has indeed all the $s_i \to t_i$ types: for every type $s_i \to t_i$ of the intersection it checks that the body e has type t_i under the assumption that the parameter x has type s_i. The rule actually is (and must be) more general since it allows the type checker to infer for the function a type t strictly smaller than the one at the index, since the rule states that it is possible to subtract from the index any finite number of arrow types, provided that t remains non-empty.[4] This is necessary to step 5 of our process. But before moving to the next step, note that the intersection of arrows can be used to type overloaded functions. Indeed, our framework is compatible with overloading, since the following containment

$$[\![(t_1 \vee t_2) \to (s_1 \wedge s_2)]\!] \subsetneq [\![(t_1 \to s_1) \wedge (t_2 \to s_2)]\!] \qquad (8)$$

is strict. So the semantic model authorises the language to define functions that return different results (e.g. one in $s_1 \backslash s_2$ and the other in $s_2 \backslash s_1$) according to whether their argument is of type t_1 or t_2. If the model had instead induced an equality between the two types above then the function could not have different behaviours for different types but should uniformly behave on them.[5]

Step 5. The last step consists in verifying whether the model of values induces that same subtyping relation as the bootstrap one. This holds only if

$$\vdash_{\mathscr{B}} v : t \quad \iff \quad \not\vdash_{\mathscr{B}} v : \neg t \qquad (9)$$

which holds true (and, together with the fact that no value has the empty type, makes the two subtyping relations coincide) thanks to the fact that the (abstr) rule deduces negated arrow types for lambda abstractions. Without it the difference of two arrow types (which in general is non-empty) might be not inhabited by a value, since the only way to deduce for an abstraction a negated arrow would be the subsumption rule. To put it otherwise, without the negated arrows in (abstr) property (9) would fail since, for instance, both $\not\vdash \lambda^{Int \to Int} x.(x+3) : \neg(Bool \to Bool)$, and $\not\vdash \lambda^{Int \to Int} x.(x+3) : Bool \to Bool$ would hold.

[4] Equivalently, it states that we can deduce for a λ-abstraction every non-empty type t obtained by intersecting the type indexing the abstraction with a finite number of negated arrow types that do not already contain the index.

[5] Overloading requires the addition of a type-case in the language. Without it intersection of arrows can just be used to give more specific behaviour, as for $\lambda^{Odd \to Odd \wedge Even \to Even} x.x$ which is more precise than $\lambda^{Int \to Int} x.x$.

3 Semantic π-calculus: $\mathbb{C}\pi$

In this section we repeat the 5 steps process for the π-calculus.

Step 1. The types we are going to consider are the following ones

$$t \quad ::= \quad \mathbb{0} \mid \mathbb{1} \mid ch^{+}(t) \mid ch^{-}(t) \mid ch(t) \mid \neg t \mid t \vee t \mid t \wedge t$$

without any recursion. As customary $ch^{+}(t)$ is the type of channels on which one can expect to receive values of type t, $ch^{-}(t)$ is the type of channels on which one is allowed to send values of type t, while $ch(t)$ is the type of channels on which one can send and expect to receive values of type t.

Step 2. The set-theoretic intuition of the above types is that they denote sets of channels. In turn a channel can be seen as a box that is tightly associated to the type of the objects it can transport. So $ch(t)$ will be the set of all boxes for objects of type t, $ch^{-}(t)$ the set of all boxes in which one can put something of type t while $ch^{+}(t)$ will be the set of boxes in which one expects to find something of type t. This yields the following interpretation

$$\begin{aligned}
[\![ch(t)]\!] &= \{c \mid c \text{ is a box for objects in } [\![t]\!]\}\\
[\![ch^{+}(t)]\!] &= \{c \mid c \text{ is a box for objects in } [\![s]\!] \text{ with } s \leq t\}\\
[\![ch^{-}(t)]\!] &= \{c \mid c \text{ is a box for objects in } [\![t]\!] \text{ with } s \geq t\}.
\end{aligned}$$

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

$$[\![ch^{+}(t)]\!] = \Big\{[\![s]\!] \mid s \leq t\Big\} \qquad\qquad [\![ch^{-}(t)]\!] = \Big\{[\![s]\!] \mid s \geq t\Big\} \tag{10}$$

while the invariant channel type $ch(t)$ will be interpreted as the singleton $\{[\![t]\!]\}$. Of course, there is a circularity in the definitions in (10), since the subtyping relation is not defined, yet. So we rather use the following interpretations $[\![ch^{+}(t)]\!] = \{[\![s]\!] \mid [\![s]\!] \subseteq [\![t]\!]\}$, $[\![ch^{-}(t)]\!] = \{[\![s]\!] \mid [\![s]\!] \supseteq [\![t]\!]\}$, which require to have a domain that satisfies $\{[\![t]\!] \mid t \in \mathbf{Types}\} \subseteq \mathscr{D}$. This is not straightforward but doable, as shown in [CNV05].

Step 3. As for the λ-calculus we can use the definition of the model given by (10) to deduce some interesting relations. First of all, the reader may have already noticed that

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

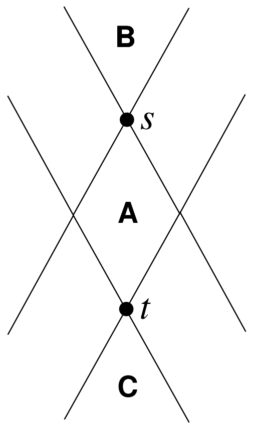

Fig. 1. Deciding atomicity

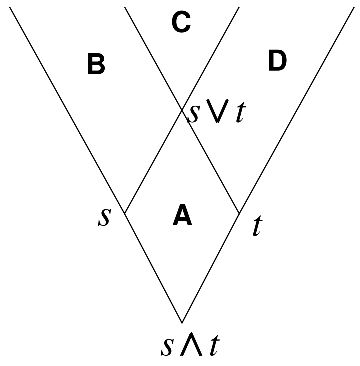

Fig. 2. Some equations

thus, strictly speaking, the *ch* constructor is nothing but syntactic sugar for the intersection above (henceforth, we will no longer consider this constructor and concentrate on the covariant and contravariant channel type constructors). Besides this relation, far more interesting relations can be deduced and, quite remarkably, in many case this can be done graphically. Consider the definitions in (10): they tell us that the interpretation of $ch^+(t)$ is the set of the interpretations of all the types smaller than or equal to t. As such, it can be represented by the downward cone starting from t. Similarly, the upward cone starting from t represents $ch^-(t)$. This illustrated in Figure 1 where the upward cone B represents $ch^-(s)$ and the downward cone C represents $ch^+(t)$. If we now pass on Figure 2 we see that $ch^-(s)$ is the upward cone B+C and $ch^-(t)$ is the upward cone C+D. Their intersection is the cone C, that is the upward cone starting from the least upper bound of s and t which yields the following equation

$$ch^-(s) \wedge ch^-(t) = ch^-(s \vee t)\ . \tag{11}$$

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

$$ch^-(s) \vee ch^-(t) \lneq ch^-(s \wedge t)\ . \tag{12}$$

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

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

We could continue to devise such equations, but the real challenge is to decide whether two generic types are one subtype of the other. As in the case for λ-calculus we can reduce the problem of subtyping two types to the decision of the emptiness of a type (the difference of the two types). If we put this type in disjunctive normal form, then it comes to decide whether $\bigwedge_{i\in P} t_i \wedge \bigwedge_{j\in N} \neg t'_j = \mathbb{0}$, that is whether $\bigwedge_{i\in P} t_i \leq \bigvee_{j\in N} t'_j$. With the type constructors specific to $\mathbb{C}\pi$ this expands to $\bigwedge_{i\in I} ch^+(t^i_1) \wedge \bigwedge_{j\in J} ch^-(t^j_2) \leq \bigvee_{h\in H} ch^+(t^h_3) \vee \bigvee_{k\in K} ch^-(t^k_4)$. Since intersections can always be pushed inside type constructors (we saw it in (11) for ch^- types, the reader can easily check it for ch^+), then we end up with checking the following inequality:

$$ch^+(t_1) \wedge ch^-(t_2) \quad \leq \quad \bigvee_{h\in H} ch^+(t^h_3) \vee \bigvee_{k\in K} ch^-(t^k_4)\ . \tag{13}$$

This is indeed the most difficult part of the job, since while in some cases it is easy to decide the inclusion above (for instance, when $t_2 \not\leq t_1$ since then the right-hand side is empty), in general, this requires checking whether a type is *atomic*, that is whether its only proper subtype is the empty type (for sake of simplicity the reader can think of the atomic types as the singletons of the type system[6]). The general case is treated in [CNV05], but to give an idea of the problem consider the equation above with only two types s and t with $t \lneq s$, and let us try to check if:

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

Once more, a graphic representation is quite useful. The situation is represented in Figure 1 where the region A represents the left-hand side of the inequality, while the region B+C is the right hand side. So to check the subtyping above we have to check whether A is contained in B+C. At first sight these two regions looks completely disjoint, but observe that they have at least two points in common, marked in bold in the figure (they are respectively the types $ch(s)$ and $ch(t)$). Now, the containment holds if the region A does not contain any other type besides these two. This holds true if and only if there is no other type between s and t, that is if and only if $s\backslash t$ (i.e. $s \wedge \neg t$) is an atomic type.

Step 4. The next step is to devise a π-calculus that fits the type system we have just defined. Consider the dual of equation (12):[7]

$$ch^+(s) \vee ch^+(t) \lneq ch^+(s \vee t) \tag{14}$$

and in particular the fact that the inclusion is strict. A suitable calculus must distinguish the two types above. The type on the left contains either channels on which we will always read s-objects or always read t-objects, while the type on

[6] Nevertheless, notice that according to their definition, atomic types may be neither singletons nor finite. For instance $ch(\mathbb{0})$ is atomic, but in the model of values it is the set of all the synchronisation channels; these are just token identifiers on a denumerable alphabet, thus the type is denumerable as well.

[7] To check this inequality turn Figure 2 upside down.

the right contains channels on which objects of type s or of type t may arrive interleaved. If we use a channel with the left type and we can test the type of the first message we receive on it, then we can safely assume that all the following messages will have the same type. Clearly using in such a context a channel with the type on the right would yield a run-time type error, so the two types are observationally different. This seems to suggest that a suitable calculus should be able to test the types of the messages received on a channel, which yields to the following definition of the calculus:

$$\begin{array}{lll} \textit{Channels} & \alpha & ::= x \mid c^t \\ \textit{Processes} & P & ::= \overline{\alpha}(\alpha) \mid \sum_{i\in I}\alpha(x:t_i)P_i \mid P_1 \| P_2 \mid (\nu c^t)P \mid\ !P \end{array}$$

The main difference with respect to the standard π-calculus is that we have introduced channel *values*, since a type-case must not be done on open terms. Thus, c^t is a physical box that can transport objects of type t: channels are tightly connected to the type of the objects they transport. Of course, restrictions are defined on channels, rather than channel variables (since the latter could be never substituted). The type-case is then performed by the following reduction rule

$$\overline{c}_1^s(c_2^t) \parallel \textstyle\sum_{i\in I} c_1^s(x:t_i)P_i \quad \rightarrow \quad P_j[c_2^t/x] \qquad\qquad \text{if } ch(t) \leq t_j$$

This is the usual π-calculus reduction with three further constraints: (i) synchronisation takes place on channels (rather than on channel variables),(ii) it takes place only if the message is a channel value (rather than a variable), and (iii) only if the type of the message (which is $ch(t)$) matches the type t_j of the formal parameter of the selected branch of the summation. The last point is the one that implements the type-case. It is quite easy to use intersection and negation types to force the various branches of the summation to be mutually exclusive or to obey to a first match policy. We leave it as an exercise to the reader.

As usual, the type system assigns a type to messages (i.e. channels) and checks well-typing of processes. It is very compact and we report it below

$$\frac{}{\Gamma \vdash c^t : ch(t)}\ \text{(chan)} \qquad \frac{}{\Gamma \vdash x : \Gamma(x)}\ \text{(var)} \qquad \frac{\Gamma \vdash \alpha : s \leq_{\mathscr{B}} t}{\Gamma \vdash \alpha : t}\ \text{(subsum)}$$

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

$$\begin{smallmatrix} t \leq \bigvee_{i\in I} t_i \\ t_i \wedge t \neq \mathbb{0} \end{smallmatrix}\ \frac{\Gamma \vdash \alpha : ch^+(t) \quad \Gamma, x : t_i \vdash P_i}{\Gamma \vdash \sum_{i\in I}\alpha(x:t_i).P_i}\ \text{(input)} \qquad \frac{\Gamma \vdash \beta : t \quad \Gamma \vdash \alpha : ch^-(t)}{\Gamma \vdash \overline{\alpha}(\beta)}\ \text{(output)}$$

The rules are mostly self explaining. The only one that deserves some comments is (input), which checks that the channel α can be used to read, and that each branch is well-typed. Note the two side conditions of the rule: they respectively require that for every t message arriving on α there must be at least one branch able to handle it (thus $t \leq \bigvee_{i\in I} t_i$ forces summands to implement exhaustive type-cases), and that for every branch there must be some message that may select it (thus the $t_i \wedge t \neq \mathbb{0}$ conditions ensure the absence of dead branches). Also note that in the subsumption rule we have used the subtyping relation induced by the bootstrap model (the one we outlined in the previous step) and that in rule (new) the environment is the same in the premise and the conclusion since we restrict channels and not variables.

Step 5. The last step consists in verifying whether the set of values induces the same subtyping relation as the bootstrap model, which is indeed the case. Note that here the only values are the channel constants and that they are typed just by two rules, (chan) and (subsum). Note also that, contrary to the λ-calculus, we do not need to explicit consider in the typing rule intersections with negated types. This point will be discussed in Section 4.6.

4 Challenges, perspectives, and open problems

In the previous sections we gave a brief overview of the semantic subtyping approach and a couple of instances of its application. Even though we have done drastic simplifications to the calculi we have considered, this should have given a rough idea of the basic mechanisms and constitute a sufficient support to understand the challenges, perspectives, and open problems we discuss next.

4.1 Atomic types

We have shown that in order to decide the subtyping relation in $\mathbb{C}\pi$ one must be able to decide the atomicy of the types (more precisely, one must be able to decide whether a type contains a finite number of atomic types and, if this is the case, to enumerate them). Quite surprisingly the same problem appears in λ-calculus (actually, in any semantic subtyping based system) as soon as we try to extend it with polymorphic types. Imagine that we embed our types with type variables $X, Y, \ldots$. Then the "natural" (semantic) extension of the subtyping relation is to quantify the interpretations over all substitutions for the type variables:

$$t_1 \leq t_2 \quad \overset{\text{def}}{\iff} \quad \forall s. [\![t_1[s/X]]\!] \subseteq [\![t_2[s/X]]\!] \,. \tag{15}$$

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

$$(t, X) \leq (t, \neg t) \vee (X, t). \tag{16}$$

We may need to check such an inequality when type-checking the body of a function polymorphic in X where we apply a function whose domain is the type on the right to an argument of the type on the left.

It is easy to see that this inequality holds if and only if t is atomic. If t is not atomic, then it has at least one non-empty proper subtype, and (15) does not hold when we substitute this subtype for X. If instead t is atomic, then for all X either $t \leq X$ or $t \leq \neg X$: if the second case holds then X is contained in $\neg t$, thus the left hand type of the inequality is included in the first clause on the right hand type. If X does contain t, then all the elements on the left except those in (t,t) are included by the first clause on the right, and the elements in (t,t) are included by the second clause.

Note that the example above does not use any fancy or powerful type constructor, such arrows or channels: it only uses products and type variables. So the problem we exposed is not a singularity but applies to all polymorphic extensions of semantic subtyping where, once more, deciding subtyping reduces to deciding whether some type is atomic or not.

Since these atomic types pop out so frequently a first challenge is understanding why this happens and therefore how much the semantic subtyping technique is tightened to the decidability of atomicity. In particular, we may wonder whether it is reasonable and possible to study systems in which atomic types are not denotable so that the set of subtypes of a type is dense.

4.2 Polymorphic types

The previous section shows that the atomic types problem appears both in $\mathbb{C}\pi$ and in the study of polymorphism for $\mathbb{C}$Duce. From a theoretical viewpoint this is not a real problem: in the former case Castagna *et al.* [CNV05] show atomicity to be decidable, which implies the decidability of the subtyping relation; in the functional case Hosoya *et al.* [HFC05] argue that the subtyping relation based on the substitution interpretation can be reduced to the satisfiability of a constraint system with negative constraints. However, from a practical point of view the situation is way more problematic, probably not in the case of $\mathbb{C}\pi$, since it is not intended for practical immediate applications, but surely is in the case targeted by Hosoya *et al.* since the study performed there is intended to be applied to programming languages, and no practical algorithm to solve this case is known. For this reason in [HFC05] a different interpretation of polymorphic types is given. Instead of interpreting type variables as “place holders” where to perform substitutions, as stated by equation (15), they are considered as “marks” that indicate the parametrised subparts of the values. The types are then interpreted as sets of marked values (that is, usual values that are marked by type variables in the correspondence of where these variables occur in the type), and are closed by mark erasure. Once more, subtyping is then set containment (of mark-erasure closed sets of marked values) which implies that the subtyping relation must either preserve the parametrisation (i.e. the marks), or eliminate part of it in the supertype. More specifically, this requires that the type variables in the supertype are present in the same position in the subtype. This rules out critical cases such as the one of equation (16) which does not hold since no occurrence of X in the type on the left of the equation corresponds to the X occurring in the type on the right.

Now, the marking approach works because the only type constructor used in [HFC05] is the product type, which is covariant. This is enough to model XML types, and the polymorphism one obtains can be (and actually is) applied to the XDuce language. However, it is still matter of research how to implement the marking approach in the presence of contravariant type constructors: first and foremost in the presence of arrow types, as this would yield the definition of a polymorphic version of $\mathbb{C}$Duce; but also, it would be interesting to study the marking approach in the presence of the contravariant ch^{-} type constructor, to check how it combines with the checks of atomicity required by the mix with the ch^{+} constructor, and see whether markings could suggests a solution to avoid this check of atomicity.

On a different vein it would be interesting to investigate the relation between parametricity (as intended in [Rey83,ACC93,LMS93]) and the marking approach to polymorphism. According to parametricity, a function polymorphic (parametric) in a type variable X cannot look at the elements of the argument which have type X, but must return them unmodified. Thus with respect to those elements, the function is just a rearranging function and it behaves uniformly on them whatever the actual type of these elements is. Marks have more or less the same usage and pinpoint those parts of the argument that must be used unchanged to build the result (considering the substitution based definition of the subtyping relation would correspond to explore the semantic properties of the type parameters, as the example of equation (16) clearly shows). However, by the presence of "ad hoc" polymorphism (namely, the type-case construction evocated in Footnote 5 and discussed in Section 4.5) the polymorphic functions in [HFC05] can look at the type of the parametric (i.e. marked) parts of the argument, decompose it, and thus behave differently according to the actual type of the argument. Therefore, while the marking approach and parametric polymorphism share the fact that values of a variable type are never constructed, they differ in presenting a uniform behaviour whatever the type instantiating a type variable is.

Another direction that seems worth pursuing is to see if it is possible to recover part of the substitution based polymorphic subtyping as stated by equation (15), especially in $\mathbb{C}\pi$ where the test of atomicity is already necessary because of the presence of the covariant and contravariant cones.

Finally, one can explore a more programming language oriented approach and check whether it is possible to define reasonable restrictions on the definition of polymorphic functions (for instance by allowing polymorphism to appear only in top-level functions, by forbidding a type variable to occur both in covariant and in contravariant position, by constraining the use of type variables occurring in the result type of polymorphic functions, etc.) so that the resulting language provides the programmer with sufficient polymorphism for practical usage, while keeping it simple and manageable.

4.3 The nature of semantic subtyping

The importance of atomic types also raises the question about the real nature of the semantic subtyping, namely, is semantic subtyping just a different way to axiomatise a subtyping relation that could be equivalently axiomatised by classic syntactic techniques, or is it something different? If we just look at the $\mathbb{C}$Duce case, then the right answer seems to be the first one. As a matter of facts, we could have probably arrived to define the same system without resorting to the bootstrapping technique and the semantic interpretation, but just finding somehow the formulae (6) and (7) and distributing them at the premises of some inference rules in which the types (4) and (5) are equated to $\mathbb{0}$. Or alternatively we could have arrived to this system by looking at the axiomatisation for the positive fragment of the $\mathbb{C}$Duce type system given in [DCFGM02], and trying to extend it to negation types.

But if we consider $\mathbb{C}\pi$, then we are no longer sure about the right answer. In Section 3 we just hinted at the fact that checking subtyping involves checking whether some types are atomic, but we did not give further details. The complete specification can be found in Theorem 2.6 of [CNV05], and involves the enumeration of all the atomic types of a finite set. Looking at that definition, it is unclear whether it can be syntactically axiomatised, or defined with a classical deduction system. Since the relation is proved to be decidable, then probably the answer is yes. But it is clear that finding such a solution without resorting to semantic techniques would have been very hard, if not impossible. And in any case one wonders whether in case of a non decidable relation this would be possible at all.

As a matter of facts, we do not know whether the semantic subtyping approach is an innovative approach that yields to the definition of brand new type systems or it is just a new way to define old systems (or rather, systems that could be also defined in the good old syntactic way). Whichever the answer is, it seems interesting trying to determine the limits of the semantic subtyping approach, that is, its degree of freedom. More precisely, the technique to "close the circle" introduced in [FCB02] and detailed in [CF05] is more general than the one presented here in the introduction. Instead of defining a particular model it is possible to characterise a class of models which are those that induce the same containment relation as the intended semantics (that is, that satisfy an equation such as (10) but customised for the type constructors at issue). This relies on the definition of an auxiliary function—called the extensional interpretation [FCB02]—which fixes the intended semantics for the type constructors. So a more technical point is to investigate whether and to which extent it is possible to systematise the definition of the extensional interpretation. Should one start with a given model of values and refine it, or rather try to find a model and then generalise it? And what are the limits of such a definition? For instance, is it possible to define an extensional interpretation which induces a containment where the inequality (14) is an equality? And more generally is it possible to characterise, even roughly, the semantic properties that could be captured by a model? Because, as we show in the next section, there are simple extensions of the type systems we met so far for which a model does not exist.

4.4 Recursive types and models

As we said at the end of the introduction, to complete the 5 steps of the semantic subtyping approach is far from being trivial. One of the main obstacles, if not the main one, may reside in the definition of a model. Not only that the model may be hard to find, but also that sometimes it does not exist. A simple illustration of this can be given in $\mathbb{C}\pi$. In the first step of the definition of $\mathbb{C}\pi$ in Section 3 we carefully specified that the types were not recursive: as pointed out in [CNV05], if we allow recursion inside channel types, then there does not exist any model. To see why, consider the following recursive type:

$$t = \mathtt{int} \vee (ch(t) \wedge ch(\mathtt{int}))\ .$$

If we had a model, then either $t = \mathtt{int}$ or $t \neq \mathtt{int}$ hold. Does $t = \mathtt{int}$? Suppose it does, then $ch(t) \wedge ch(\mathtt{int}) = ch(\mathtt{int})$ and $\mathtt{int} = t = \mathtt{int} \vee ch(\mathtt{int})$, which is not true since $ch(\mathtt{int})$ is not contained in $\mathtt{int}$. Therefore it must be $t \neq \mathtt{int}$. According to our semantics this implies $ch(t) \wedge ch(\mathtt{int}) = \mathbb{0}$, because they are interpreted as two distinct singletons (whence the invariance of ch types). Thus $t = \mathtt{int} \vee \mathbb{0} = \mathtt{int}$, contradiction. The solution is to avoid recursion inside channel types, for instance by requiring that on every infinite branch of a regular type there are only finitely many occurrences of the channel type constructors. Nevertheless, this is puzzling since the natural extension with recursion is inconsistent.

It is important to notice that this problem is not a singularity of $\mathbb{C}\pi$: it also appears in $\mathbb{C}$Duce as soon as we extend its type system by reference types, as explained in [CF05]. This raises the problem to understand what the non-existence of a model means. Does it correspond to a limit of the semantic subtyping approach, a limit that some different approach could overcome, or does it instead characterise some mathematical properties of the semantics of the types, by reaching the limits that every semantic interpretation of these types cannot overcome?

Quite remarkably the restriction on recursive types in $\mathbb{C}\pi$ can be removed, by moving to a *local* version of the calculus [Mer00], where only the output capability of a channel can be communicated. This can be straightforwardly obtained by restricting the syntax of input processes so that they only use channel constants (that is, $\sum_{i\in I} c^t(x:t_i)P_i$ instead of $\sum_{i\in I}\alpha(x:t_i)P_i$), which makes the type $ch^+(t)$ useless. Without this type, the example at the beginning of this section cannot be constructed (by removing $ch^+(t)$ we also remove $ch(t)$ which is just syntactic sugar) and indeed it is possible build a model of the types with full recursion. The absence of input channel types makes also the decision algorithm considerably simpler since equation (13) becomes:

$$ch^-(s) \;\leq\; \bigvee_{k\in K} ch^-(t_k) \tag{17}$$

and it is quite easy to check (e.g. graphically) that (17) holds if and only if there exists $k \in K$ such that $t_k \leq s$. Last but not least, the types of the local version of of $\mathbb{C}\pi$ are enough to encode the type system of $\mathbb{C}$Duce (this is shown in Section 4.7). However, as we discuss in Section 4.6, new problems appear (rather, old problems reappear). So the approach looks like too a short blanket, that if you pull it on one side uncovers other parts and seems to reach the limits of the type system.

4.5 Type-case and type annotations

Both $\mathbb{C}$Duce and $\mathbb{C}\pi$ make use of type-case constructions. In both cases the presence of a type-case does not look strictly necessary to the development, but it is strongly supported, if not induced, by some semantic properties of the models. We already discussed these points while presenting the two calculi.

For $\mathbb{C}$Duce we argued that equation (8) and in particular the fact that the subtyping inequality it induces

$$(t_1 \vee t_2) \to (s_1 \wedge s_2) \lneqq (t_1 \to s_1) \wedge (t_2 \to s_2) \tag{18}$$

is in general strict, suggests the inclusion of overloaded function in the language to distinguish the two types: an overloaded function can have different behaviour for t_1 and t_2, so it can belong to the right hand side, and not to the left hand side where all the functions uniformly return results in the intersection of s_1 and s_2. Of course, from a strict mathematical point of view it is not necessary for a function to be able to distinguish on the type of their argument in order to be in the right hand side but not in the left one: it suffices that it takes, say, a single point of t_1 into s_1/s_2 to be in the difference of the two types. If from a mathematical viewpoint this is the simplest solution from a programming language point of view, this is not the easy way. Indeed we want to be able to program this function (as long as we want that the model based on values induces the same subtyping relation as the bootstrap model). Now imagine that t_1 and t_2 are function types. Then a function which would have a different behaviour on just one point could not be programmed by a simple equality check on the input (such as "if the input is the point at issue then return a point in s_1/s_2 otherwise return something in s_2") as we cannot check equality on functions: the only thing that we can do is to apply them. This would imply a non trivial construction of classes of functions which have a distinguished behaviour on some specific points. It may be the case that such construction does not result technically very difficult (even if the presence of recursive types suggests the contrary). However constructing it would not be enough since we should also type-check it and, in particular, to prove that the function is typed by the difference of the two types in (18): this looks as an harder task. From a programming language perspective the easy mathematical solution is the difficult one, while the easy solution, that is the introduction of type-cases and of overloaded functions, has a hard mathematical sense (actually some researchers consider it as a mathematical non-sense).

For $\mathbb{C}\pi$ we raised a similar argument about the strictness of

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

The presence of a type-case in the processes is not strictly necessary to the existence of the model (values do not involve processes but just messages) but it makes the two types observationally distinguishable. One could exclude the type-case from the language, but then we would have a too restrictive subtyping relation because it would not let values in the right type to be used where values of the left type are expected, even if the two types would not be operationally distinguishable: it would be better in that case to have the equality hold (as in the system defined by Hennessy and Riely [HR02] where no type-case primitive is present).

These observations make us wonder how much the semantic subtyping approach is bound to the presence of a type-case. We also see that if for instance in $\mathbb{C}$Duce we try to provide a language without overloading, the formal treatment becomes far more difficult (see Section 5.6 of [Fri04]). Therefore one may also wonder whether the semantic subtyping approach is unfit to deal with languages that do not include a type case. Also, since we have a type-case, then we annotated explicitly by their type some values: λ-abstractions in $\mathbb{C}$Duce and channel constants in $\mathbb{C}\pi$. One may wonder if any form of partial type reconstruction is possible[8], and reformulate the previous question as whether the semantic subtyping approach is compatible with any form of type reconstruction.

The annotations on the λ-abstractions raise even deeper questions. Indeed all the machinery on λ-calculus works because we added these explicit annotations. The point is that annotations and abstractions constitute an indissociable whole, since in the presence of a type-case the annotations observably change the semantics of the abstractions: using two different annotations on the same λ-abstraction yields two different behaviours of the program they are used in. For instance $\lambda^{Odd \to Odd \wedge Even \to Even} x.x$ will match a type-case on $Odd \to Odd$ while $\lambda^{Int \to Int} x.x$ will not. We are thus deeply changing the semantics of the elements of a function space, or at least of the λ-terms as usually intended. This raises a question which is tighten to—but much deeper than—the one raised by Section 4.3, namely which is the mathematical or logical meaning of the system of $\mathbb{C}$Duce, and actually, is there any? A first partial answer to this question has been answered by Dezani *et al.* [DCFGM02] who showed that the subtyping relation induced by the model of Section 2 restricted to its positive part (that is arrows, unions, intersections but no negations) coincides with the relevant entailment of the $\mathbf{B}_+$ logic (defined 30 years before). However, whether analogous characterisations of the system with negation exist is still an open question. This seems a much more difficult task since the presence of negation requires deep modifications in the semantics of functions and in their typing. Thus, it still seems unclear whether the semantic subtyping technique for λ-calculus is just a syntactic hack that makes all the machinery work, or it hides some underlying mathematical feature we still do not understand.

4.6 Language with enough points and deduction of negations

We have seen in step 4 of Section 2, that lambda abstractions are typed in an unorthodox way, since the rule (abstr) can subtract any finite number of arrow types as long as the type is non-empty. In step 5 of the same section we justified this rule by the fact that we wanted a language that provided enough values to inhabit all the non-empty types. This property is important for two reasons: (i) if the bootstrap model and the model of values induce the same subtyping relation, then it is possible to consider types as set of values, which is an easier concept

[8] Full type reconstruction for $\mathbb{C}$Duce is undecidable, since it already is undecidable for the λ-calculus with intersection types where typability is equivalent to strong normalizability.

to reason on (at least for a programmer) than the bootstrap model, and (ii) if two different types cannot be distinguished by a value than we would have too a constraining type system, since it would forbid to interchange the values of the two types even though the types are operationally indistinguishable.

The reader may have noticed that we do not have this problem in $\mathbb{C}\pi$. Indeed given a value, that is a channel c^t, it is possible to type it with the negation of all channel types that cannot be deduced for it. In particular we can deduce for c^t the types $\neg ch^{+}(s_1)$ and $\neg ch^{-}(s_2)$ for all $s_1 \not\geq t$ and $s_2 \not\leq t$, and this is simply obtained by subsumption, since it is easy to verify that all these types are supertypes of the minimum type deduced for c^t that is $ch^{+}(t) \wedge ch^{-}(t)$. For instance if $s_1 \not\geq t$ then $ch^{-}(t) \leq \neg ch^{+}(s_1)$ and so is the intersection.

But subsumption does not work in the case of $\mathbb{C}$Duce: to deduce by subsumption that $\lambda^{Int \to Int} x.(x+3) : \neg(Bool \to Bool)$ one should have $Int \to Int \leq \neg(Bool \to Bool)$, which holds if and only if $Int \to Int \wedge Bool \to Bool$ is empty, which is not since it contains the overloaded functions of the corresponding type.

Interestingly, the same problem pops up again if we consider the local version of $\mathbb{C}\pi$. In the absence of covariant channels it is no longer possible to use the same rules to deduce that c^t has type $\neg ch^{-}(s)$ for $s \not\leq t$. Indeed we can only deduce that $c^t : ch^{-}(t)$ and this is *not* a subtype of $\neg ch^{-}(s)$ (since $ch^{-}(t) \wedge ch^{-}(s)$ is non-empty: it contains $c^{s \vee t}$), thus subsumption does not apply and we have to modify the rule for channels, so that is uses the same techniques as (abstr):

$$\frac{t_i \not\leq t}{\Gamma \vdash c^t : ch^{-}(t) \wedge \neg ch^{-}(t_1) \wedge \ldots \wedge \neg ch^{-}(t_n)} \text{ (chan)}$$

Having rules of this form is quite problematic. First because one loses the simplicity and intuitiveness of the approach, but more importantly because the system no longer satisfies the minimum typing property, which is crucial for the existence of a typing algorithm. The point is that the minimum "type" of an abstraction is the intersection of the type at its index with all the negated arrow types that can be deduced for it. But this is not a type since it is an infinite intersection, while in our type system only finite intersections are admitted.[9] In order to recover the minimum typing property and define a type algorithm, Alain Frisch [Fri04] introduces some syntactic objects, called schemas, that represent

[9] Of course the problem could be solved by annotating values (channels or λ-abstractions) also with negated types and considering it as the minimum type of the value. But it seems to us an aberration to precisely state all the types a term does not have, especially from a programming point of view, since it would require the programmer to forecast all the possible usages in which a function must not have some type. In any case in the perspective of type reconstruction evocated in the previous section this is a problem that must be tackled.

in a finite way the infinite intersection above, but this does not allow the system to recover simplicity and makes it lose its set-theoretic interpretation.

Here it is, thus, yet another problematic behaviour shared between $\mathbb{C}$Duce and $\mathbb{C}\pi$. So once more the question is whether this problem is a limitation of semantic subtyping or it is implicit in the usage of negation types. And as it can be "solved" in the case case of $\mathbb{C}\pi$ by considering the full system instead of just the local version, is it possible to find similar (and meaningful) solutions in other cases (notably for $\mathbb{C}$Duce)?

Finally, it would be interesting to check whether the semantic subtyping type system could be used to define a denotational semantics of the language by relating the semantic of an expression with the set of its types and (since our type system is closed by finite intersections and subsumption) build a filter model [BCD83].

4.7 The relation between $\mathbb{C}\pi$ and $\mathbb{C}$Duce

There exist several encodings of the λ-calculus into the π-calculus (see part VI of [SW02] for several examples), so it seems interesting to study whether the encoding of $\mathbb{C}$Duce into $\mathbb{C}\pi$ is also possible. In particular we would like to use a continuation passing style encoding as proposed in Milner's seminal work [Mil92] according to which a λ-abstraction is encoded as a (process with a free) channel that expects two messages, the argument to which the function must be applied and a channel on which to return the result of the application. Of course, in the $\mathbb{C}$Duce/$\mathbb{C}\pi$ case a translation would be interesting only if it preserved the properties of the type system, in particular the subtyping relation. In other terms, we want that our translation $\{\!\{\}\!\} : \mathbf{Types}_{\mathbb{C}\mathrm{duce}} \to \mathbf{Types}_{\mathbb{C}\pi}$ satisfies the property $t = \mathbb{0}$ if and only if $\{\!\{t\}\!\} = \mathbb{0}$ (or equivalently $s \leq t$ iff $\{\!\{s\}\!\} \leq \{\!\{t\}\!\}$).

Such an encoding can be found in a working draft we have been writing with Mariangiola Dezani and Daniele Varacca [CDV05]. The work presented there starts from the observation that the natural candidate for such an encoding, namely the typed translation used in [SW02] for $\lambda V^{\to}$ (the simply-typed call-by-value λ-calculus) and defined as $\{\!\{s \to t\}\!\} = ch^{-}(\{\!\{s\}\!\} \times ch^{-}(\{\!\{t\}\!\}))$ does not work for $\mathbb{C}$Duce/$\mathbb{C}\pi$ (from now on we will omit the inner mapping parentheses and write $ch^{-}(s \times ch^{-}(t))$ instead). This can be seen by considering that the following equality holds in $\mathbb{C}$Duce

$$s \to (t \wedge u) \quad = \quad (s \to t) \wedge (s \to u) \tag{19}$$

while if we apply the encoding above, the translation of the left hand side is a subtype of the translation of the right hand side but not viceversa. Once more, this is due to the strictness of some inequality, since the translation of the codomain of the left hand side $ch^{-}(t \wedge u)$, contains the translation of the codomains of the right hand side $ch^{-}(t) \vee ch^{-}(u)$ (use equation (11) and distribute union over product) but not viceversa.

So the idea of [CDV05] is to translate $s \to t$ as $ch^{-}(s \times ch^{\lambda}(t))$ where $ch^{\lambda}(t)$ is a type that is (i) contravariant (since it must preserve the covariance of arrow codomains), (ii) satisfies $ch^{\lambda}(t \wedge u) = ch^{\lambda}(t) \vee ch^{\lambda}(u)$ (so that it preserves equation (19)) and (iii) is a supertype of $ch^{-}(t)$ (since we must be able to pass on it the channel on which the result of the function is to be returned).

Properties (i) and (ii) can be satisfied by adding a double negation as for $\neg ch^{-}(\neg t)$: the double negation preserves the contravariance of ch^{-} while the inner negation by De Morgan's laws yields the distributivity of the union. For (iii) notice that $\neg ch^{-}(\neg t) \setminus ch^{-}(t) = ch(\mathbb{1})$, so it suffices to add the missing point by defining $ch^{\lambda}(t) = \neg ch^{-}(\neg t) \vee ch(\mathbb{1})$. With such a definition the translation $ch^{-}(s \times ch^{\lambda}(t))$ has the wanted properties. Actually, in [CDV05] it is proved that the three conditions above are necessary and sufficient to make the translation work, which induces a class of possible translations parametric in the definition of ch^{λ} (see [CDV05] for a characterisation of the choices for ch^{λ}). $ch^{\lambda}(t)$ is a supertype of $ch^{-}(t)$ but the latter is also the greatest channel type contained in $ch^{\lambda}(t)$. So there is gap between $ch^{\lambda}(t)$ and $ch^{-}(t)$ which constitutes the definition space of $ch^{\lambda}(t)$. What is the meaning of this gap, that is of $ch^{\lambda}(t) \setminus ch^{-}(t)$? We do not know, but it is surely worth of studying, since it has important consequences also in the interpretation of terms. The translation of terms is still work in progress, but we want here hint at our working hypotheses since they outline the importance of $ch^{\lambda}(t) \setminus ch^{-}(t)$. We want to give a typed translation of terms, where the translation of a term e of type t is a process with only one unrestricted channel α of type $ch^{-}(\{\!\{t\}\!\})$ (intuitively, this is the channel on which the process writes the result of e). We note this translation as $\{\!\{e\}\!\}_{\alpha}$. Consider the rule (abstr) for functions and note that the body of an abstraction is typed several times under several assumptions. If we want to be able to prove that the translation preserves typing, then the translation must mimic this multiple typing. This can be done in $\mathbb{C}\pi$ by using a summation, and thus by translating $\lambda^{\wedge_{i\in I} s_i \to t_i} x.e$ into a process that uses the unrestricted channel $\alpha : ch^{-}(\{\!\{\wedge_{i\in I} s_i \to t_i\}\!\}) = ch^{-}(ch^{-}(\vee_{i\in I}(s_i \times ch^{\lambda}(t_i))))$ as follows:

$$\{\!\{\lambda^{\wedge_{i\in I} s_i \to t_i} x.e\}\!\}_{\alpha} = (\nu f^{\vee_{i\in I}(s_i \times ch^{\lambda}(t_i))})(\overline{\alpha}(f) \parallel !\textstyle\sum_{i\in I} f(x : s_i, r : ch^{-}(t_i))\{\!\{e\}\!\}_r)$$

Unfortunately, the translation above is not correct since it is not exhaustive. More precisely, it does not cover the cases in which the second argument is in $ch^{\lambda}(t_i) \setminus ch^{-}(t_i)$: to type $\{\!\{e\}\!\}_r$ the result channel r must have type $ch^{-}(t_i)$ (since the only types the $\mathbb{C}$Duce type system deduces for e are the t_i's), but the encoding of arrow types uses $ch^{\lambda}(t_i)$ in second position. Thus, it seems important to understand what the difference above means. Is it related to the negation of arrow types in the (abstr) rule? Note that in this section we worked on the local version of $\mathbb{C}\pi$, so we have recursive types (Section 4.4) but also negated channels in the (chan) typing rule (Section 4.6). If for local $\mathbb{C}\pi$ we use the rule without negations then $ch^{\lambda}(t) \setminus ch^{-}(t) = \varnothing$, so the encoding above works but we no longer have a consistent type system. We find again that our blanket is too short to cover all the cases. Is this yet another characterisation of some limits of the approach? Does it just mean that Milner's translation is unfit to model overloading? Or does it instead suggest that the encoding of $\mathbb{C}$Duce into $\mathbb{C}\pi$ has not a lot of sense and that we had better study how to integrate $\mathbb{C}$Duce and $\mathbb{C}\pi$ in order to define a concurrent version of $\mathbb{C}$Duce?

4.8 Dependent types

As a final research direction for the semantic subtyping we want to hint at the research on dependent types. Dependent types raise quite naturally in type systems for the π-calculus (e.g. [YH00,Yos04]), so it seems a natural evolution of the study of $\mathbb{C}\pi$. Also, dependent types in the π-calculus are used to check the correctness of cryptographic protocols (see the research started by Gordon and Jeffrey [GJ01]) and unions, intersections, and negations of types look very promising to express properties of programs. Thus, it may be worth of study the definition of an extension of Gordon an Jeffrey systems with semantic subtyping, especially in the light of the connection of the latter with XML and the use of this in protocols for webservices.

Also quite interesting would be to study the extension of first order dependent type theory $\lambda\Pi$ [HHP93]. As far as we know, all the approaches to add subtyping to $\lambda\Pi$ are quite syntactic, since the subtyping relation is defined on β_2 normal forms (see for instance [AC96] or, for an earlier proposal, [Pfe93]). Even more advanced subtype systems, such as [CC01,Che99], still relay on syntactic properties such as the strong normalisation of the β_2-reduction, since the subtyping rules essentially mimic the β_2-reduction procedure. It would then be interesting to check whether the semantic subtyping approach yields a more semantic characterisation of the subtyping relation for dependent types.

5 Conclusion

The goal of this article was twofold: (i) to give an overview of the semantic subtyping approach and an aperçu of its generality by applying it both to sequential and concurrent systems and (ii) to show the new questions it raises. Indeed we were much more interested in asking questions than giving answers, and it is in this perspective that this paper was written. Some of the questions we raised are surely trivial or nonsensical, some others will probably soon result as such, but we do hope that at least one among them will have touched some interesting mathematical problem being worth of pursuing. In any case we hope to have interested the reader in this approach.

Acknowledgements. Most of this work was prepared while visiting Microsoft Research in Cambridge and it greatly benefited of the stimulating and friendly environment that I enjoyed during my stay, a clear break with respect to my current situation. In particular, I want to warmly thank Nick Benton (Section 4.5 results from a long discussion I had with him), Gavin Bierman, Luca Cardelli, and Andy Gordon. I want also to thank Mario Coppo, Mariangiola Dezani, and Daniele Varacca for their suggestions on various versions of this work. All my gratitude to Microsoft Research and all its personnel for their warm hospitality.

References

[AC96] D. Aspinall and A. Compagnoni. Subtyping dependent types. In *11th Ann. Symp. on Logic in Computer Science*, pages 86–97, 1996.

[ACC93] M. Abadi, L. Cardelli, and P.-L. Curien. Formal parametric polymorphism. *Theoretical Computer Science*, 21:9–58, 1993. Special issue in honour of Corrado Böhm.

[APP91] Martín Abadi, Benjamin Pierce, and Gordon Plotkin. Faithful ideal models for recursive polymorphic types. *International Journal of Foundations of Computer Science*, 2(1):1–21, March 1991. Summary in Fourth Annual Symposium on Logic in Computer Science, June, 1989.

[AW93] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the Seventh ACM Conference on Functional Programming and Computer Architecture*, pages 31–41, Copenhagen, Denmark, June 93.

[BCD83] H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983.

[CC01] G. Castagna and G. Chen. Dependent types with subtyping and late-bound overloading. *Information and Computation*, 168(1):1–67, 2001.

[CDV05] G. Castagna, M. Dezani, and D. Varacca. Encoding ℂDuce into ℂπ. Working draft, February 2005.

[CF05] G. Castagna and A. Frisch. A gentle introduction to semantic subtyping. In Proceedings of *PPDP '05, the 7th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming*, Lisboa, Portugal, 2005. ACM Press. Joint ICALP-PPDP keynote talk.

[Che99] G. Chen. Dependent type system with subtyping. *Journal of Computer Science and Technology*, 14(1), 1999.

[CNV05] G. Castagna, R. De Nicola, and D. Varacca. Semantic subtyping for the π-calculus. In *LICS '05, 20th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 2005.

[Dam94] F. Damm. Subtyping with union types, intersection types and recursive types II. Research Report 816, IRISA, 1994.

[DCFGM02] M. Dezani-Ciancaglini, A. Frisch, E. Giovannetti, and Y. Motohama. The relevance of semantic subtyping. In *Intersection Types and Related Systems*. Electronic Notes in Theoretical Computer Science 70(1), 2002.

[FCB02] 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.

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

[GJ01] A. Gordon and A. Jeffrey. Authenticity by typing for security protocols. In *CSFW 2001: 14th IEEE Computer Security Foundations Workshop*, pages 145–159, 2001.

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

[HHP93] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[Hos01] Haruo Hosoya. *Regular Expression Types for XML*. PhD thesis, The University of Tokyo, 2001.

[HP01] Haruo Hosoya and Benjamin C. Pierce. Regular expression pattern matching for XML. In *The 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2001.

[HP03] H. Hosoya and B. Pierce. XDuce: A typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.

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

[LMS93] Giuseppe Longo, Kathleen Milsted, and Sergei Soloviev. The genericity theorem and parametricity in the polymorphic λ-calculus. *Theor. Comput. Sci.*, 121(1-2):323–349, 1993.

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

[Mil92] R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[Pfe93] F. Pfenning. Refinement types for logical frameworks. In *Informal Proceedings of the 1993 Workshop on Types for Proofs and Programs*, May 1993.

[Rey83] J.C. Reynolds. Types, abstractions and parametric polymorphism. In R.E.A. Mason, editor, *Information Processing '83*, pages 513–523. North-Holland, 1983.

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

[YH00] N. Yoshida and M. Hennessy. Assigning types to processes. In *Proc. of the 15th IEEE Symposium on Logic in Computer Science*, pages 334–348, 2000.

[Yos04] Nobuko Yoshida. Channel dependent types for higher-order mobile processes. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 147–160. ACM Press, 2004.