A Gentle Introduction to Semantic Subtyping

Giuseppe Castagna
CNRS
École Normale Supérieure
Paris, France

Alain Frisch
INRIA
Rocquencourt
France

ABSTRACT

Subtyping relations are usually defined either syntactically by a formal system or semantically by an interpretation of types into an untyped denotational model. In this work we show step by step how to define a subtyping relation semantically in the presence of functional types and dynamic dispatch on types, without the complexity of denotational models, and how to derive a complete subtyping algorithm. It also provides a recipe to add set-theoretic union, intersection, and negation types to your favourite language.

The presentation is voluntarily kept informal and discursive and the technical details are reduced to a minimum since we rather insist on the motivations, the intuition, and the guidelines to apply the approach.

Categories and Subject Descriptors: D.3.3 [Programming Languages]: Language Constructs and Features — Data types and structure; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs — Type structure; F.4.1 [Mathematical Logic and Formal Languages]: Mathematical Logic — Lambda calculus and related systems

General Terms: Language, Theory.

Keywords: Typing; Subtyping; Intersection, Union, and Negation Types.

1. Introduction

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, [2, 10]), 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 (we discuss them later on) 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 [18, 17, 16] with the work on XDuce. The key idea is that in order to define the subtyping relation semantically one does not need to start from a model of the whole language: a model of the types suffices. In particular Hosoya and Pierce take as model of types the set of values of the language. Their notion of model cannot capture functional values. On the one hand, the resulting type system is poor since it lacks function types. On the other hand, it manages to integrate union, product and recursive types and still keep the presentation of the subtyping relation and of the whole type system quite simple.

In [12, 11], together with Véronique Benzaken, we extended the work on XDuce and reframed it in a more general setting: we show a technique to define semantic subtyping in the presence of a rich type system including function types, but also arbitrary boolean combinations (union, intersection, and negation types) and in the presence of lately bound overloaded functions and type-based pattern matching. The aim of [12, 11] was to provide a theoretical foundation on the top of which to build the language ℂDuce [6], an XML-oriented transformation language. This motivation needed a rather heavy technical development that concealed a side—but important—contribution of the work, namely a generic and uniform technique (or rather, a cookbook of techniques) to define semantic subtyping when straightforward set-theoretic interpretation does not work, in particular for arrow types. Here we concentrate on this second aspect of the work: we get rid of many features (e.g. patterns and pattern matching, full-fledged overloading, pattern variable type inference,. . .), skip many technical details, and focus on the basic intuition to gradually introduce our approach. This results in a presentation along which we explain the reader how to take her/his favourite set of type constructors (e.g. arrows, but also records, lists, pointers, channels, etc.) and add to it a complete set of boolean combinators: union, intersection and negation types.

The description of a general technique to extend semantic subtyping to general types systems with arrow and complete boolean combinator types is just one way to read our work, and it is the one we decided to emphasise in this presentation. However it is worth mentioning that there exist at least two other readings for the results and techniques presented here.

A first alternative reading is to consider this work as a research on the definition of a general purpose higher-order XML transformation language: indeed, this was the initial motivation of [12, 11] and the theoretical work done there constitutes the fundamental basis for the definition *and the implementation* of the XML transformation language ℂDuce.

A second way of understanding this work is as a quest for the

Joint ICALP/PPDP keynote talk. A four pages abstract of this work is included in the proceedings of ICALP 2005, LNCS n. 3580, Springer, 2005.

PPDP'05, July 11–13, 2005, Lisbon, Portugal.
Copyright 2005 ACM 1-59593-090-6/05/0007 ...$5.00.

generalisation of lately bound overloaded functions to intersections types. The intuition that overloaded functions should be typed by intersection types was always felt but never fully formalised or understood. On the one hand we had the longstanding research on intersection types with the seminal works by the Turin research group on typed lambda calculus [5, 9]. However functions with intersection types had a uniform behaviour, in the sense that even if they worked on arguments of different types they always executed the same code on all of these types[1]. So functions with intersections types looked closer to parametric polymorphism (in which we enumerate the possible domains) rather than overloaded functions which are able to discriminate on the type of the argument and execute a different code for each different type. On the other hand there was the research on overloaded functions as used in programming languages which accounted for functions formed by different pieces of code selected according to the type of the argument the function is applied to. However, even if the types of these functions are apparently close to intersection types, they never had the set theoretic intuition of intersections. So for example in the λ&-calculus [7] overloaded functions have types that are characterised by the same subtyping relation as intersection types, but they differ from the latter by the need of special formation rules that have no reasonable counterpart in intersection types. The overloaded functions defined here and, even more, those defined in [12] finally reconcile the two approaches: they are typed by intersection types (with a classical/set-theoretic interpretation) and their definitions may intermingle code shared by all possible input types with pieces of code that are specific to only some particular input types. Therefore they nicely integrate the two styles of programming.

Finally it is important to stress that although here we deploy our construction for a λ-calculus with higher-order functions, the technique is quite general and can be used mostly unchanged for quite different paradigms, as for instance it is done in [8] for the π-calculus (we discuss it at the end of Section 3.4).

In what follows we will privilege clarity over exhaustiveness. Therefore even though the presentation is correct some technical details are only partially covered: all such gaps can be filled by referring to the technical development of [12, 13, 11].

Our hope is that this work will provide the reader with enough intuition and a detailed roadmap to decide whether it is possible/interesting to endow her/his favourite language with a set-theoretically defined subtyping relation.

2. The intuition

When dealing with syntactic subtyping one usually proceeds as follows. First, one defines a language, then, somewhat independently, the set of (syntactic) types and a subtyping relation on this set. This relation is defined axiomatically, in an inductive (or coinductive in case of recursive types) way. The type system, consisting of the set of types and of the subtyping relation, is coupled to the language by a *typing relation*, usually defined via some typing rules by induction on the terms of the language. The meaning of types is only given by the rules defining the subtyping and the typing relations.

The semantic subtyping approach described here diverges from the above only for the definition of the subtyping relation. Instead of using a set of rules, this relation is defined semantically: we do it by defining a *set-theoretic* model of the types and by stating that one type is subtype of another if the interpretation of the former is a *subset* of the interpretation of the latter. As for syntactic subtyping, the definition is parametric in the set of base types and their subtyping relation (in our case, their interpretation).

2.1 Advantages of semantic subtyping

The semantic approach is more technical and constraining, and this may explain why it has obtained less attention than syntactic subtyping. However it presents several advantages:

1. When type constructors have a natural interpretation in the model, the subtyping relation is by definition complete with respect to its intuitive interpretation as set inclusion: when $t \leq s$ does not hold, it is possible to exhibit an element of the model which is in the interpretation of t and not of s, even in presence of arrow types (this property is used in ℂDuce to return informative error messages to the programmer); in the syntactic approach one can just say that the formal system does not prove $t \leq s$, and there may be no clear criterion to assert that some meaningful additional rules would not allow to prove it. This argument is particularly important with a rich type algebra, where type constructors interact in non trivial ways; for instance, when considering arrow, intersection and union types, one must take into account many distributivity relations, such as, for example, $(t_1 \vee t_2) \rightarrow s \simeq (t_1 \rightarrow s) \wedge (t_2 \rightarrow s)$. Forgetting any of these rules yields a type system that, although sound, does not match (that is, it is not complete with respect to) the intuitive semantics of types.
2. In the syntactic approach deriving a subtyping algorithm requires a strong intuition of the relation defined by the formal system, while in the semantic approach it is a simple matter of "arithmetic": it simply suffices to use the interpretation of types and well-know boolean algebra laws to decompose subtyping on simpler types (as we show in Section 3.2). Furthermore, as most of the formal effort is done with the semantic definition of subtyping, studying variations of the algorithm (e.g., optimisations or different rules) turns out to be much simpler (this is common practise in database theory where, for example, optimisations are derived directly from the algebraic model of data).
3. While the syntactic approach requires tedious and error-prone proofs of formal properties, in the semantic approach many of them come for free: for instance, the transitivity of the subtyping relation is trivial (as set-containment is transitive), and this makes proofs such as cut elimination or transitivity admissibility pointless.

Although these properties seem quite appealing, the technical details of the approach hinder its development: in the semantic approach, one must be very careful not to introduce any circularity in the definitions. For instance, if the type system depends on the subtyping relation—as this is generally the case—one cannot use it to define the semantic interpretation which must thus be untyped; also, usually the model corresponds to an untyped denotational semantics, and types are interpreted as ideals and this precludes the set-theoretic interpretation of negative types (as the complement of ideals is not an ideal). For these reasons all the semantic approaches to subtyping previous to our work presented some limitations: no higher-order functions, no complement types, and so on. The main contribution of our work is the development of a formal framework that overcomes these limitations.

> EXCURSUS. The reader should not confuse our research with the long-standing research on set-theoretic models of subtyping. In that case one starts from a syntactically (i.e. axiomatically) defined subtyping relation and seeks a set-theoretic model where this rela-

[1] A notable exception to this is John Reynolds work on the coherent overloading and the language Forsythe [19, 20].

> tion is interpreted as inclusion. Our approach is the opposite: instead of starting from a subtyping relation to arrive to a model, we start by defining a model in order to arrive to a subtyping relation. Thus in our approach types have a strong substance even before introducing the typing relation.

2.2 A model of types

To define semantic subtyping we need a set-theoretic model of types. The source of most of (if not all) the problems comes from the fact that this model is usually defined by starting from a model of the terms of the language. That is, we consider a denotational interpretation function that maps each term of the language into an element of a semantic domain and we use this interpretation to define the interpretation of the types (typically—but not necessary, e.g. PER models [4]—as the image of the interpretation of all terms of a given type). If we consider functional types then in order to interpret functional term application we have to interpret the duality of functions as terms and as functions on terms. This yields the need to solve complicated recursive domain equations that hardly combines with a set-theoretic interpretation of types, whence the introduction of restrictions in the definition of semantic subtyping (e.g. no function types, no negation types, etc . . .).

Note however that in order to define semantic subtyping all we need is a set-theoretic *model of types*. The construction works even if we do not have a model of terms. To push it to the extreme, in order to define subtyping we do not need terms at all, since we could imagine to define type inclusion for types independently from the language we want to use these types for. More plainly, the definition of a semantic subtyping relation needs neither an interpretation for applications (that is an applicative model) nor, thus, the solution of complicated domain equations.

The key idea to generalise semantic subtyping is then to dissociate the *model of types* from the *model of terms* and define the former independently from the latter. In other words, the interpretation of types must not forcedly be based on, or related to an interpretation of terms (and actually in the some concrete examples we will give we interpret types in structures that cannot be used for an interpretation of terms), and as a matter of fact we do not need an interpretation of terms even to exist for the semantic subtyping construction to go through[2].

2.3 Types as sets of values

Nevertheless, to ensure type safety (i.e. well-typed programs cannot go wrong) the meaning of types has to be somewhat correlated with the language. A classical solution, that belongs to the types folklore[3] is to interpret types as sets of *values*, that is, as the results of *well-typed* computations in the language. More formally, the values of a typed language are all the terms that are well-typed, closed, and in normal form. So the idea is that in order to provide an interpretation of types we do not need an interpretation of all terms of the language (or of just the well-typed ones): the interpretation of the values of the language suffices to define an interpretation of types. This is much an easier task: since a closed application usually denotes a redex, then by restricting to the sole values we avoid the need to interpret application and, therefore, also the need to solve complicated domain equations. This is the solution adopted by XDuce, where values are XML documents and types are sets of documents (more precisely, regular languages of documents).

But if we consider a language with arrow types, that is a language with higher order functions, then the applications come back again: arrow types must be interpreted as sets of function values, that is, as sets of well-typed closed lambda abstractions, and applications may occur in the body of these abstractions. Here is where XDuce stops and it is the reason why it does not include arrow types.

2.4 A circularity to break

Introducing arrow types is then problematic because it slips applications back again in the interpretation of types. However this does not mean that we need a semantic interpretation for application, it just implies that we must define how application is typed. Indeed functional values are *well-typed* lambda abstractions, so to interpret functional types we must be able to type lambda abstractions and in particular to type the applications that occur in their body. Now this is not an easy task in our context: in the absence of higher order functions the set of values of type constructors such as products or records can be inductively defined from basic types without resorting to any typing relation (this is why the Hosoya Pierce approach works smoothly). With the arrow type constructor, instead, this can be done only by using a typing relation, and this yields to the circularity we hinted at in the introduction and that is shown in Figure 1: in order to define the subtyping relation we need an interpretation of the types of the language; for this we have to define which are the values of an arrow type; this needs that we define the typing relation for applications, which in turns needs the definition of the subtyping relation.

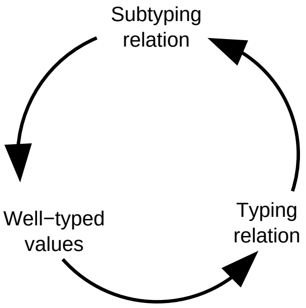

Figure 1: Circularity

Thus, if we want to define the semantic subtyping of arrow types we must find a way the avoid this circularity. The simplest way to avoid it is to break it, and the development we did so far clearly suggests where to break it. We always said that to define (semantic) subtyping we *must* have a model of types; it is also clear that the typing relation *must* use subtyping; on the contrary it is not strictly necessary for our model to be based on the interpretation of values, this is just convenient as it ties the types with the language the types are intended for. This is therefore the weakest link and we can break it. So the idea is to start from a model (of the types) defined independently (but not too much) from the language the types are intended for (and therefore independently from its values), and then from that define the rest: subtyping, typing, set of values. We will then show how to relate the initial model to the obtained language and recover the initial "types as set of values" interpretation: namely, we will "close the circle".

2.5 Set-theoretic models

Let us then show more in details how we shall proceed. We do not need to define a particular language, the definition of types will suffice. Therefore we start from the following syntax for types:

$$t ::= \mathbf{0} \mid \mathbf{1} \mid t \to t \mid t \times t \mid \neg t \mid t \vee t \mid t \wedge t$$

[2] As Pierre-Louis Curien suggested, the construction we propose is a *pied de nez* to (it cocks a snook at) denotational semantics, as it uses a semantic construction to define a language for which, possibly, no denotational semantics is known.

[3] A survey on the "Types" mailing list traces this solution back to Bertrand Russell and Alfred Whitehead's Principia Mathematica. Closer to our interests it seems that the idea independently appeared in the late sixties early seventies and later back again in seminal works by Roger Hindley, Per Martin-Löf, Ed Lowry, John Reynolds, Niklaus Wirth and probably others (many thanks to the many "typers" who answered to our survey).

where **0** and **1** respectively correspond to the empty and universal types (these are sometimes denoted by the pair $\bot$, $\top$ or **Bottom**, **Top**). For the sake of generality, we also consider recursive types. There are several way to formalise them. We can: (1) introduce them with explicit binders $\mu x.t[x]$, or (2) define them as regular trees generated by the grammar above, or (3) define them as the solution of systems of equations. In all cases, we need a contractivity constraint to rule out meaningless expressions such as $t \wedge (t \wedge (t \wedge (\ldots)))$. Namely, we require that every infinite branch has infinitely many occurrences of the $\times$ or of the $\to$ constructors [3]. The formal development is still sound for a system without recursive types, so the reader can follow it by having in mind only inductive types. However, most of the machinery described here is motivated by the presence of recursive types. For instance, the existence of non-universal models (see § 3.1) and thus the existence of different semantic subtyping relations, depends on the joint presence of recursive and arrow types.

The second step is to define precisely what a *set-theoretic* model for these types is. As Hindley and Longo [14] give some general conditions that characterise models of λ-calculus, so here we want to give the conditions that an interpretation function must satisfy in order to characterise a set-theoretic model of our types. So let $\mathscr{T}$ denote the set of (regular and contractive) types generated by the grammar above, $\mathscr{D}$ some domain, and $\llbracket \_ \rrbracket$ an interpretation function from $\mathscr{T}$ to $\mathscr{P}(\mathscr{D})$. The conditions that $\llbracket \_ \rrbracket$ must satisfy to define a set-theoretic model are mostly straightforward, namely:

1. $\llbracket t_1 \vee t_2 \rrbracket = \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket$;
2. $\llbracket t_1 \wedge t_2 \rrbracket = \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket$;
3. $\llbracket \neg t \rrbracket = \mathscr{D} \backslash \llbracket t \rrbracket$;
4. $\llbracket \mathbf{1} \rrbracket = \mathscr{D}$;
5. $\llbracket \mathbf{0} \rrbracket = \varnothing$;
6. $\llbracket t \times s \rrbracket = \llbracket t \rrbracket \times \llbracket s \rrbracket$;

7*. $\llbracket t \to s \rrbracket = \quad ??? \quad$.

The first six conditions convey the intuition that our model is set theoretic: so the intersection of types must be interpreted as set intersection, the union of types as set-theoretic union and so on (the sixth condition requires some closure properties on $\mathscr{D}$ and the definition of embedding functions, but we prefer not to enter in such a level of detail at this point of our presentation and delay detailed presentation to the end of this work). But the definition is not complete yet as we still have to establish the seventh condition (highlighted by a $*$) that constraints the interpretation of arrow types. This condition is more complicated. Again it must convey the intuition that the interpretation is set theoretic, but while the first six conditions are language independent, this conditions strongly depends on the language and in particular on the kind of functions we want to implement in our language. We give detailed examples about this in [13]. The set theoretic intuition we have of function spaces is that a function is of type $t \to s$ if whenever applied to a value of type t it returns a result of type s. Intuitively, if we interpret functions as binary relations on $\mathscr{D}$, then $\llbracket t \to s \rrbracket$ is the set of binary relations in which if the first projection is in (the interpretation of) t then the second projection is in s, namely $\{ f \subseteq \mathscr{D}^2 \mid \forall (d_1, d_2) \in f.\, d_1 \in \llbracket t \rrbracket \Rightarrow d_2 \in \llbracket s \rrbracket \}$. Note that this set is equivalent to $\mathscr{P}(\overline{\llbracket t \rrbracket \times \overline{\llbracket s \rrbracket}})$, where the overline denotes set complement. If the language is expressive enough, we can do as if every binary relation in this set was an element of $\llbracket t \to s \rrbracket$; thus, we would like to say that the seventh condition is:

$$\llbracket t \to s \rrbracket = \mathscr{P}(\overline{\llbracket t \rrbracket \times \overline{\llbracket s \rrbracket}}) \qquad (1)$$

But this is completely meaningless. First, technically, this would imply that $\mathscr{P}(\mathscr{D}^2) \subseteq \mathscr{D}$, which is impossible for cardinality reasons. Also, remember that we want eventually to re-interpret types as sets of values of the language, and functions in the language are *not* binary relations (they are syntactic objects). However what really matters is not the exact mathematical nature of the elements of $\mathscr{D}$, but only the relations they create between types. The idea then is to do *as if* the above condition held.

Since this point is central to our model, let us explain it differently. Recall that the only reason why we want to accurately state what set-theoretic model of types is, is to precisely define the subtyping relation for syntactic types. In other words, we do not define an interpretation of types in order to formally and mathematically state what the syntactic types *mean* but, more simply, we define it in order to state how they are *related*. So, even if we would like to say that a type $t \to s$ must be interpreted in the model as $\mathscr{P}(\overline{\llbracket t \rrbracket \times \overline{\llbracket s \rrbracket}})$ as stated by (1), for what it concerns the goal we are aiming at, it is enough to require that a model must interpret functional types so as the induced subtyping relation is the same as the one the condition (1) would induce. So we will consider as being "set-theoretic" every interpretation function $\llbracket \_ \rrbracket$ such that

$$\llbracket t_1 \to s_1 \rrbracket \subseteq \llbracket t_2 \to s_2 \rrbracket \iff \mathscr{P}(\overline{\llbracket t_1 \rrbracket \times \overline{\llbracket s_1 \rrbracket}}) \subseteq \mathscr{P}(\overline{\llbracket t_2 \rrbracket \times \overline{\llbracket s_2 \rrbracket}})$$

and similarly for any boolean combination of arrow types.

Formally, we associate to $\llbracket \_ \rrbracket$ an extensional interpretation $\mathscr{E}\llbracket \_ \rrbracket$ that behaves as $\llbracket \_ \rrbracket$ except for arrow types, for which we use the condition above as definition:

$$\mathscr{E}\llbracket t \to s \rrbracket = \mathscr{P}(\overline{\llbracket t \rrbracket \times \overline{\llbracket s \rrbracket}})$$

More precisely we have the following definition

Definition 2.1. *Let $\llbracket \_ \rrbracket : \mathscr{T} \to \mathscr{P}(\mathscr{D})$ be an interpretation function. The extensional interpretation of $\llbracket \_ \rrbracket$ is the function $\mathscr{E}\llbracket \_ \rrbracket : \mathscr{T} \to \mathscr{P}(\mathscr{D}^2 + \mathscr{P}(\mathscr{D}^2))$ defined as:*

$$\begin{array}{ll}
\mathscr{E}\llbracket \mathbf{0} \rrbracket = \varnothing & \mathscr{E}\llbracket \mathbf{1} \rrbracket = \mathscr{D}^2 + \mathscr{P}(\mathscr{D}^2) \\
\mathscr{E}\llbracket t_1 \vee t_2 \rrbracket = \mathscr{E}\llbracket t_1 \rrbracket \cup \mathscr{E}\llbracket t_2 \rrbracket & \mathscr{E}\llbracket t_1 \wedge t_2 \rrbracket = \mathscr{E}\llbracket t_1 \rrbracket \cap \mathscr{E}\llbracket t_2 \rrbracket \\
\mathscr{E}\llbracket \neg t \rrbracket = \mathscr{E}\llbracket \mathbf{1} \rrbracket \backslash \mathscr{E}\llbracket t \rrbracket & \mathscr{E}\llbracket t \times s \rrbracket = \llbracket t \rrbracket \times \llbracket s \rrbracket \\
\mathscr{E}\llbracket t \to s \rrbracket = \mathscr{P}(\overline{\llbracket t \rrbracket \times \overline{\llbracket s \rrbracket}}) &
\end{array}$$

□

Note that we use $\llbracket \_ \rrbracket$ in the right-hand side for the $\to$ case, that is, we only re-interpret top-level arrow types. Now we can express the fact that $\llbracket \_ \rrbracket$ behaves (from the point of view of subtyping) as if functions were binary relations. This is obtained by writing the missing seventh condition, not in the form of 7*, but as follows:

7. $\llbracket t \rrbracket = \varnothing \iff \mathscr{E}\llbracket t \rrbracket = \varnothing$

or, equivalently, $\llbracket t_1 \rrbracket \subseteq \llbracket t_2 \rrbracket \iff \mathscr{E}\llbracket t_1 \rrbracket \subseteq \mathscr{E}\llbracket t_2 \rrbracket$.[4]

To put it otherwise, if we wanted an interpretation $\llbracket \_ \rrbracket$ of the types that were faithful with respect to the semantics of the language, then we should require for all t that $\llbracket t \rrbracket = \mathscr{E}\llbracket t \rrbracket$. But for cardinality reasons this is impossible in a set-theoretic framework. However we do not need such a strong constraint on the definition of $\llbracket \_ \rrbracket$ since all we ask to $\llbracket \_ \rrbracket$ is to characterise the *containment* of types, and to that end it suffices to characterise the zeros of $\llbracket \_ \rrbracket$. Indeed

$$s \leq t \iff \llbracket s \rrbracket \subseteq \llbracket t \rrbracket \iff \llbracket s \rrbracket \cap \overline{\llbracket t \rrbracket} = \varnothing \iff \llbracket s \wedge \neg t \rrbracket = \varnothing$$

So instead of asking that $\llbracket \_ \rrbracket$ and $\mathscr{E}\llbracket \_ \rrbracket$ coincide on all points, we require a weaker constraint, namely that they have the same zeros:

$$\llbracket t \rrbracket = \varnothing \iff \mathscr{E}\llbracket t \rrbracket = \varnothing$$

We said that the above seventh condition (actually, the definition of the extensional interpretation) depends on the language the

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

type system is intended for. We show in [13] different variations of this conditions to match different sets of definable transformations. However, we can already see that the condition above accounts for languages in which functions possibly are

1. *Non-deterministic*: since the condition does not prevent the interpretation of a function space to contain a relation with two pairs (d, d_1) and (d, d_2) with $d_1 \neq d_2$.
2. *Non-terminating*: since the condition does not force a relation in $[\![t \to s]\!]$ to have as first projection the whole $[\![t]\!]$. A different reason for this is that every arrow type is inhabited (note indeed that the empty set belongs to the interpretation of every arrow type), so in particular are all the types of the form $t \to \mathbf{0}$; now, all the functions in such types must be always non-terminating on their domain (if they returned a value this would inhabit $\mathbf{0}$).
3. *Overloaded*: this is subtler than the two previous cases as it is a consequence of the fact that condition does not force $[\![(t_1 \vee t_2) \to (s_1 \wedge s_2)]\!]$ to be equal to $[\![(t_1 \to s_1) \wedge (t_2 \to s_2)]\!]$, but just the former to be included in the latter. Imagine indeed that the language at issue does not allow the programmer to define overloaded functions. So it is not possible to define functions that distinguish the types of their argument, and in particular to have a function that when applied to an argument of type t_1 returns a result in s_1 while returns a (possibly different) s_2 result for t_2 arguments. Therefore the only functions in $(t_1 \to s_1) \wedge (t_2 \to s_2)$ are those in $(t_1 \vee t_2) \to (s_1 \wedge s_2)$.

2.6 Bootstrapping the definition

Now that we have defined what a set-theoretic model for our types is, we can choose a particular one that we use to define the rest of the system. Suppose that there exists at least one pair $(\mathscr{D}, [\![]\!])$ that satisfies the conditions of set-theoretic model, and choose any of them, no matter the one. Let us call this model the *bootstrap model*, and denote it by $(\mathscr{B}, [\![]\!]_{\mathscr{B}})$.

This bootstrap model defines a particular subtyping relation on our set of types $\mathscr{T}$:

$$s \leq_{\mathscr{B}} t \qquad \Longleftrightarrow \qquad [\![s]\!]_{\mathscr{B}} \subseteq [\![t]\!]_{\mathscr{B}}$$

We can then pick any language that uses the types in $\mathscr{T}$ (and whose semantics conforms with the intuition underlying the model condition on function types), define its typing rules and use in the subsumption rule the subtyping relation $\leq_{\mathscr{B}}$ we have just defined. We write $\Gamma \vdash_{\mathscr{B}} e : t$ for the typing judgement of the language.

We have just defined a language $\mathscr{L}$ whose subtyping relation is defined set-theoretically. Of course for the time being this language is just a virtual one but let us use it to outline how to close circularity (in Section 3.3 we will show a concrete example of such a language).

2.7 Closing the circle

So, what are the relations between the bootstrap model and the obtained virtual language $\mathscr{L}$? And in particular, what is the relation between the bootstrap model and the values of this language? Have we lost all the intuition underlying the "types as sets of values" interpretation?

To answer all these questions, consider the abstract language $\mathscr{L}$ we obtained and in particular all its values, that is all its well-typed closed terms in normal form. This induces a new interpretation of the types $\mathscr{T}$. More precisely this induces the interpretation $(\mathscr{V}, [\![]\!]_{\mathscr{V}})$ where $\mathscr{V}$ is the set of all values of the obtained language and $[\![]\!]_{\mathscr{V}}$ is the mapping defined as $[\![t]\!]_{\mathscr{V}} = \{v \in \mathscr{V} \mid \vdash_{\mathscr{B}} v : t\}$.

If the typing relation for $\mathscr{L}$ is defined appropriately[5], it turns out that this interpretation satisfies the conditions of being a set-theoretic model, therefore we can use it to define a new subtyping relation on $\mathscr{T}$:

$$s \leq_{\mathscr{V}} t \qquad \Longleftrightarrow \qquad [\![s]\!]_{\mathscr{V}} \subseteq [\![t]\!]_{\mathscr{V}}$$

We could imagine to start again the process, that is to use this subtyping relation in the subsumption rule of our language, and use the resulting sets of values to define yet another subtyping relation and so on. But this is not necessary as the process has already converged. This is stated by one of the central results of our work:

$$s \leq_{\mathscr{B}} t \qquad \Longleftrightarrow \qquad s \leq_{\mathscr{V}} t$$

that is, the subtyping relation induced by the bootstrap model already defines the subtyping relation of the "types as sets of values" model of the resulting calculus. We have closed the circle we broke.

3. A taste of formalism

In the next three sections we introduce a concrete bootstrap model and a concrete programming language and use them to put our technique at work.

3.1 Universal model

We said that the model condition we imposed on the interpretation of arrow types relaxes the condition (1) that for cardinality reasons no model can satisfy. But have we relaxed it enough? In other words, does there exist a model, that is a pair $(\mathscr{D}, [\![]\!])$, that satisfies all the seven conditions? The answer is positive, as we can exhibit the following model.

Let $\mathscr{U}$ be the least solution of the equation $X = X^2 + \mathscr{P}_f(X^2)$, where $\mathscr{P}_f(X)$ denotes the finite powerset of X. Note that the elements of $\mathscr{U}$ are finitely generated by the following abstract syntax:

$$d \quad ::= \quad (d_1, d_2) \ \mid \ \{(d_1, d_1'), \ldots, (d_n, d_n')\} \qquad {\scriptstyle (n \geq 0)}$$

Then define the following interpretation function $[\![\_]\!]_{\mathscr{U}} : \mathscr{T} \to \mathscr{P}(\mathscr{U})$:

- $[\![\mathbf{0}]\!]_{\mathscr{U}} = \varnothing$;
- $[\![\mathbf{1}]\!]_{\mathscr{U}} = \mathscr{U}$;
- $[\![\neg t]\!]_{\mathscr{U}} = \mathscr{U} \setminus [\![t]\!]_{\mathscr{U}}$;
- $[\![s \vee t]\!]_{\mathscr{U}} = [\![s]\!]_{\mathscr{U}} \cup [\![t]\!]_{\mathscr{U}}$;
- $[\![s \wedge t]\!]_{\mathscr{U}} = [\![s]\!]_{\mathscr{U}} \cap [\![t]\!]_{\mathscr{U}}$;
- $[\![s \times t]\!]_{\mathscr{U}} = \{(d_1, d_2) \mid d_1 \in [\![s]\!]_{\mathscr{U}}, d_2 \in [\![t]\!]_{\mathscr{U}}\}$;
- $[\![t \to s]\!]_{\mathscr{U}} = \{\{(d_1, d_1'), \ldots, (d_n, d_n')\} \mid d_i \in [\![t]\!]_{\mathscr{U}} \Rightarrow d_i' \in [\![s]\!]_{\mathscr{U}}\}$.

Note that $[\![s \times t]\!]_{\mathscr{U}} = [\![s]\!]_{\mathscr{U}} \times [\![t]\!]_{\mathscr{U}}$, while for arrow types we have $[\![t \to s]\!]_{\mathscr{U}} = \mathscr{P}_f(\overline{[\![t]\!]_{\mathscr{U}} \times \overline{[\![s]\!]_{\mathscr{U}}}})$ (note the *finiteness* in $\mathscr{P}_f$).

It is quite easy to verify (by induction on the elements of $\mathscr{U}$) that these relations indeed define a (unique) function $\mathscr{T} \to \mathscr{P}(\mathscr{U})$, and that $(\mathscr{U}, [\![\_]\!]_{\mathscr{U}})$ satisfies the seven conditions that characterise a model (simply note that $\mathscr{P}_f(X) = \varnothing \iff \mathscr{P}(X) = \varnothing$).

So, now we know that a model exists. Actually in [11] it is shown that there exist several different models and that they induce different subtyping relations[6].

To define the type system, we need to choose a specific model. Any model would produce a sound type system and allow us to

[5] By "appropriately" we mean that it must comply with the intuitive behaviour that we had in mind when we introduced the extensional interpretation. In particular, it must be such that $\vdash v : t \Longleftrightarrow \not\vdash v : \neg t$.

[6] For instance, let us consider the recursive type $t = (\mathbf{0} \twoheadrightarrow \mathbf{1}) \wedge \neg(t \to \mathbf{0})$. In the model we have just built this type is empty, that is $t \leq_{\mathscr{U}} \mathbf{0}$, but it is possible to build a different model $\mathscr{D}$ such that $t \not\leq_{\mathscr{D}} \mathbf{0}$.

close the circle. The algorithm to compute the subtyping relation, however, does depend on the choice of the model. The model $\mathscr{U}$ is a natural choice because it is universal, in the sense that it induces the largest possible subtyping relation; formally, for every model $\mathscr{D}$ and types $t_1, t_2 \in \mathscr{T}$:

$$t_1 \leq_{\mathscr{D}} t_2 \quad \Rightarrow \quad t_1 \leq_{\mathscr{U}} t_2$$

Finally note that $\mathscr{U}$ is too poor a structure to be used as the target for interpreting any serious programming language, as it cannot express but finite functions. Nevertheless it is enough for interpreting types (actually, for characterising type containment).

3.2 Type representation and subtyping algorithm

Now that we have a semantically defined subtyping relation the time has come to think of an effective way to check whether two types are in the subtyping relation. In order to define the subtyping algorithms it results much easier to work with types that are written in a canonical form. Finding a canonical form is not very hard since we are helped in it by the semantic interpretation of types. So let us consider again our types that can be seen as the set of regular trees coinductively defined by:

$$t ::= \mathbf{0} \mid \mathbf{1} \mid t \to t \mid t \times t \mid \neg t \mid t \vee t \mid t \wedge t$$

Let call *atom* either an arrow type $t \to t$ or a product type $t \times t$. A type is in *disjunctive normal form* if and only if it is a finite union of finite intersections of atoms or their negations (in the latter case we speak of *negative atoms*). For instance:

$$(a_1 \wedge a_2 \wedge \neg a_3) \vee (a_4 \wedge \neg a_5) \vee (\neg a_6 \wedge \neg a_7) \vee (a_8 \wedge a_9)$$

where a_i'a are atoms (we identify $\mathbf{0}$ with the empty union and $\mathbf{1}$ with the empty intersection).

We say that two types s and t are *equivalent* if they have the same interpretation (that is $[\![s]\!] = [\![t]\!]$) and we denote it by $s \simeq t$. Every type of our system is equivalent to a type in disjunctive normal form. Therefore without loss of generality we can consider this form as the representation of types we work on.

We can further clean our representation by noting that if the inner intersections mix atoms with different constructors, then they degenerate either to $\mathbf{0}$, or to a unique type according to the polarities of the involved atoms[7]: so for instance $(s_1 \times t_1) \wedge (s_2 \to t_2) \simeq \mathbf{0}$ and $(s_1 \times t_1) \wedge \neg(s_2 \to t_2) \simeq s_1 \times t_1$. Therefore we only consider unions of intersections on atoms of the same sort (that is, that do not mix product and arrow types), e.g. :

$$\begin{array}{c} ((s_1 \times t_1) \wedge (s_2 \times t_2) \wedge \neg(s_3 \times t_3)) \\ \vee \\ (\neg(s_4 \to t_4) \wedge \neg(s_5 \to t_5)) \\ \vee \\ ((s_2 \times t_3) \wedge (s_4 \times t_1)) \end{array}$$

Finally we can organise the outer union into two packets by grouping together all the intersections on the same sorts (here above the first and third addenda):

$$\begin{array}{c} ((s_1 \times t_1) \wedge (s_2 \times t_2) \wedge \neg(s_3 \times t_3)) \\ \vee \\ ((s_2 \times t_3) \wedge (s_4 \times t_1)) \end{array} \quad \bigvee \quad (\neg(s_4 \to t_4) \wedge \neg(s_5 \to t_5))$$

Now note that every addendum can be represented by a pair (P, N) where P is the set of the positive atoms of the intersection and N the negatives ones. For instance the first addendum is the pair $(\{(s_1 \times t_1), (s_2 \times t_2)\}, \{(s_3 \times t_3)\})$. Therefore every packet can be represented by a set S of such pairs, under the following form:

$$\bigvee_{(P,N) \in S} \left((\bigwedge_{a \in P} a) \wedge (\bigwedge_{a \in N} \neg a) \right)$$

Thus two of such sets are all we need to represent every type. For instance our previous type is represented by the pair

$$\left(\left\{ \begin{array}{c} (\{(s_1 \times t_1), (s_2 \times t_2)\}, \{(s_3 \times t_3)\}) \\ (\{(s_2 \times t_3), (s_4 \times t_1)\}, \{\}) \end{array} \right\}, \{(\{\}, \{(s_4 \to t_4), (s_5 \to t_5)\})\} \right)$$

It is interesting to notice that this is not just a theoretical representation but it is also the representation we used in early versions of the ℂDuce interpreter (the current implementation is using more efficient partial decision trees).

But let us go back to our problem, which is the one of defining algorithms that verify whether two types s and t are in the subtyping relation. The key observation for what follows is again that the problem of deciding whether two types are in subtyping relation can be reduced to the problem of deciding whether a type is empty. Recall

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

Since $\varnothing = [\![\mathbf{0}]\!]$ then

$$s \leq t \iff s \wedge \neg t \simeq \mathbf{0}$$

Since every type can be represented as the union of addenda of uniform sort and a union is empty only if all its addenda are empty, then in order to decide the emptiness of every type it suffices to establish when the terms

$$A = (\bigwedge_{a \in P} a) \wedge (\bigwedge_{a \in N} \neg a)$$

are empty for P and N formed by types of the same sort (all products or all arrows). Or equivalently this results to deciding

$$(\bigwedge_{a \in P} a) \leq (\bigvee_{a \in N} a)$$

that is, we must be able to decide whether

$$(\bigwedge_{s \times t \in P} s \times t) \wedge (\bigwedge_{s \times t \in N} \neg(s \times t)) \tag{2}$$

and

$$(\bigwedge_{s \to t \in P} s \to t) \wedge (\bigwedge_{s \to t \in N} \neg(s \to t)) \tag{3}$$

are equivalent to $\mathbf{0}$. So the algorithm must decompose this problem into simpler subproblems, and this can be done by using some general algebraic rules combined with the properties of the semantic interpretation. In particular we have that the type in (2) is equivalent to $\mathbf{0}$ if and only if for every $N' \subseteq N$:

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

while the type in (3) is equal to zero if and only if there exists some $(t' \to s') \in N$ such that for every $P' \subseteq P$:

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

The proof of this second equivalence can be found in [11] (the first being a well known property of products of sets). For the time

[7] The case when an intersection mix only negative atoms needs a special treatment.

$$\frac{\Gamma \vdash e : s \leq_{\mathscr{B}} t}{\Gamma \vdash e : t}\ (\textit{subsum}) \qquad \frac{}{\Gamma \vdash x : \Gamma(x)}\ (\textit{var}) \qquad \frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1,e_2) : t_1 \times t_2}\ (\textit{pair}) \qquad \frac{\Gamma \vdash e : t_1 \times t_2}{\Gamma \vdash \pi_i(e) : t_i}\ (\textit{proj})$$

$$\frac{\Gamma \vdash e_1 : t_1 \to t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 e_2 : t_2}\ (\textit{appl}) \qquad \frac{\begin{array}{l} t \equiv (\bigwedge_{i=1..n} s_i \to t_i) \setminus (\bigvee_{j=1..m} s'_j \to t'_j) \not\leq \mathbf{0} \\ (\forall i)\ \Gamma, (f : t), (x : s_i) \vdash e : t_i \end{array}}{\Gamma \vdash \mu f^{(s_1 \to t_1; \ldots; s_n \to t_n)}(x).e : t}\ (\textit{abstr})$$

$$\begin{array}{l} \text{(for } s_1 \equiv s \wedge t,\ s_2 \equiv s \wedge \neg t) \\ \dfrac{\Gamma \vdash e : s \qquad (\forall i = 1,2).\ (s_i \simeq \mathbf{0} \vee \Gamma, (x : s_i) \vdash e_i : t')}{\Gamma \vdash (x = e \in t)\mathbf{?}e_1 \mathbf{:} e_2 : t'}\ (\textit{typecase}) \end{array}$$

Figure 2: Typing rules

being note that we have obtained a specification of the subtyping algorithm: to decide if $s \leq t$ we consider $s \wedge \neg t$, put it in disjunctive normal form with homogeneous atoms, and then coinductively saturate it (since we have recursive types) by decomposing it according to the rules above [8]. The termination of the algorithm is ensured by the regularity and contractiveness of our types, while soundness and completeness are given by the correctness of the transformations we did right above, and the universality of the model.

Of course this is just a high level specification of the algorithm. If we implemented it plainly we surely obtain a program with backtracking and an explosion of memory usage. The current implementation of subtyping in ℂDuce use a lightweight solver for monotonic boolean constraints to remove backtracking and extensively uses caching techniques and set-theoretic heuristics.

3.3 Language

So let us consider our type system whose types are those defined in Section 2.5, namely

$$t ::= \mathbf{0} \mid \mathbf{1} \mid t \to t \mid t \times t \mid \neg t \mid t \vee t \mid t \wedge t$$

and whose subtyping relation is the relation $\leq_{\mathscr{B}}$ induced by some (bootstrap) model $\mathscr{B}$ of the types above (in practice we will always choose the bootstrap model to be the universal model $\mathscr{U}$ of Section 3.1). We said that the conditions we chose account for languages with possibly non-terminating and overloaded functions. Therefore, we apply this type system to a simply typed lambda calculus with recursive and overloaded functions.

The syntax of *expressions* of the language is as follows:

$$\begin{array}{rlll} e & ::= & x & \text{variable} \\ & \mid & \mu f^{(s_1 \to t_1; \ldots; s_n \to t_n)}(x).e & \text{abstraction, } n \geq 1 \\ & \mid & e_1 e_2 & \text{application} \\ & \mid & (e_1,e_2) & \text{pair} \\ & \mid & \pi_i(e) & \text{projection, } i = 1,2 \\ & \mid & (x = e \in t)\mathbf{?}e_1 \mathbf{:} e_2 & \text{binding type case} \end{array}$$

The language is simple. The core is a lambda-calculus with pairs and projections. Functions are of the form $\mu f^{(s_1 \to t_1; \ldots; s_n \to t_n)}(x).e$. We used the binder μ to denote the fact that the definition may be recursive, that is that f may occur free in the body e of the function[9]. The other point to remark is that the function specifies a set of arrow types. When the specified arrows are at least two, it denotes the overloaded nature of the function. In the definition $\mu f^{(s_1 \to t_1; \ldots; s_n \to t_n)}(x).e$ the specified type indicates that when this function is applied to an argument of type s_i then it returns a result of type t_i. Therefore the set of arrows indicates that we have to check that e has type t_i under the hypothesis that x has type s_i.

Note that alone this does not suffice to have *real* overloading, that is the execution of different code for different input types: we can use the option of specifying several arrows just to give a more precise typing. For instance, if we had in our calculus integers, even and odd types, we could write

$$\mu \mathsf{s}^{(\mathsf{Even} \to \mathsf{Odd}; \mathsf{Odd} \to \mathsf{Even})}(x).x + 1$$

instead of the less precise $\mu \mathsf{s}^{(\mathsf{Int} \to \mathsf{Int})}(x).x + 1$. But even though two arrow types are specified in the definition of s, this function is by no means overloaded.

So in order to endow our language with *real* overloading we also add a type case expression

$$(x = e \in t)\mathbf{?}e_1 \mathbf{:} e_2$$

that binds e to x and executes e_1 or e_2 according to whether e is of type t or not.[10]

Because of this expression, the semantics of the language is defined in terms of the typing of the terms (or of the values at least). Therefore before giving the reduction semantics we have to define the type system. We have the types, we have the subtyping relation, it remains to define which terms have which types. This is done by the deduction system in Figure 2.

The first five rules are completely standard and do not deserve any particular comment. Just note that the subsumption rule uses the bootstrap subtyping relation $\leq_{\mathscr{B}}$.

Let us explain the rule for typing abstractions. As a first approximation we can consider the case for $m = 0$, that is, when the type t of the function is the intersection of all the types specified in its interface: $\bigwedge_{i=1..n} s_i \to t_i$. The rule thus verifies that the function has indeed all these types, that is, for every input type $s_i \to t_i$, it checks that the body e has type t_i under the assumption that the parameter x has type s_i. Since the function may be recursively defined, it is also assumed that the function identifier f has type t, which is the type given to the abstraction itself. The rule actually is (and must be) more general since it allows us to take for t a type strictly smaller than the intersection of the types in the interface: indeed, it is possible to remove any finite number of arrow types from the in-

[8] In the absence of recursive types we use the transformation rules above to inductively decompose the problem.

[9] In the untyped λ-calculus this definition corresponds to the term $\mathbf{Y}(\lambda f.\lambda x.e)$ were $\mathbf{Y}$ is a fixpoint combinator. However in our typed framework we cannot see recursion as a special case of application of a combinator as we need to determine the recursion variable in order to precisely type the body of the function and, more precisely, to repeat it in case of overloading.

[10] Note that binding the variable in the type case provides a more precise typing, since it allows us to type e_1 and e_2 under different hypothesis for x (see the *typecase* typing rule).

tersection, provided that t remains non-empty. The technical reason for this will be explained later on.

The rule for the type case is also quite complex. It first deduces the type s of the checked argument, then it separately checks the two branches under the hypothesis that e (thus x) is in t (and thus in $s \wedge t$) or not (that is in $s \wedge \neg t$). The rule discards the branches *that cannot be selected* (which is safely approximated by the fact that the corresponding s_i is empty). The reader may wonder why we do not return a type error when one of the two branches cannot be selected. As a matter of fact this is a key feature for typing overloaded functions, where the body is repeatedly checked under different hypothesis for some of which the s_i of some typecase may be empty.[11] This simple function should clarify the point:

$$\mu \mathsf{f}^{(\mathsf{Int}\to\mathsf{Int};\mathsf{Bool}\to\mathsf{Bool})}(x).(y = x \in \mathsf{Int})\,\mathbf{?}\,(y+1) : \mathsf{not}(y)$$

when we type the body under the hypothesis $x : \mathsf{Int}$, then the second branch cannot be selected while under $x : \mathsf{Bool}$ is the first one that cannot be selected.

We can now give the reduction rules:

$$\begin{array}{rcll}
(\mu f^{(\dots)}(x).e)v & \to & e[x/v, (\mu f^{(\dots)}(x).e)/f] & \\
(x = v \in t)\,\mathbf{?}\,e_1 : e_2 & \to & e_1[x/v] & \text{if } v \in [\![t]\!] \\
(x = v \in t)\,\mathbf{?}\,e_1 : e_2 & \to & e_2[x/v] & \text{if } v \notin [\![t]\!] \\
\pi_i(v_1, v_2) & \to & v_i &
\end{array}$$

the first rule is plain β-reduction in the presence of recursion: the actual parameter is substituted for the formal one, and the whole function is replaced for the recursion variable in the body. Context rules, that we omit, implement standard call-by-value, left-most reduction, and v ranges over *values* that is closed normal forms; they are well-typed closed terms generated by the following production:

$$v ::= \mu f^{(\dots)}(x).e \mid (v,v)$$

At this point, we have a language, a type system, and a small-step operational semantics. It is then possible to prove type soundness by classical syntactical methods (subject reduction theorem, progress lemma). The set-theoretic definition of subtyping helps to prove intermediate lemmas, such as elimination of subsumption or admissibility of the conjunction rule (namely, $\Gamma \vdash e : t_1 \wedge t_2 \iff \Gamma \vdash e : t_1 \wedge \Gamma \vdash e : t_2$). With a syntactic presentation of the subtyping relation, we would need to prove a lot of tedious properties on the subtyping relation (monotonicity, Boolean laws,...), that in the semantic approach follow straightforwardly from the set-theoretic definition. Note that even if the subtyping relation is defined semantically, we do not have to deal with complex denotational models of the language.

More important for the goals of this paper is the question whether this language closes the circle as we anticipated in Section 2.7. To answer this question, consider the language we obtained and in particular all its values. This induces the interpretation $(\mathscr{V}, [\![]\!]_{\mathscr{V}})$ where $\mathscr{V}$ is the set of values we defined in the production right above and $[\![]\!]_{\mathscr{V}}$ is the mapping defined as $[\![t]\!]_{\mathscr{V}} = \{v \in \mathscr{V} \mid \vdash v : t\}$.

It turns out that this interpretation satisfies the conditions of being a set-theoretic model, therefore we can use it to define a new subtyping relation on $\mathscr{T}$:

$$s \leq_{\mathscr{V}} t \iff [\![s]\!]_{\mathscr{V}} \subseteq [\![t]\!]_{\mathscr{V}}$$

and this relation satisfies the property

$$s \leq_{\mathscr{B}} t \iff s \leq_{\mathscr{V}} t \qquad (4)$$

that is, the subtyping relation induced by the bootstrap model already defines the subtyping relation of the "types as sets of values" model of the resulting calculus. We have closed the circle we broke.

What is the intuition behind this technical result, and why does it hold? Equation (4) tells us that the language we defined is rich enough, since there always exists a value to separate two distinct types. In other terms, it tells us that the set of values of our language is a model of types with "enough points". This can be understood by noting that the ⇒ direction of (4) is straightforward (as it comes from subsumption), so what (4) states is that—whatever bootstrap model we started from—whenever $s \not\leq_{\mathscr{B}} t$, then we can exhibit a value v of our language such that $\vdash v : s$ and $\not\vdash v : t$. In particular, the peculiar feature of our language that makes its set of values rich enough, is that functional values can specify several arrow types. The multiplicity of arrow types allows us to exhibit values whose minimum type is an intersection of arrow types (that is, to define overloaded functions values). So, in particular, in the general case of arrow subtyping, for instance $\bigwedge_{i=1..n} s_i \to t_i \not\leq \bigvee_{j=1..m} s'_j \to t'_j$, it is easy to exhibit a value that is in the former type but not in the latter, since any function of the form $\mu f^{(s_1 \to t_1;\dots;s_n \to t_n)}(x).e$ can be used. Indeed, the rule $(abstr)$ assigns to this value the type $(\bigwedge_{i=1..n} s_i \to t_i) \setminus (\bigvee_{j=1..m} s'_j \to t'_j)$. This explains why we need to allow these negation of arrow types in the rule $(abstr)$, and it is interesting to notice that this extra flexibility does not jeopardise type soundness in the system.

To put it differently, equation (4) says that the subsumption rule

$$t \leq_{\mathscr{B}} s \Longrightarrow (\forall e.\ \Gamma \vdash e : t \ \Rightarrow\ \Gamma \vdash e : s)$$

is actually an equivalence:

$$t \leq_{\mathscr{B}} s \iff (\forall e.\ \Gamma \vdash e : t \ \Rightarrow\ \Gamma \vdash e : s)$$

and, indeed, we can restrict our attention to values:

$$t \leq_{\mathscr{B}} s \iff (\forall v.\ \vdash v : t \ \Rightarrow\ \vdash v : s)$$

And in particular ⟸ states that

$$t \not\leq_{\mathscr{B}} s \Longrightarrow (\exists v.\ \vdash v : t \text{ and } \not\vdash v : s)$$

The presence of enough values to separate all distinct types (as the s and t above) makes the language "appropriate"—as we informally said in Section 2.7, cf. Footnote 5—to our development. Therefore it is not by chance that (4) holds, but it is due to the fact that the language we defined precisely matches the intuition we followed in defining the semantics of types.

Another important result relates the subtyping relation and the operational semantics. Consider two types t_1 and t_2 such that $t_1 \leq \mathbf{0} \to \mathbf{1}$ (that is, all the values in t_1 are function abstractions). Then the equation $t_1 \leq t_2 \to t$ has a smallest solution t, and this solution is also characterised (up to equivalence) by the semantic property: $[\![t]\!]_{\mathscr{V}} = \{v \mid v_1 v_2 \xrightarrow{*} v, \vdash v_1 : t_1, \vdash v_2 : t_2\}$.

3.4 Semantic subtyping at work

Now that we have established our basic system and studied its properties, we can consider to add new type constructors to it. In order to do that we will proceed semantically, of course. This means that for every new type constructor we will add a constraint to the model definition which intuitively will force a set-theoretic interpretation of the new types. In other terms we use the function $\mathscr{E}[\![\;]\!]$ to associate to each type constructor the set-theoretic intuition we have of it and then, somewhat mechanically, we will derive the subtyping rules for it.

[11] Actually, this is far from being a minor point: not considering the return type of unused branches is the main difference between dynamic overloading and type-case (or, equivalently, the `dynamic` types of [1]). The latter always returns the union of the result types of all the branches and, as such, it is not able to discriminate different input types.

Reference types.

The first type constructor we add to our types is the constructor for reference types, $\mathtt{ref}\,t$. To add it to our semantic subtyping framework we have thus to establish what is a set-theoretic interpretation for $\mathtt{ref}\,t$. Let us proceed extensionally as we did for function spaces. Intuitively we want to interpret $\mathtt{ref}\,t$ as the set of *values* of type $\mathtt{ref}\,t$. A value of type $\mathtt{ref}\,t$ is a container, a box, that can contain all the values of type t. It can then thus be identified with the set of all values it can contain. In other terms we want to interpret a value of type $\mathtt{ref}\,t$ as the set $\{\mathtt{ref}\,v \mid v \in [\![t]\!]\}$ which is of course isomorphic to $[\![t]\!]$. But then instead speaking of *a* value of type $\mathtt{ref}\,t$ we should rather speak of *the* value of type $\mathtt{ref}\,t$: since all the "boxes" in $\mathtt{ref}\,t$ can contain exactly the same values, then they are identified to the set above and, intensionally, they are all the same. Thus the interpretation of $\mathtt{ref}\,t$ must be a singleton containing the unique value of type $\mathtt{ref}\,t$ which in its turn is univocally identified by $[\![t]\!]$. This of course is true only if $[\![t]\!] \neq \varnothing$, because otherwise $\mathtt{ref}\,t$ has an empty interpretation as well (because to create the reference, we need an initial value of type t). This explains why the condition we would like to impose for reference types is the following one:

$$[\![\mathtt{ref}\,t]\!] = \begin{cases} \{[\![t]\!]\} & \text{if } [\![t]\!] \neq \varnothing \\ \varnothing & \text{otherwise} \end{cases} \tag{5}$$

Of course such a condition has the same set-theoretic problems as the condition (1) we initially established for arrow types, but before handling it let us introduce a second type constructor:

Lazy evaluation.

The second type constructor we add to our language is the constructor for lazy expressions, $\mathtt{lazy}\,t$. Intuitively a value of $\mathtt{lazy}\,t$ is a blocked computation that when unblocked returns a value of type t, so in a programming language a value of type $\mathtt{lazy}\,t$ is an expression of the form $\mathtt{lazy}\,e$ where e is a closed *expression* (i.e. possibly not in normal form) of type t. Our motivation for introducing lazy types in ℂDuce is to model streaming processing of XML. Intuitively, one might imagine to define the type of infinite streams of integers as the recursive type $t = \mathsf{Int} \times t$. Unfortunately $t \simeq \mathbf{0}$, therefore adding such streams to the language would inhabit the empty type, making the whole type system collapse. The solution is the addition of lazy types. The idea is that if we want to have, say, a stream of integers, we can type it by the following recursive type: $s = \mathsf{Int} \times \mathtt{lazy}\,s$. So for instance $(\mu f^{(\mathsf{Int}\to s)}(x).(x,\mathtt{lazy}\,(f(x+1))))0$ is the stream of all natural numbers. Note that the same solution without the "blocking" lazy constructor yields to the definition of a diverging expression: $(\mu f^{(\mathsf{Int}\to t)}(x).(x, f(x+1)))0)$, as it is natural since every value it returned would inhabit the empty type (recall that $t = \mathsf{Int} \times t$ is empty).

To give a set-theoretic intuition of lazy types is quite easy: $\mathtt{lazy}\,t$ must be interpreted as the set of all values of that type, that is the set $\{\mathtt{lazy}\,e \mid e \text{ is closed and } e : t\}$. But each $\mathtt{lazy}\,e$ is identified by all possible results it can return, namely $\{v \mid e \to^* v\}$. This is a subset of $[\![t]\!]$, from which we deduce the following condition for lazy types:

$$[\![\mathtt{lazy}\,t]\!] = \mathscr{P}([\![t]\!]) \tag{6}$$

Note that both conditions (5) and (6) induce the same set-theoretic problems as the equation (1) for functional types. So the solution is the same as there: we do not want equality to hold rather the subtyping relation to be defined *as if*. Therefore instead of adding some new condition 7, it simply suffices to extend the definition of $\mathscr{E}[\![\,]\!]$ for the new type constructor as follows:

$$\begin{array}{rcl} \mathscr{E}[\![\mathtt{ref}\,t]\!] & = & \begin{cases} \{[\![t]\!]\} & \text{if } [\![t]\!] \neq \varnothing \\ \varnothing & \text{otherwise} \end{cases} \\ \mathscr{E}[\![\mathtt{lazy}\,t]\!] & = & \mathscr{P}([\![t]\!]) \end{array}$$

and the condition 7 originally introduced for function spaces makes everything work with the new type constructors as well.

Finally it remains to extend the definition of the subtyping algorithm for the new type constructors, which come to establish when

$$(\bigwedge_{a\in P} a) \leq (\bigvee_{a\in N} a)$$

holds, when the a's are all ref types or all lazy types. This is quite easy for ref types:

$$(\bigwedge_{\mathtt{ref}\,s\in P} \mathtt{ref}\,s) \leq (\bigvee_{\mathtt{ref}\,t\in N} \mathtt{ref}\,t) \iff \begin{array}{l} \exists \mathtt{ref}\,s \in P,\ s \simeq 0, \text{or} \\ \exists \mathtt{ref}\,s_1, \mathtt{ref}\,s_2 \in P, s_1 \not\simeq s_2, \text{or} \\ \exists \mathtt{ref}\,s \in P,\ \exists \mathtt{ref}\,t \in N,\ s \simeq t \end{array}$$

Note that in the case of P and N being singletons we recover the classic invariant subtyping relation for references types (enhanced with the case to deal with $\mathtt{ref}\,\mathbf{0}$).

> EXCURSUS. Daniele Varacca noticed that if we allow recursion inside reference types, then there does not exist any model. To see why, consider the following recursive type suggested by Daniele:
>
> $$t = \mathsf{Int} \vee (\mathtt{ref}\,t \wedge \mathtt{ref}\,\mathsf{Int})$$
>
> If we had a model, then either $t = \mathsf{Int}$ or $t \neq \mathsf{Int}$ hold. Does $t = \mathsf{Int}$? Suppose it does, then $\mathtt{ref}\,t \wedge \mathtt{ref}\,\mathsf{Int} = \mathtt{ref}\,\mathsf{Int}$ and $\mathsf{Int} = t = \mathsf{Int} \vee \mathtt{ref}\,\mathsf{Int}$, which is not true since $\mathtt{ref}\,\mathsf{Int}$ is not contained in Int. Therefore it must be $t \neq \mathsf{Int}$. According to our semantics this implies $\mathtt{ref}\,t \wedge \mathtt{ref}\,\mathsf{Int} = \mathbf{0}$, because they are interpreted as two distinct singletons. Thus $t = \mathsf{Int} \vee \mathbf{0} = \mathsf{Int}$, contradiction. The solution is to avoid recursion inside reference types, for instance by requiring that on every infinite branch of a regular type there are only finitely many occurrences of the ref type constructor. It is then possible to define a model, but this is a completely new work: see [8].

For lazy types, instead, we expect to find a covariant subtyping relation. By performing some set-theoretic computation we arrive to determine that $(\bigwedge_{\mathtt{lazy}\,s\in P} \mathtt{lazy}\,s) \wedge (\bigwedge_{\mathtt{lazy}\,t\in N} \neg\mathtt{lazy}\,t)$ is equivalent to $\mathbf{0}$ if and only if there exists $\mathtt{lazy}\,t \in N$ such that for every $P' \subseteq P$:

$$\left(\bigwedge_{\mathtt{lazy}\,s\in P'} s \wedge \neg t\right) \simeq \mathbf{0}$$

or equivalently we have that $(\bigwedge_{\mathtt{lazy}\,s\in P} \mathtt{lazy}\,s) \leq (\bigvee_{\mathtt{lazy}\,t\in N} \mathtt{lazy}\,t)$ if and only if there exists $\mathtt{lazy}\,t \in N$ such that for every $P' \subseteq P$:

$$(\bigwedge_{\mathtt{lazy}\,s\in P'} s) \leq t$$

from which the covariance of the lazy types appears in a clearer way.

Note that this is a special case of function types where the domain is $\mathbf{1}$. Indeed the values of the lazy types can be assimilated to functions that wait for any argument to carry on the rest of the computation, that is $\mathtt{lazy}\,t = \mathbf{1} \to t$. However while this holds in the simplified setting we presented here, this is no longer true in the more complex framework used in [12] where a different model condition is used to account for the presence of type errors.

Back to product types.

In the definition we give for set theoretic models earlier in this section we stated for product types that the following equation had to hold

$$\llbracket t_1 \times t_2 \rrbracket = \llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket \tag{7}$$

and we let understand that such a condition implied some closure properties on the domain $\mathscr{D}$ in particular it had to contain a subset isomorphic to its square type: $\mathscr{D} \simeq \ldots + \mathscr{D}^2$. We did not dwell on this point as it is easy to find domains that are closed for finite products. However even if the condition (7) can be easily satisfied it is nevertheless constraining, as it states how the product types are to be interpreted. By now we know that while for boolean type constructors $(\wedge, \vee, \neg)$ it is very important to fix their interpretation, all that matters for atom type constructors is rather the relation induced by their interpretation. For all the other atomic type constructors $(\rightarrow, \mathtt{ref}, \mathtt{lazy})$ we just imposed that they must be interpreted so that the induced relation were as if they were set-theoretic. We did it by resorting to an extensional interpretation of types. Product types are no exception. Therefore rather than assuming some closure properties of the target $\mathscr{D}$ we replace the condition (7) by an equivalent condition on $\mathscr{E}\llbracket \_ \rrbracket$.

To summarise consider the type system obtained by allowing arbitrary boolean combinations of types with the type constructors $\rightarrow, \times, \mathtt{ref}, \mathtt{lazy}$:

$$t ::= \mathbf{0} \mid \mathbf{1} \mid t \rightarrow t \mid t \times t \mid \mathtt{lazy}\, t \mid \mathtt{ref}\, t \mid \neg t \mid t \vee t \mid t \wedge t$$

Let $\llbracket \_ \rrbracket$ be an interpretation function of the types above in some set $\mathscr{D}$. We say that $\llbracket \_ \rrbracket$ is *set-theoretic* if and only if (i) it interprets boolean constructors as the corresponding set-theoretic operators on $\mathscr{D}$ and (ii) the counterimage of the empty set is the same as for its associated extensional interpretation:

1. $\llbracket \mathbf{0} \rrbracket = \varnothing$;
2. $\llbracket \mathbf{1} \rrbracket = \mathscr{D}$;
3. $\llbracket t_1 \vee t_2 \rrbracket = \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket$;
4. $\llbracket t_1 \wedge t_2 \rrbracket = \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket$;
5. $\llbracket \neg t \rrbracket = \mathscr{D} \backslash \llbracket t \rrbracket$;
6. $\llbracket t \rrbracket = \varnothing \iff \mathscr{E}\llbracket t \rrbracket = \varnothing$

where the extensional interpretation associated to $\llbracket\ \rrbracket$ is defined as:

$$\begin{array}{lcl}
\mathscr{E}\llbracket \mathbf{0} \rrbracket & = & \varnothing \\
\mathscr{E}\llbracket \mathbf{1} \rrbracket & = & \mathscr{P}(\mathscr{D}^2) + \mathscr{D}^2 + \mathscr{P}(\mathscr{D}) + \mathscr{P}(\mathscr{P}(\mathscr{D})) \\
\mathscr{E}\llbracket t_1 \vee t_2 \rrbracket & = & \mathscr{E}\llbracket t_1 \rrbracket \cup \mathscr{E}\llbracket t_2 \rrbracket \\
\mathscr{E}\llbracket t_1 \wedge t_2 \rrbracket & = & \mathscr{E}\llbracket t_1 \rrbracket \cap \mathscr{E}\llbracket t_2 \rrbracket \\
\mathscr{E}\llbracket \neg t \rrbracket & = & \mathscr{E}\llbracket \mathbf{1} \rrbracket \backslash \mathscr{E}\llbracket t \rrbracket \\
 & & \\
\mathscr{E}\llbracket t \rightarrow s \rrbracket & = & \mathscr{P}(\overline{\llbracket t \rrbracket \times \overline{\llbracket s \rrbracket}}) \\
\mathscr{E}\llbracket t \times s \rrbracket & = & \llbracket t \rrbracket \times \llbracket s \rrbracket \\
\mathscr{E}\llbracket \mathtt{lazy}\, t \rrbracket & = & \mathscr{P}(\llbracket t \rrbracket) \\
\mathscr{E}\llbracket \mathtt{ref}\, t \rrbracket & = & \begin{cases} \{\llbracket t \rrbracket\} & \text{if } \llbracket t \rrbracket \neq \varnothing \\ \varnothing & \text{otherwise} \end{cases}
\end{array}$$

Note that the conditions for being a set-theoretic model are no longer seven but just six (since products are now handled as the other type constructors) and that the extensional interpretation is defined on $\mathscr{E}\llbracket \_ \rrbracket : \mathscr{T} \rightarrow \mathscr{P}(\,\mathscr{P}(\mathscr{D}^2) + \mathscr{D}^2 + \mathscr{P}(\mathscr{D}) + \mathscr{P}(\mathscr{P}(\mathscr{D}))\,)$ since we have added $\mathscr{P}(\mathscr{D})$ and $\mathscr{P}(\mathscr{P}(\mathscr{D}))$ to extensionally interpret `lazy` and `ref` types, respectively.

And that's all. Now it remains to find a model and define the corresponding subtyping relation (which is far from being straightforward: see the excursus earlier in this section).

So the moral of our approach is, be strict in the interpretation of boolean combinators (define precisely how they must be interpreted), and loose in the interpretation of type constructors (state only how containment should look like, extensionally).

Other paradigms.

Finally, we want to stress that although here we applied the semantic subtyping approach to add boolean combinators to a simply typed λ-calculus, our technique is general and applies to other paradigms, as well. For instance, in [8] our technique is applied to define the $\mathbb{C}\pi$-calculus, a π-calculus where boolean combinators are added to the type constructors $\mathtt{ch}^+(t)$ and $\mathtt{ch}^-(t)$; these classify all the channels on which it is possible to read or, respectively, to write a value of type t.

The use of boolean combinations makes it possible to get rid of the invariant channel type constructor since this can be defined as the intersection of the other two: $\mathtt{ch}(t) \overset{\text{def}}{=} \mathtt{ch}^+(t) \wedge \mathtt{ch}^-(t)$. The definition of the extensional interpretation is still needed for cardinality reasons, however bootstrapping here has a different flavour, as it generates a model that is much closer to the model of values. Interestingly, this model is defined by a fixpoint construction.

Another interesting point is that every channel *value* in $\mathbb{C}\pi$ has a minimum type (which has the form $\mathtt{ch}(t)$ for some t). A consequence of this is that the typing rule for channels does not require to include arbitrary differences with other types, as it is the case here with the (*abstr*) typing rule of this work.

In $\mathbb{C}\pi$ one finds the same paradox as the one with recursive reference types in the previous excursus (this is not surprising since a reference closely resembles to a read/write channel), but this problem can be get rid of by restricting to a "local" variant of the calculus which forbids to read from a channel received in input. This is a standard technique in π-calculus and can be implemented in $\mathbb{C}\pi$ by deleting the covariant $\mathtt{ch}^+(\,)$ type constructor. But this also makes the invariant constructor $\mathtt{ch}(\,)$ be no longer definable, which in turn makes the minimum typing property for channel values fail. In this case in order to still have a model of values, one is obliged to introduce in the typing rule for channels arbitrary (non-empty) differences of other channel types, exactly as in the (*abstr*) rule presented here. This is not the only striking resemblance between $\mathbb{C}$Duce and $\mathbb{C}\pi$: for instance it seems worth mentioning that in order to decide the subtyping relation for the $\mathbb{C}\pi$, one tackles the same difficulties as in deciding general subtyping for the polymorphic version of $\mathbb{C}$Duce [15], namely, one must be able to decide whether a type is a singleton or not. A more in-depth study of the relation between $\mathbb{C}$Duce and $\mathbb{C}\pi$ is under way.

4. Conclusion

We have informally explained the intuition and the basic techniques to develop a type system based on a semantically (set-theoretically) defined subtyping relation. As we anticipated in the introduction, our approach is quite generic: we mean by that that while not all the components we introduced may be needed when considering a particular language (universality is not an issue if recursive types are absent, bootstrap is useless in the presence of name-based subtyping, and so on), they nevertheless provide a bunch of recipes that can be used to test the feasibility of a semantic definition of subtyping.

We hope to have provided the reader with enough information so that she/he now feels familiar enough with the "semantic subtyping" setting to dig in the technical details of [12, 11, 13], and try to follow these guidelines to apply, if possible, the approach to the design of her/his favourite language.

Acknowledgements.

This article stems out from many interesting discussions we had with other researchers. We are grateful to all of them and in particular to Mariangiola Dezani and Daniele Varacca for their many suggestions and conspicuous feedback on the article. We want to thank Amy Felty, Catuscia Palamidessi, and the program committees of PPDP 2005 and ICALP 2005 for offering us this rare opportunity to present our work. This work was partially funded by the ACI Data-masses project "XML Transformation Languages: Logic and Applications" (TraLaLA).

5. References

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[2] 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.

[3] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4), September 1993.

[4] A. Asperti and G. Longo. *Categories, Types and Structures: An Introduction to Category Theory for the Working Computer Scientist*. MIT-Press, 1991.

[5] 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.

[6] 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.

[7] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, 1995.

[8] 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.

[9] M. Coppo and M. Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. *Notre-Dame Journal of Formal Logic*, 21(4):685–693, October 1980.

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

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

[12] 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.

[13] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic Subtyping. Extended version of [12], in preparation., 2005.

[14] R. Hindley and G. Longo. Lambda-calculus models and extensionality. *Zeit. Math. Logik Grund. Math.*, 26(2):289–319, 1980.

[15] 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.

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

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

[18] 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.

[19] John C. Reynolds. The coherence of languages with intersection types. In Takayasu Ito and Albert R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 675–700, Berlin, 1991. Springer-Verlag.

[20] John C. Reynolds. Design of the programming language Forsythe. Technical Report CMU-CS96 -146, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1996.