Semantic Subtyping

Alain Frisch
Département d'Informatique
École Normale Supérieure, Paris, France
Alain.Frisch@ens.fr

Giuseppe Castagna
C.N.R.S., *Département d'Informatique*
École Normale Supérieure, Paris, France
Giuseppe.Castagna@ens.fr

Véronique Benzaken
LRI, UMR 8623, C.N.R.S.
Université Paris-Sud, Orsay, France
Veronique.Benzaken@lri.fr

Abstract

Usually subtyping relations are defined either syntactically by a formal system or semantically by an interpretation of types in an untyped denotational model. In this work we show how to define a subtyping relation semantically, for a language whose operational *semantics is* driven by types*; we consider a rich type algebra, with product, arrow, recursive, intersection, union and complement types. Our approach is to "bootstrap" the subtyping relation through a notion of set-theoretic model* of the type algebra.

The advantages of the semantic approach are manifold. Foremost we get "for free" many properties (e.g., the transitivity of subtyping) that, with axiomatized subtyping, would require tedious and error prone proofs. Equally important is that the semantic approach allows one to derive *complete algorithms for the subtyping relation or the propagation of types through patterns. As the subtyping relation has a natural (inasmuch as semantic) interpretation, the type system can give informative error messages when static type-checking fails. Last but not least the approach has an immediate impact in the definition* and the implementation *of languages manipulating XML documents, as this was our original motivation.*

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 most widespread) consists in defining the subtyping relation by axiomatizing it in a formal system (a set of inductive or coinductive rules); in the semantic approach (for instance, [1, 3]), 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 is much more technical and constraining, and this may explain why it has obtained less attention than syntactic subtyping. However when used 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 can be used 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, $(t_1 \vee t_2) \to s \simeq (t_1 \to s) \wedge (t_2 \to s)$.
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". Furthermore, as most of the formal effort is done with the semantic definition of subtyping, studying variations of the algorithm (optimizations, different rules) results much simpler (this is common practice in database theory where, for example, optimizations 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, which 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 interpretation of negative types. For these reasons all the semantic approaches to subtyping previous to our work presented some limitation: no higher-order functions, no complement types, and so on. The main contribution of our work is the development of a formal framework that overtakes these limitations.

The starting point of this work is XDuce [7, 5, 6] a language for defining transformations of XML documents. XDuce is a functional language with an elegant definition and it provides many features and solutions that are interesting both from theoretical and practical viewpoints. Its type system relies on the observation that the types of XML documents can be represented as regular tree expressions where the typing relation coincide with (regular) language membership.

XDuce fits well the semantic approach to subtyping: as functions are not first class, values are just XML documents, and subtyping is exactly regular tree language inclusion. In this work we present the functional core of our language ℂDuce (read "seduce") which extends XDuce with first class functions, and makes boolean connectives (union, intersection, complement) explicit in the type algebra. We also recast XDuce features in a less XML-specific framework. This yields the first, in our ken, subtyping system with recursive types and arbitrary boolean combinations (interpreted as their set-theoretic counterpart).

In XDuce a programmer can easily write overloaded functions, but the type system is not powerful enough to type them: as a matter of fact XDuce functions can be defined in terms of a pattern matching that is very close to type case. It seems an undue restriction not to allow different matching branches to return different types, so to have overloaded functions. Therefore ℂDuce extends XDuce also for it allows the definition of (late bound) overloaded functions (see [2]).

Since in ℂDuce one can express overloaded functions, and patterns (Section 3) can discriminate on types, then we are in the presence of a type-driven operational semantics. In this paper, we show how to apply the semantic subtyping in such a setting: as it is not possible to consider a model of the language before the type system is defined, we introduce a notion of model *of the type algebra* (Section 2), start with a non-computational *bootstrap* model, use it to define first the subtyping relation, then the typing one, and finally the (operational) semantics of the language. One of the main results of this paper (Theorem 4.1) is that the typing relation endows the set of (well-typed) values of the language with a structure of model equivalent to the bootstrap model.

Finally, while the negation (complement) connective raises many difficulties in denotational models (e.g., the complement of an ideal is not an ideal) we avoid them by resorting to a purely set-theoretic notion of model.

2. Types

2.1. Type syntax

We want to define the syntax of a type system that includes recursive, product, arrow, intersection, union, and negation types but excludes ill-formed (i.e., unguarded) recursions such as $\mu\alpha.\alpha \vee \alpha$. This can be obtained by the following BNF productions:

$$\begin{array}{lrcl}
\textit{Types} & T & ::= & \alpha \mid C \mid \mu\alpha.C \\
\textit{Combinations} & C & ::= & A \mid \neg C \mid C \vee C \mid C \wedge C \\
\textit{Atoms} & A & ::= & T \rightarrow T \mid T \times T \mid b \mid \mathbf{0} \mid \mathbf{1}
\end{array}$$

where α denotes type variables and b ranges over a set of (atomic) basic types $\mathbb{B}$. The syntax above corresponds to the types we have in mind for our language but is unfit to the technical developments that follow: if we used such a presentation, then we would be obliged to introduce some syntactic congruence over types allowing us to deduce $\mu\alpha.C \equiv C[\mu\alpha.C/\alpha]$ (in order to account for recursive types in definitions and algorithms) and, say, $C \equiv C \vee C$ or $\mathbf{1} \equiv \neg\mathbf{0}$ (in order that algorithms, which proceed by saturating some sets of types, terminate), etc. For this reason, and because the algorithms we define next work "sort-wise" (arrows with arrows, products with products, and basic types with basic types), we rather use a (multi-sorted) algebraic presentation of syntactic types, which is equivalent to the one above but tailored for the technical developments that follow. The advantage of the algebraic presentation is that it hugely simplifies the statement of precise theorems and it is close to implementation; its drawback is that it results quite technical and tedious. So a reader mainly interested in language features can just skip to Section 2.2 and use, instead, the BNF presentation above (modulo the implicit semantic equivalences for boolean connectives and recursive types).

Shallow type expressions. We define the set of types as a fixpoint of a functor that constructs boolean combinations of types over a set. More precisely, we consider $\mathcal{T}X$, the set of *shallow* type expressions over X, whose elements are, intuitively, boolean combination of atoms of the form $x \rightarrow y$, $x \times y$, or b, with $x, y \in X$. Without loss of generality, we consider boolean combinations that are sorted—they separate, say, arrows from products—and in disjunctive normal form—i.e., finite unions of finite intersections of atoms and negation of atoms—: it turns out that these choices simplify the algorithms. For a set A of atoms, we can represent disjunctive normal forms on A as elements of $\mathcal{P}_f(\mathcal{P}_f(A) \times \mathcal{P}_f(A))$ (where $\mathcal{P}_f(A)$ is the finite powerset of A); each term of the union is represented by the pair formed by the set of positive atoms and the set of negative atoms. For example $a \vee (b \wedge c \wedge \neg d) \vee \neg e$ is represented by $\{(\{a\}, \varnothing), (\{b, c\}, \{d\}), (\varnothing, \{e\})\}$. We use the letter u to range over sorts ($u \in \{\mathbf{basic}, \mathbf{prod}, \mathbf{fun}\}$).

Definition 2.1 (Shallow type expressions) *The functors $\mathcal{T}$ and $\mathcal{T}_u$ from sets to sets are defined by:*

$$\begin{array}{lcl}
\mathcal{T}X & = & \prod_u \mathcal{P}_f(\mathcal{P}_f(\mathcal{T}_u X) \times \mathcal{P}_f(\mathcal{T}_u X)) \\
\mathcal{T}_{\mathbf{basic}}X & = & \mathbb{B} \\
\mathcal{T}_{\mathbf{prod}}X & = & X^2 \\
\mathcal{T}_{\mathbf{fun}}X & = & X^2
\end{array}$$

An element (x, y) of $\mathcal{T}_{\mathbf{prod}}X$ (respectively, $\mathcal{T}_{\mathbf{fun}}X$) is written as $x \times y$ (respectively, $x \rightarrow y$).

Definition 2.2 (Type algebra) *A* type algebra *is a set T together with an implicit bijection $T \to \mathcal{T}T$. Elements of T (respectively, of $T_u = \mathcal{P}_f(\mathcal{P}_f(\mathcal{T}_u T) \times \mathcal{P}_f(\mathcal{T}_u T))$, and of $A_u = \mathcal{T}_u T$) are called* types *(respectively,* u-types*, and* atomic u-types*). A type t is thus a triple of u-types $(t_{\mathbf{basic}}, t_{\mathbf{prod}}, t_{\mathbf{fun}})$ that we write also $(t_u)_{u:sort}$.*

This definition means that every shallow type expression over types is again a type, and conversely, every type can be seen as a shallow type expression over types.[1]

Regularity. We can decompose a type $t \in T$ to the set of types it is built from (the b and the types just below a constructor). For t to be representable in a computer, the transitive closure of this decomposition must be finite.

Definition 2.3 (Base, regularity) *Let T be a type algebra; for a type $t = (t_u)_{u:sort}$ the* plinth $\sqsupset(t) \subseteq T \cup \mathbb{B}$ *collects all the elements of T and of $\mathbb{B}$ that appear in t seen as a shallow type expression over T:*

$$\sqsupset(t) = \bigcup_u \bigcup_{(P,N)\in t_u} \bigcup_{a\in P\cup N} \sqsupset(a)$$
$$\sqsupset(b) = \{b\} \text{ for } b \in \mathbb{B}$$
$$\sqsupset(t_1 \to t_2) = \sqsupset(t_1 \times t_2) = \{t_1, t_2\}$$

A finite *set $\mathcal{X} \subseteq T \cup \mathbb{B}$ is a* base *if for all $t \in \mathcal{X}$, $\sqsupset(t) \subseteq \mathcal{X}$. A type t is said to be $\mathcal{X}$-regular if $\sqsupset(t) \subseteq \mathcal{X}$; it is regular if it is $\mathcal{X}$-regular for some base $\mathcal{X}$. A type algebra is regular if all its elements are regular.*

Theorem 2.4 *Let $\mathcal{X}$ be a base. The set of $\mathcal{X}$-regular types is finite.*

The proof of this and all other theorems can be found in the extended version available at `http://www.cduce.org`.

The notion of base will be an important ingredient for proving the termination of the algorithms we define. Indeed, the algorithms proceed by saturating some sets of types via some operations; to prove the termination, it is enough to prove that all the generated types belong to a given base (which depends on the arguments of the algorithm).

Recursive type algebras. Let T be a type algebra. Recursive types are introduced as the solutions of systems of the form $\{x_1 = \tau_1; \ldots; x_n = \tau_n\}$ where the τ_i are either types (i.e., elements of T) or elements of $\mathcal{T}X$ with $X = \{x_1, \ldots, x_n\}$. A solution is a substitution σ from variables x_i to types t_i that makes the equations hold; a substitution is formally a function $\sigma : X \to T$, and $\mathcal{T}\sigma : \mathcal{T}X \to T = \mathcal{T}T$ is its extension to shallow expressions over X. Note that we do not require a solution to be unique: in this way we leave the implementation of the type algebra free to choose whether to make two types such as $\mu\alpha.\alpha \to \alpha$ and $\mu\beta.\beta \to \beta$ equal or not.

Definition 2.5 (Recursive type algebra) *A type algebra T is recursive if for every finite set X and every function $\tau : X \to T + \mathcal{T}X$, there exists a function $\sigma{:}X \to T$ such that $\sigma(x) = \tau(x)$ if $\tau(x) \in T$ and $\sigma(x) = (\mathcal{T}\sigma)s(x)$ if $\tau(x) \in \mathcal{T}X$.*

Convention 2.6 *From now on, we will fix a* regular *and* recursive *type algebra T and work with it.*

Finally let us introduce some notation that allows us to recover the BNF presentation given at the beginning of the section:

Notation 2.7 (Boolean connectives) *We define the following abbreviations for types:*

$$\mathbf{0} \stackrel{def}{=} (\mathbf{0}_u)_{u:sort} \qquad \mathbf{1} \stackrel{def}{=} (\mathbf{1}_u)_{u:sort}$$
$$t^1 \wedge t^2 \stackrel{def}{=} (t^1_u \wedge t^2_u)_{u:sort} \qquad t^1 \vee t^2 \stackrel{def}{=} (t^1_u \vee t^2_u)_{u:sort}$$
$$\neg t \stackrel{def}{=} (\neg_u t_u)_{u:sort}$$

where for every sort u we have:

$$\mathbf{0}_u \stackrel{def}{=} \varnothing \qquad \mathbf{1}_u \stackrel{def}{=} \{(\varnothing, \varnothing)\}$$
$$t^1_u \vee t^2_u \stackrel{def}{=} t^1_u \cup t^2_u$$
$$t^1_u \wedge t^2_u \stackrel{def}{=} \{(P^1 \cup P^2, N^1 \cup N^2) \mid (P^i, N^i) \in t^i_u\}$$
$$\neg_u t_u \stackrel{def}{=} \bigwedge_{(P,N)\in t_u} \{(\{a\}, \varnothing) \mid a \in N\} \cup \{(\varnothing, \{a\}) \mid a \in P\}$$

The reader can easily check that the definitions above simply correspond to the intuition of types as multi-sorted boolean combinations in disjunctive normal form in which boolean operators are applied component-wise. So for example the notation for negation is obtained by a simple application of De Morgan's laws.

Convention 2.8 *We identify an element $a \in A_u$ with $\{(\{a\}, \varnothing)\} \in T_u$ and an element $t_u \in T_u$ with $t = (t_{u'})_{u':sort} \in T$ where $t_{u'} = t_u$ for $u' = u$ and $t_{u'} = \mathbf{0}_{u'}$ for $u' \neq u$. This gives natural inclusions $A_u \subseteq T_u \subseteq T$.*

With this convention, every u-type t_u can be written:

$$t_u = \bigvee_{(P,N)\in t_u} \bigwedge_{a\in P} a \wedge \bigwedge_{a\in N} \neg_u a$$

(we omit the u in $\neg_u$ in such an equation when no confusion is possible) and a type t can be written as a (disjoint) union of u-types:

$$t = \bigvee_{u:\text{sort}} t_u$$

2.2. Models and subtyping relations

To define the subtyping relation, we must start by giving some meaning to basic types. In the syntactic approach this is done by fixing a subtyping relation for these basic types. In our semantic approach, this is obtained by choosing an interpretation of atomic basic types as sets.

[1] Types are products rather than coproducts as we want to consider heterogeneous combinations of types. For example, consider the type of integer lists $\mu\ell.(\mathsf{int}\times\ell)\vee\mathsf{nil}$. This expression denotes a type that is solution of the equation

$$x = (\ \{(\{\mathsf{nil}\}, \varnothing)\}\ ,\ \{(\{(\mathsf{int}, x)\}, \varnothing)\}\ ,\ \varnothing\ ).$$

Note that the first two projections of the tuple solution are not empty.

Definition 2.9 (Interpretation of $\mathbb{B}$) *An* interpretation *of $\mathbb{B}$ is a set $\mathbb{C}$ of* constants *together with a function $\mathcal{B}[\![\_]\!] : \mathbb{B} \to \mathcal{P}(\mathbb{C})$.*

Convention 2.10 *From now on we will always refer to a chosen fixed interpretation $(\mathbb{C}, \mathcal{B}[\![\_]\!])$. Moreover, for each constant $c \in \mathbb{C}$, we assume the existence of an atomic basic type $t_c \in \mathbb{B}$ such that, (i) $c \in \mathcal{B}[\![t_c]\!]$ and, (ii) $\forall b \in \mathbb{B}.\ (\mathcal{B}[\![t_c]\!] \subseteq \mathcal{B}[\![b]\!])$ or $(\mathcal{B}[\![t_c]\!] \cap \mathcal{B}[\![b]\!] = \varnothing)$.*

Premodels: structures and interpretations. Continuing the semantic approach to subtyping, we have to interpret all types as subsets of a given structure.

Definition 2.11 (Structure) *A* structure *is a* set *$\mathcal{D}$ partitioned into $\mathcal{D}_{\textbf{basic}}$, $\mathcal{D}_{\textbf{prod}}$, and $\mathcal{D}_{\textbf{fun}}$, together with two bijections $\partial_{\textbf{basic}} : \mathbb{C} \to \mathcal{D}_{\textbf{basic}}$ and $\partial_{\textbf{prod}} : \mathcal{D}^2 \to \mathcal{D}_{\textbf{prod}}$ such that the order $\partial_{\textbf{prod}}(d_1, d_2) > d_1$, $\partial_{\textbf{prod}}(d_1, d_2) > d_2$ is well-founded.*

Convention 2.12 *In a structure, the bijections $\partial_{\textbf{prod}}$ and $\partial_{\textbf{basic}}$ are kept implicit, so that we can write: $\mathcal{D}_{\textbf{prod}} = \mathcal{D}^2$ and $\mathcal{D}_{\textbf{basic}} = \mathbb{C}$.*

What we do next is to use a structure to define a set theoretic interpretation of our type algebra.

Definition 2.13 (Premodel) *Let T be a type algebra. A* premodel *of T is a structure $\mathcal{D}$ together with an interpretation function $[\![\_]\!] : T \to \mathcal{P}(\mathcal{D})$ that satisfies the following properties*[2] *(for all $b \in \mathbb{B}, t, t_1, t_2 \in T, u$: sort):*

1. $[\![b]\!] = \mathcal{B}[\![b]\!] \subseteq \mathcal{D}_{\textbf{basic}}$;
2. $[\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!] \subseteq \mathcal{D}_{\textbf{prod}}$;
3. $[\![t_1 \to t_2]\!] \subseteq \mathcal{D}_{\textbf{fun}}$;
4. $[\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$;
5. $[\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$;
6. $[\![\neg t]\!] = \mathcal{D} \backslash [\![t]\!]$;
7. $[\![\mathbf{1}_u]\!] = \mathcal{D}_u$.

As we see there is no real restriction on the interpretation of the arrow types as this is done at the level of *models*. The main reason for the definition of a premodel of T is that it induces the following subtyping relation on T:

Definition 2.14 (Subtyping) *Let T be a type algebra and $(\mathcal{D}, [\![\_]\!])$ a premodel of T. The subtyping relation $\leq_{\mathcal{D}}$ induced on T is defined as follows:*

$$t \leq_{\mathcal{D}} s \iff [\![t]\!] \subseteq [\![s]\!]$$

The associated equivalence is written $\simeq_{\mathcal{D}}$.

It is straightforward from its definition that $\leq_{\mathcal{D}}$ is a preorder relation. Note that $\leq_{\mathcal{D}}$ is determined by the equivalence class of $\mathbf{0}$, because $t \leq_{\mathcal{D}} s \Leftrightarrow t \wedge \neg s \simeq_{\mathcal{D}} \mathbf{0}$. Henceforward we will omit the subscript $\mathcal{D}$ of $\leq$ when clear from the context.

[2] This definition implies $[\![\mathbf{0}]\!] = \varnothing$. To make the use of $\partial_{\textbf{prod}}$ explicit, the second property should be read $[\![t_1 \times t_2]\!] = \{\partial_{\textbf{prod}}(d_1, d_2) \mid d_1 \in [\![t_1]\!]\ ,\ d_2 \in [\![t_2]\!]\}$. Moreover, because of the equality $\mathbf{1}_{\textbf{fun}} \wedge (t_1 \to t_2) = t_1 \to t_2$, the condition $[\![t_1 \to t_2]\!] \subseteq \mathcal{D}_{\textbf{fun}}$ is redundant and added just for clarity.

Models. Consider the type algebra T. For this algebra we have a class of possible premodels that vary according to the interpretation of arrow types of the algebra. Each premodel induces a subtyping relation. Now among all these premodels we select those whose induced subtyping relation satisfies a given property and we call them *models*; the property is that the subtyping relation behaves "as if" arrow types were interpreted extensionally, as sets of binary relations (graphs of possibly non-deterministic and non-terminating functions). For a premodel $\mathcal{D}$, we write $\mathcal{D}_\Omega = \mathcal{D} \cup \{\Omega\}$ where Ω is a distinguished element that does not belong to $\mathcal{D}$ (intuitively, it represents the type error).

Definition 2.15 (Extensional arrows) *Let T be a type algebra and $(\mathcal{D}, [\![\_]\!])$ a premodel of T. The* extensional interpretation *of an atomic* **fun***-type $(t \to s) \in A_{\textbf{fun}}$ is defined as*[3]*: $\mathcal{E}[\![t \to s]\!] = \{f \subseteq \mathcal{D} \times \mathcal{D}_\Omega \mid \forall (d_1, d_2) \in f.\ d_1 \in [\![t]\!] \Rightarrow d_2 \in [\![s]\!]\ \}$. We naturally extend this definition to every* **fun***-type $t_{\textbf{fun}} \in T_{\textbf{fun}}$:*

$$\mathcal{E}[\![t_{\textbf{fun}}]\!] = \textstyle\bigcup_{(P,N) \in t_{\textbf{fun}}} (\ (\bigcap_{a \in P} \mathcal{E}[\![a]\!]) \backslash (\bigcup_{a \in N} \mathcal{E}[\![a]\!])\ )$$

Definition 2.16 (Model) *A premodel $\mathcal{D}$ is a* model *if for every* **fun***-type $t_{\textbf{fun}}$, one has: $[\![t_{\textbf{fun}}]\!] = \varnothing \iff \mathcal{E}[\![t_{\textbf{fun}}]\!] = \varnothing$.*

This restriction is meaningful for at least three reasons :

1. It makes the subject reduction property for the language in Section 4 hold.
2. It makes the set of all well-typed values of the language be a model.
3. It matches the underlying intuition we have of function spaces as sets of possibly non-deterministic and non-terminating functions and of arrow types as constraints.

The following theorem gives a more explicit condition, that will be used in next section to derive a subtyping algorithm:

Theorem 2.17 *A premodel $\mathcal{D}$ is a model if and only if for all finite subsets $\{t_i \to s_i\}_{i \in I}$ and $\{t'_j \to s'_j\}_{j \in J}$ of $A_{\textbf{fun}}$ one has:*

$$\textstyle\bigwedge_{i \in I} (t_i \to s_i)\ \leq \bigvee_{j \in J} (t'_j \to s'_j) \iff$$
$$\exists j \in J. \begin{cases} t'_j \leq \bigvee_{i \in I} t_i \ \text{ and} \\ \forall I' \subseteq I.\ (t'_j \leq \bigvee_{i \in I'} t_i) \text{ or } (\bigwedge_{i \in I \backslash I'} s_i \leq s'_j) \end{cases}$$

The proof of this theorem relies on the observation that $\mathcal{E}[\![t \to s]\!] = \mathcal{P}(\complement_{\mathcal{D} \times \mathcal{D}_\Omega}([\![t]\!] \times \complement_{\mathcal{D}_\Omega}[\![s]\!]))$ (where $\complement_E F$ is $E \setminus F$, that is the complement of F with respect to E) and on the following lemma:

Lemma 2.18 *Let P and N be two finite sets and $(X_i)_{i \in P}$, $(Y_i)_{i \in P}$, $(X'_j)_{j \in N}$, $(Y'_j)_{j \in N}$ be four set families. Then:*

1. $\displaystyle\bigcap_{i \in P} (X_i \times Y_i) \backslash \bigcup_{j \in N} (X'_j \times Y'_j) =$

[3] The property in Definition 2.15 is one particular choice, but other choices are possible. For instance, in [4] we proposed a definition without Ω, and obtained a slightly different definition of model.

$$\bigcup_{N'\subseteq N}\left((\bigcap_{i\in P} X_i\setminus \bigcup_{j\in N'} X'_j)\times(\bigcap_{i\in P} Y_i\setminus \bigcup_{j\in N\setminus N'} Y'_j)\right);$$

2. $\bigcap_{i\in P}\mathcal{P}(X_i)\subseteq\bigcup_{j\in N}\mathcal{P}(X'_j)\iff\exists j\in N.\ \bigcap_{i\in P}X_i\subseteq X'_j$ *and the equivalence also holds for* $\mathcal{P}_f$ *instead of* $\mathcal{P}$.

2.3. Universal model and subtyping algorithm

So the model condition is sensible; but is it not too restrictive? Does there exists at least one premodel satisfying this condition, and can we exhibit an algorithm to compute its induced subtyping relation? The answers are all positive; note that for cardinality reasons, there is no structure $\mathcal{D}$ with $\mathcal{D}_{\mathbf{fun}}=\mathcal{P}(\mathcal{D}\times\mathcal{D}_\Omega)$, but Lemma 2.18 suggests to restrict to finite binary relations, that is take $\mathcal{D}_{\mathbf{fun}}=\mathcal{P}_f(\mathcal{D}\times\mathcal{D}_\Omega)$.

Theorem 2.19 (Universal model) *Let $\mathcal{S}$ be the premodel defined by interpreting functions as finite binary relations; that is, its structure is the set of all finite terms defined by:*

$$\begin{array}{lll} d & ::= c & c\in\mathbb{C}\\ & \mid\ (d_1,d_2) & \\ & \mid\ \{(d_1,d'_1),\ldots,(d_n,d'_n)\} & d'_i\in\mathcal{S}_\Omega=\mathcal{S}\cup\{\Omega\}\end{array}$$

and the interpretation of arrow types is defined by:

$$[\![t\to s]\!]=\{\{(d_1,d'_1),\ldots,(d_n,d'_n)\}\mid d_i\in[\![t]\!]\Rightarrow d'_i\in[\![s]\!]\}$$

Then:
1. *the premodel $\mathcal{S}$ is a model;*
2. *it is universal: for every model $\mathcal{D}$, and for all $t_1,t_2\in T.t_1\leq_{\mathcal{D}} t_2\ \Rightarrow\ t_1\leq_{\mathcal{S}} t_2$;*
3. *there is an algorithm that decides if $t_1\leq_{\mathcal{S}} t_2$.*

The universality property means that $\mathcal{S}$ is the model that induces the best possible subtyping relation. Note that even if in $\mathcal{S}$ types are sets of functions with finite graphs this *does not* mean that a language of this framework can only express functions with finite graphs (see Section 4). Furthermore $\mathcal{S}$ is not the only universal model: most of the reasonable (and more complex) models we can think of are universal and while there exist non-universal models, their construction is not trivial. Finally note that we are considering models *of the type algebra*: in principle these are not models of the language (e.g. there is no notion of application on *elements*).

The algorithm and the proof of universality involve the notion of *simulation*, which captures the idea of a coinductive set of rules (greatest fixpoint). The intended meaning is that a type is in some simulation if and only if its interpretation in the universal model $\mathcal{S}$ is the empty set. So the problem of finding a simulation containing $t\wedge\neg s$ is equivalent to checking $t\leq_{\mathcal{S}} s$. The actual definition of a simulation is motivated by Lemma 2.18 and Theorem 2.17.

Definition 2.20 (Simulation) *A* simulation *is a set of types $R\subseteq T$ s.t. for all $(t_v)_{v:sort}\in R, u{:}sort$ and $(P,N)\in t_u$:*

- *if* $u=$ **basic** *then:* $\bigcap_{b\in P}\mathcal{B}[\![b]\!]\subseteq\bigcup_{b\in N}\mathcal{B}[\![b]\!]$;
- *if* $u=$ **prod** *then for every* $N'\subseteq N$:
$$\left(\bigwedge_{(t\times s)\in P} t\ \wedge \bigwedge_{(t'\times s')\in N'}\neg t'\right)\in R\ \text{ or }\ \left(\bigwedge_{(t\times s)\in P} s\ \wedge \bigwedge_{(t'\times s')\in N\setminus N'}\neg s'\right)\in R;$$
- *if* $u=$ **fun** *then there is some* $(t'\to s')\in N$ *such that* $(t'\wedge\bigwedge_{(t\to s)\in P}\neg t)\in R$ *and for every* $P'\subseteq P$:
$$\left(t'\wedge\bigwedge_{(t\to s)\in P'}\neg t\right)\in R\ \text{ or }\ \left(\bigwedge_{(t\to s)\in P\setminus P'} s\wedge\neg s'\right)\in R.$$

Theorem 2.21 *Let t, s be two types. Then: $t\leq_{\mathcal{S}} s$ if and only if there is a simulation R such that $t\wedge\neg s\in R$.*

Let us assume that the inclusion between an intersection and a union of atomic basic types is decidable. Given a type t, deciding whether a type t belongs to some simulation can be done in a classical way[4]: start from the set $\{t\}$ and try to saturate it according to Definition 2.20 until reaching a simulation; because of the disjunctions in the definition of a simulation, the algorithm may have to fork and check different branches. The termination proof is easy: let $\mathcal{X}$ be a base such that t is $\mathcal{X}$-regular; then all the types that will be added to the current set are also $\mathcal{X}$-regular, and there are only a finite number a such types.

2.4. Destructors

To define the typing of patterns and expressions, we introduce type destructors, which are somewhat dual of type constructors $\times,\to$. Following our semantic approach, they will be characterized semantically (up to $\simeq_{\mathcal{D}}$ equivalence). This means that the destructors depend only on the subtyping relation induced by the (pre)model.

Theorem 2.22 (Projection) *Let $\mathcal{D}$ be a premodel, $\mathcal{X}$ a base and t a $\mathcal{X}$-regular type. If the inequality $t\leq_{\mathcal{D}} x\times\mathbf{1}$ has a solution x, then it has a least solution, written $\pi_1(t)$, which is also $\mathcal{X}$-regular. Moreover: $[\![\pi_1(t)]\!]=\{d_1\mid\exists d_2\in\mathcal{D}.\ (d_1,d_2)\in[\![t]\!]\}$. And similarly for π_2.*

Theorem 2.23 (Application) *Let $\mathcal{D}$ be a model, t and t' two types. If the inequality $t\leq_{\mathcal{D}} t'\to x$ has a solution x, then it has a least solution, written $t\bullet t'$. Moreover: $[\![t\bullet t']\!]=\{d_2\mid\exists f\in\mathcal{E}[\![t_{\mathbf{fun}}]\!].\ \exists d_1\in[\![t']\!].\ (d_1,d_2)\in f\}$.*

Explicit formulas for π_1 and $\bullet$ are given in the extended version of the article; however, the formal development only needs the semantic characterizations above and some properties we can deduce immediately, such as the monotonicity of π_1 and $\bullet$, or the fact that $(\bigwedge t_i\to s_i)\bullet s$ is defined if and only if $s\leq_{\mathcal{D}}\bigvee t_i$.

Actually, every **prod**-type can be decomposed as a finite union of atomic **prod**-types:

[4] We prefer to give the mathematical arguments necessary to prove a class of algorithms instead of giving a single fully explicit algorithm: this allows many small variations and heuristics, such as the choice of which types to add at a given step of the algorithm, or caching mechanism.

Theorem 2.24 *There exists an operator $\pi : T \to \mathcal{P}_f(T^2)$ such that, for every base $\mathcal{X}$ and every $\mathcal{X}$-regular type t:*
1. $t \wedge \mathbf{1}_{\mathbf{prod}} \simeq \bigvee_{(t_1,t_2)\in\pi(t)} t_1 \times t_2$*;*
2. $\forall(t_1,t_2)\in\pi(t).\forall i{=}1,2.(t_i$ *is $\mathcal{X}$-regular*$)$ and $(t_i \not\simeq \mathbf{0})$.

3. Pattern matching

In this section we define patterns and their semantics. Even though patterns are part of the language syntax, we can define their semantics in an arbitrary premodel.

3.1. Syntax

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

p	$::=$	x	capture, $x \in \mathbb{V}$
	$\mid$	t	type constraint, $t \in T$
	$\mid$	$p_1 \wedge p_2$	conjunction
	$\mid$	$p_1 \vert p_2$	alternative
	$\mid$	(p_1,p_2)	pair
	$\mid$	$(x := c)$	constant, $c \in \mathbb{C}$ with $[\![t_c]\!] = \{c\}$

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

Definition 3.2 (Patterns) *Given a type algebra T, and a set of variables $\mathbb{V}$, a pre-pattern p on $(\mathbb{V}, T)$ belongs to the set of (well-formed) patterns $\mathbb{P}$ on $(\mathbb{V}, T)$ if and only if it satisfies the following conditions:*

1. *the number of distinct subterms of p is finite (regularity);*
2. *for every infinite branch of p there are an infinite number of occurrences of pair nodes;*
3. *for every subterm $p_1 \wedge p_2$ of p we have $Var(p_1) \cap Var(p_2) = \varnothing$, and for every subterm $p_1|p_2$ of p we have $Var(p_1) = Var(p_2)$.*

In short, patterns are pre-patterns that (i) are regular trees, (ii) come equipped with a well-founded order (defined by $p_1 \wedge p_2 > p_1, p_2$ and $p_1|p_2 > p_1, p_2$), and (iii) in which variables in conjunctions and alternatives must satisfy some reasonable conditions. The second condition in particular means that patterns have to deconstruct values sooner or later, thus ensuring the termination of pattern matching.

3.2. Dynamic semantics

This semantics of pattern matching will be used to define in Section 4 the operational semantics of the language. It is defined here with respect to a premodel of the type algebra at issue. More precisely, we denote by d/p the result of matching an element $d \in \mathcal{D}$ with the pattern p. This yields either a failure, denoted by Ω, or a substitution of the variables of p into $\mathcal{D}$. Formally it is defined as follows:

Definition 3.3 (Semantics of pattern matching) *Given $d \in \mathcal{D}$ and $p \in \mathbb{P}$ the matching of d with p, denoted by d/p, is the element of $\mathcal{D}^{Var(p)} \cup \{\Omega\}$ defined by induction on the lexicographically ordered pair (d, p) as follows:*

$$
\begin{array}{lcll}
d/t & = & \{\} & \text{if } d \in [\![t]\!] \\
d/t & = & \Omega & \text{if } d \in [\![\neg t]\!] \\
d/x & = & \{x \mapsto d\} & \\
d/p_1 \wedge p_2 & = & d/p_1 \otimes d/p_2 & \\
d/p_1|p_2 & = & d/p_1 & \text{if } d/p_1 \neq \Omega \\
d/p_1|p_2 & = & d/p_2 & \text{if } d/p_1 = \Omega \\
(d_1,d_2)/(p_1,p_2) & = & d_1/p_1 \otimes d_2/p_2 & \\
d/(p_1,p_2) & = & \Omega & \text{if } d \notin \mathcal{D}_{\mathbf{prod}} \\
d/(x := c) & = & \{x \mapsto c\} &
\end{array}
$$

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

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

This definition is rather intuitive. There are two possible causes of failure for a pattern matching: a type constraint which is not satisfied by the matched object, or a pair pattern applied to an object which is not a pair. The alternative pattern $p_1|p_2$ has a first-match policy: the object is matched against p_2 if and only if matching with p_1 raises a failure. When a variable x appears on both sides of a pair pattern, the two captured elements are paired together. The pattern $(x := c)$ usually appears in the right-hand side of an alternative pattern to give a default value when the left-hand side fails; for instance, the variable x in the pattern $(x,\mathbf{1})|(x := c)$ extracts the first component of a pair, or is bound to the constant c when the matched value is not a pair.

3.3. Examples of patterns

To demonstrate the power of our pattern algebra, we show by some examples how recursive and pair patterns may be used to work with (heterogeneous) sequences. In this section, sequences are coded *à la* Lisp; a sequence is either $[\,]$ (a constant with $[\![t_{[\,]}]\!] = \{[\,]\}$) or a pair $(head, tail)$.

1. When applying the recursive pattern $p_1 = (x \wedge t,\mathbf{1})|(\mathbf{1},p_1)$ to a sequence, the variable x captures the first element of type t in the sequence. The operational behavior of p is simple: assume that the sequence is a pair $(head, tail)$; if *head* is of type t, then x captures it; otherwise, the matching continues with *tail*. Note that to capture the last element of type t in a sequence, it suffices to reverse the order of the patterns in the alternative.

2. The pattern $p_2 = (x \wedge t,p_2)|(\mathbf{1},p_2)|(x := [\,])$ captures from a sequence all the elements of a given type t and returns in x the sequence of these elements. Let us describe the operational behavior. If the sequence is a pair $(head, tail)$, there are two cases; if *head* is of type t, then x captures *head*, the matching continues with *tail* returning a sequence in x, and finally the two values for x are paired (that is, *head* is put in front of the returned sequence), as stated by the last case of $\otimes$ (Definition 3.3);

if *head* is not of type t, then the matching simply continues with *tail*. If instead the sequence is not a pair (the end has been reached) the empty sequence is returned.

3. The pattern $p_3 = (x,(\mathbf{1},p_3))|(x := [\,])$ captures from a sequence all the elements whose rank is odd (first, third, fifth, ...).
4. The pattern $p_4 = (x \wedge t_1,(x \wedge t_2,\mathbf{1}))|(\mathbf{1},p_4)$ captures from a sequence the first two consecutive elements d_1 of type t_1 and d_2 of type t_2, and returns the pair (d_1, d_2), whereas the pattern $p'_4 = (x \wedge t_1,(x \wedge t_2,(x := [\,])))|(\mathbf{1},p'_4)$ would return instead a sequence of length 2 with these two elements, that is $(d_1,(d_2,[\,]))$.
5. The pattern $p_5 = (x \wedge t,q)|(\mathbf{1},p_5)$ with $q = (x \wedge t,q)|(x := [\,])$ captures from a sequence the first and longest consecutive subsequence of elements of type t.

3.4. Static semantics and pattern algorithms

To give typing rules for pattern matching in the language, we need to study the behavior of patterns on types.

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

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

In other terms, $\wr p \int$ is a type formed by all and only those elements that make p succeed. If t is a subset of this type, then (t/p) denotes the type environment of the variables of p, as it associates to each variable x of p the *exact* type of its values, when the pattern is matched to an element of type t. As the previous section demonstrates, our pattern algebra allows to express complex extractions, and we get exact typing even for them, as opposed to XDuce pattern matching algorithm [6] where variables capturing subsequences that are not in tail position do not get exact typing (the idea underlying the algorithm of each theorem is to derive a system of equations on types from the semantic condition, and use a general algorithm to solve it, as described in Appendix A).

For example, consider the pattern p_1 of Section 3.3. This pattern succeeds if and only if it is applied to a sequence containing at least one element of type t. Indeed, the algorithm returns for $\wr p_1 \int$ a type $s \simeq (t \times \mathbf{1}) \vee (\mathbf{1} \times s)$. Consider now the type u of the sequences alternating elements of type t_1 and t_2, i.e. $u = (t_1 \times (t_2 \times u)) \vee t_{[\,]}$, and the type environment returned by applying p_1 to $u \wedge \wr p_1 \int$ (which defines the type of x, i.e. the only capture variable of p_1). If we write t'_i for $t_i \wedge t$, then the algorithm returns for $(u \wedge \wr p_1 \int / p_1)(x)$ the type t_1 if $t_1 \leq t$, and the type $t'_1 \vee t'_2$ otherwise.

4. The ℂDuce language

In the previous sections we defined the type system, the subtyping relation, the set of patterns, and a semantic notion of matching. We now use all these notions for the definition of the functional core of the ℂDuce language. A detailed description of the language and an interactive prototype are available at `http://www.cduce.org`.

4.1. Syntax

The sets $\mathbb{C}$, $\mathbb{V}$ and $\mathbb{P}$ of constants, variables and patterns have already been introduced. Let $\mathbb{O}$ denote a set of *operators*. The set $\mathbb{E}$ of expressions is defined by the syntax:

$e ::=$	c	constant, $c \in \mathbb{C}$
$\mid$	$o(e)$	operator, $o \in \mathbb{O}$
$\mid$	x	variable, $x \in \mathbb{V}$
$\mid$	$\boldsymbol{\mu} f^{(t_1 \to s_1; \ldots; t_n \to s_n)}(x).e$	abstraction
$\mid$	$e_1 e_2$	application
$\mid$	(e_1,e_2)	pair
$\mid$	$\mathtt{match}\ e\ \mathtt{with}\ p_1 \Rightarrow e_1 \mid p_2 \Rightarrow e_2$	pattern matching

Before defining the operational semantics of the language we need to define its type system. Indeed the computations of our language are completely type-dependent. This is so because the pattern matching semantics is defined for an interpretation of types as sets of values, and this interpretation is induced by the type system. This may result clearer if we observe that with this syntax for every type t one can define its characteristic function χ_t as follows:

$$\boldsymbol{\mu}\chi_t^{(\mathbf{1} \to \mathsf{bool})}(x).\mathtt{match}\ x\ \mathtt{with}\ t \Rightarrow \mathsf{true} \mid \neg t \Rightarrow \mathsf{false}$$

It is then obvious that it is not possible to give the semantics of this expression without having associated a type to each expression (or at least to each value).

As the semantics depends on the type system, and the rules of the type systems are motivated by the semantics, we first introduce these two objects formally and then comment them and the constructs of the syntax above together.

4.2. Type system

To define the type system, we have to fix:

- an arbitrary *bootstrap* model $\mathcal{D}$; it induces a subtyping relation $\leq$, which in turns defines the type operators π_i and $\bullet$, and the pattern matching operator (t/p);
- for each operator $o \in \mathbb{O}$, a type $t_o \in T$ and a monotonic function $o[\_] : \{t \in T \mid t \leq t_o\} \to T$. Intuitively, the type t_o denotes all the values on which the operator can operate, and $o[t]$ denotes all the possible results of applying the operator to a value of type t.

The typing judgment $\Gamma \vdash e : t$ is defined by the set of rules in Figure 1. The environments Γ are partial maps from variables to types. We work modulo α-conversion and always suppose that when two environments are concatenated, their domains are disjoint.

4.3. Small-step operational semantics

Now that, thanks to the bootstrap model, we defined the well-typed terms of the language, we can select among them the set $\mathcal{V}$ of *values*. More precisely we distinguish among all *closed* and *well-typed* expressions the following ones:

$$\frac{}{\Gamma \vdash c : t_c}\ (\textit{const}) \qquad \frac{}{\Gamma \vdash x : \Gamma(x)}\ (\textit{var}) \qquad \frac{\Gamma \vdash e : t \leq t_o}{\Gamma \vdash o(e) : o[t]}\ (\textit{op}) \qquad \frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1,e_2) : t_1 \times t_2}\ (\textit{pair})$$

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

$$\begin{array}{l}(\text{for } s_1 \equiv s \wedge \llparenthesis p_1 \rrparenthesis,\ s_2 \equiv s \wedge \neg \llparenthesis p_1 \rrparenthesis)\\ \dfrac{\Gamma \vdash e : s \leq \llparenthesis p_1 \rrparenthesis \vee \llparenthesis p_2 \rrparenthesis \qquad \Gamma, (s_i/p_i) \vdash e_i : t_i}{\Gamma \vdash \mathtt{match}\, e\, \mathtt{with}\, p_1 \Rightarrow e_1 \,|\, p_2 \Rightarrow e_2 : \bigvee_{\{i \mid s_i \not\simeq \mathbf{0}\}} t_i}\ (\textit{match})\end{array} \qquad \frac{\Gamma \vdash e : s \leq t}{\Gamma \vdash e : t}\ (\textit{subsum})$$

Figure 1. Typing rules

$$v \quad ::= \quad c \mid \mu f^{(t_1 \to s_1; \ldots; t_n \to s_n)}(x).e \mid (v_1,v_2)$$

The type system gives a natural interpretation of types as sets of values, and it turns out that $\mathcal{V}$ is indeed a model.

Theorem 4.1 *Let $[\![t]\!]_{\mathcal{V}} = \{v \mid\ \vdash v : t\}$. The pair $(\mathcal{V}, [\![\_]\!]_{\mathcal{V}})$ is a model and it induces the same subtyping relation as the bootstrap model.*

This result depends on the presence of multiple arrow types in abstraction interfaces; for instance, without these overloaded functions, a relation such as $(t_1 \to s_1) \wedge (t_2 \to s_2) \simeq (t_1 \vee t_2) \to (s_1 \wedge s_2)$ would hold in $\mathcal{V}$, and we would have to adapt the definition of model to get a corresponding property.

Two important consequences of this theorem are that (i) a value is of type t if and only if it is not of type $\neg t$, and (ii) $t \leq s$ really means that all the values of type t are values of type s. Since the type destructors $\bullet$, π_i and the pattern matching operators on types $\llparenthesis p \rrparenthesis$ and (t/p) depend only on the subtyping relation induced by the (pre)model, then one can reinterpret their meaning in terms of sets of values.

The operational semantics of the language is given in Figure 2, where Ω is a special object denoting a type error, which is not an expression of the language; v/p is the substitution that results from applying pattern p to v, as introduced in Definition 3.3 for an arbitrary premodel (note that the definition v/p depends on the model $\mathcal{V}$, hence on the type system); $e[\sigma]$ is the application of the substitution σ to the expression e (the standard substitution $[v/x]$ is a special case in which the pattern is a capture variable); and finally $C[\,]$ denotes an evaluation context, defined as:

$$\begin{array}{l} C[\,] ::= [\,] \mid o(C[\,]) \mid (C[\,],e) \mid (e,C[\,]) \mid C[\,]e \mid eC[\,] \mid \\ \qquad (\mathtt{match}\ C[\,]\ \mathtt{with}\ p_1 \Rightarrow e_1 \,|\, p_2 \Rightarrow e_2) \end{array}$$

For each operator o, we fix a binary relation $\stackrel{o}{\to} \subseteq [\![t_o]\!]_{\mathcal{V}} \times \mathbb{E}$ such that: if $\vdash v : t$, $t \leq t_o$ and $v \stackrel{o}{\to} e$ then $\vdash e : o[t]$.

4.4. Commentaries on the language, its type system and its semantics

The language syntax has constants, variables and pairs, which are used as customary and do not deserve any particular comments. The (*subsum*) rule implies that the type system is semantic in the sense that the typing judgment $\vdash e : t$ only depends on the $\simeq$-equivalence class of t. Operator applications are precisely typed: $o[\_]$ can be seen as an abstract version of $o(\_)$, describing its behavior when only the type of the argument is known.

The matching expression follows a first match policy: the second pattern is used if and only if the first one failed to match the value (that is, $v/p_1 = \Omega$). Each p_i binds the variables $\textit{Var}(p_i)$ in e_i. To understand the typing rule (*match*), recall that (t/p) is the typing environment that maps every variable $x \in \textit{Var}(p)$ to the type formed by all the values that x can assume when a value of type t is matched against the pattern p. For the pattern matching to be exhaustive, the type s of the matched value must be a subtype of $\llparenthesis p_1 \rrparenthesis \vee \llparenthesis p_2 \rrparenthesis$. Because of the first match policy, the value has actually type $s \wedge \llparenthesis p_1 \rrparenthesis$ when p_1 succeed to match it, and type $s \wedge \neg \llparenthesis p_1 \rrparenthesis$ otherwise. If a pattern cannot succeed ($s_i \simeq \mathbf{0}$), the result of the corresponding branch is discarded: this is useful when typing an overloaded abstraction (some branches may be useless when checking under a given constraint in the interface, and their result types must not be taken into account to prove the constraint at issue: see the example farther on).

Less standard is the definition of functions. The expression $\mu f^{(t_1 \to s_1; \ldots; t_n \to s_n)}(x).e$ denotes a possibly recursive definition (as f may occur free in e) and it intuitively corresponds to the least fixpoint of $\lambda f.\lambda x.e$; the arrow types in an abstraction are constraints on its behavior. Together with pattern matching, they allow f to be an overloaded function.

The rule (*abstr*) may seem overly complicated; it is a refined version of:

$$\frac{(\forall i)\, \Gamma, (x : t_i), (f : \bigwedge t_i \to s_i) \vdash e : s_i}{\Gamma \vdash \mu f^{(t_1 \to s_1; \ldots; t_n \to s_n)}(x).e : \bigwedge t_i \to s_i}\ (\textit{abstr'})$$

which probably looks more familiar. In this form, it is the standard rule for λ-abstraction, the body being type-checked once for each constraint given in the abstraction. A closed and well-typed abstraction $v = \mu f^{(t_1 \to s_1; \ldots; t_n \to s_n)}(x).e$ is a value of the language, so for any type t, one must have either $\vdash v : t$ or $\vdash v : \neg t$; with the (*subsum*) rule, one can prove that v has type $t \to s$ if and

Reductions	$o(v)$	$\to$ e	if	$\vdash v : t_o, v \xrightarrow{o} e$
	$v_1 v_2$	$\to$ $e[v_1/f; v_2/x]$	if	$v_1 = \mu f^{(\dots)}(x).e$
	$\mathtt{match}\, v\, \mathtt{with}\, p_1 \Rightarrow e_1 \mid p_2 \Rightarrow e_2$	$\to$ $e_1[v/p_1]$	if	$v/p_1 \neq \Omega$
	$\mathtt{match}\, v\, \mathtt{with}\, p_1 \Rightarrow e_1 \mid p_2 \Rightarrow e_2$	$\to$ $e_2[v/p_2]$	if	$v/p_1 = \Omega, v/p_2 \neq \Omega$
	$C[e_1]$	$\to$ $C[e_2]$	if	$e_1 \to e_2$
Errors	$o(v)$	$\to$ Ω	if	$\not\vdash v : t_o$
	$v_1 v_2$	$\to$ Ω	if	$v_1 \notin \mathcal{V}_{\mathbf{fun}}$
	$\mathtt{match}\, v\, \mathtt{with}\, p_1 \Rightarrow e_1 \mid p_2 \Rightarrow e_2$	$\to$ Ω	if	$v/p_1 = \Omega, v/p_2 = \Omega$
	$C[e]$	$\to$ Ω	if	$e \to \Omega$

Figure 2. Small-step operational semantics

only if $\bigwedge t_i \to s_i \leq t \to s$. But when this is not the case, the (*abstr'*) rule does not prove that $\vdash v : \neg(t \to s)$. This is exactly the purpose of the conjunction of negative arrow types t' in the rule (*abstr*).

We next show an example to illustrate the need of discarding useless branches when typing a pattern matching in the body of an overloaded function. Consider the typing of:
$\mu f^{(\mathsf{int} \to \mathsf{bool}; \mathsf{bool} \to \mathsf{int})}(x).\mathtt{match}\; x\; \mathtt{with}\; \mathsf{int} \Rightarrow \mathsf{true} \mid \mathsf{bool} \Rightarrow 3$
When checking the first constraint of the interface, $\mathsf{int} \to \mathsf{bool}$, the pattern matching is typed under the assumption x:int; since the first branch of the match accepts any value of type int, then the second branch is useless (this corresponds to $s_2 \simeq \mathbf{0}$ in rule (*match*)). If we were obliged to take its result type int into account, then we could not prove the constraint $\mathsf{int} \to \mathsf{bool}$, but just $\mathsf{int} \to (\mathsf{bool} \vee \mathsf{int})$.

The abstraction interface can be used to express fine-grained constraints on the behavior; let us show this by an example. Let b and empty be two atomic basic types (for instance, $\mathsf{empty} = t_{[]}$) and t, s two types such that $t = s \vee \mathsf{empty}$ and $s = b \times t$. Values of type t can be seen as lists of elements of type b (with a terminator in empty), and values of type s are non-empty lists. A concatenation function can be written:

$$\begin{aligned} &\mu f^{(t \times t \to t)}(x).\mathtt{match}\; x\; \mathtt{with} \\ &\qquad (\mathsf{empty}, \ell) \Rightarrow \ell \\ &\qquad \mid ((\mathit{head}, \mathit{tail}), \ell) \Rightarrow (\mathit{head}, f(\mathit{tail}, \ell)) \end{aligned}$$

But we can use a stronger constraint in the interface —and force a better type (strictly smaller than $t \times t \to t$)— such as $((t \times t) \wedge \neg(\mathsf{empty} \times \mathsf{empty}) \to s\, ;\, (\mathsf{empty} \times \mathsf{empty}) \to \mathsf{empty})$ and leave the rest unchanged: the function still type-checks. Thus besides to type the union of different expressions, overloaded types can also be used to give a finer description of the behavior of a same expression.

4.5. Properties

Theorem 4.2 (Subsumption elimination) *Let $\Gamma \Vdash e : t$ denote the judgment defined by the rules in Figure 3 plus the rules in Figure 1 without (abstr), (appl) and (subsum). If $\Gamma \vdash e : t$ then there exists a type $t' \leq t$ such that $\Gamma \Vdash e : t'$.*

Using a substitution lemma and the semantic characterization of the pattern matching type operator (t/p), it is very easy to prove that well-typed programs cannot go wrong:

Theorem 4.3 (Subject reduction) *If $\vdash e : t$, then*
(1) $e \not\xrightarrow{*} \Omega$ *and*
(2) $e \xrightarrow{*} e'$ *implies* $\vdash e' : t$.

Note that the second point (preservation of typing) does *not* hold for the type system with (*abstr'*) instead of (*abstr*) while the first one (absence of type error) of course does. Though not as important as the subject reduction property, the following result shows that the circle is really complete:

Theorem 4.4 *Let t_1 and t_2 be two types such that $t_1 \bullet t_2$ is defined. Then: $\vdash v : t_1 \bullet t_2 \Leftrightarrow \exists v_1, v_2.\ (\vdash v_1 : t_1) \wedge (\vdash v_2 : t_2) \wedge (v_1 v_2 \xrightarrow{*} v)$.*

4.6. Typing algorithms

For the type-driven dynamic semantics to be effective, one must be able to decide the *type-checking* problem for the *values* of the language; in other terms one must be able to decide if for a given type t and value v, $\vdash v : t$ holds. Note that as v is a value, it is by definition a well-typed expression. Here is a naive algorithm. First determine the universe u of v. The problem is then to check if there is some $(P, N) \in t_u$ such that $\forall a \in P.\ v \in [\![a]\!]$ and $\forall a \in N.\ v \notin [\![a]\!]$. If v is a constant c, the tests $(c \in \mathcal{B}[\![b]\!])$ are easy. If v is an abstraction $\mu f^{(t_1 \to s_1; \dots; t_n \to s_n)}(x).e$, the condition $(v \in [\![t_0 \to s_0]\!])$ simply means $\bigwedge t_i \to s_i \leq t_0 \to s_0$. If v is a pair (v_1, v_2), the condition $(v \in [\![t_1 \times t_2]\!])$ means $(v_1 \in [\![t_1]\!]$ and $v_2 \in [\![t_2]\!])$, hence two recursive calls.

For the static semantics to be effective, one must be able to decide the *type-inference* problem for the *expressions* of

(for $t \equiv \bigwedge_{i=1..n} t_i \to s_i$)

$$\frac{(\forall j)\, t \not\leq t'_j \to s'_j \quad (\forall i)\, \Gamma, (x{:}t_i), (f{:}t) \Vdash e : u_i \leq s_i}{\Gamma \Vdash \mu f^{(t_1 \to s_1; \dots; t_n \to s_n)}(x).e : t \wedge \bigwedge_{j=1..m} \neg(t'_j \to s'_j)}$$

$$\frac{\Gamma \Vdash e_1 : t_1 \quad \Gamma \Vdash e_2 : t_2}{\Gamma \Vdash e_1 e_2 : t_1 \bullet t_2}$$

Figure 3. Semi-algorithmic rules

the language; that is if for a given expression e, there exists a type t such that $\vdash e : t$ holds. Theorem 4.2 is a step towards an algorithm, as it removes the only typing rule not associated to a syntactic construction in the language. Thus $\Vdash$ is syntax-directed, but it does not satisfy the subformula property since (*abstr*) does not impose the choice of the negative arrow types. Therefore this rule may require the algorithm to backtrack, possibly infinite many times. As these negative arrow types are mainly a technical trick to make the subject reduction theorem hold, one may consider instead of (*abstr*) the more restrictive rule (*abstr'*) discussed before, and we get immediately a (not complete) type-inference algorithm.

Acknowledgments We are very grateful to Juliusz Chroboczek, Mariangiola Dezani, Haruo Hosoya, Benjamin Pierce, François Pottier and Jérôme Vouillon for the interesting discussions and for their comments and stimulating remarks which helped us to improve the presentation. We would like to thank Francesco Zappa Nardelli for suggesting the name ℂDuce. Work partially supported by the European FET contract *MyThS*, IST-2001-32617.

References

[1] A. Aiken and E. 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.

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

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

[4] A. Frisch. Types récursifs, combinaisons booléennes et fonctions surchargées: application au typage de XML. DEA *Programmation*, Université Paris 7, Sept. 2001. Available at `http://www.di.ens.fr/~frisch`.

[5] H. Hosoya and B. C. Pierce. XDuce: A typed XML processing language. In *Proceedings of Third International Workshop on the Web and Databases (WebDB2000)*, 2000.

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

[7] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. In *Proceedings of the International Conference on Functional Programming (ICFP)*, 2000.

A. Pattern algorithms

Definition A.1 *Let V be a* finite *set of variables, written α, α_1, α_2, β, ... A right-hand side is either a type $t \in T$, a disjunction $\alpha_1 \vee \alpha_2$, a conjunction $\alpha_1 \wedge \alpha_2$, a negation $\neg\alpha$, a product $\alpha_1 \times \alpha_2$, or a variable α. A system on V is a mapping S from variables to right-hand sides. The system is also written $(\alpha \equiv S(\alpha))_{\alpha \in V}$.*

A semantic solution for this system is a mapping $s : V \to \mathcal{P}(\mathcal{D})$ such that for every $\alpha \in V$: $s(\alpha) = [\![t]\!]$ when $S(\alpha) = t$, $s(\alpha) = s(\alpha_1) \cup s(\alpha_2)$ when $S(\alpha) = \alpha_1 \vee \alpha_2$, $s(\alpha) = [\![\alpha_1]\!] \cap [\![\alpha_2]\!]$ when $S(\alpha) = \alpha_1 \wedge \alpha_2$, $s(\alpha) = \mathcal{D} \backslash s(\alpha_1)$ when $S(\alpha) = \neg\alpha_1$, $s(\alpha) = s(\alpha_1) \times s(\alpha_2)$ when $S(\alpha) = \alpha_1 \times \alpha_2$ and $s(\alpha) = s(\alpha_1)$ when $S(\alpha) = \alpha_1$. A syntactic representation of a semantic solution is a mapping $\sigma : V \to T$ such that for every $\alpha \in V$, $[\![\sigma(\alpha)]\!] = s(\alpha)$.

For a given system S, we write $\alpha \rightsquigarrow \beta$ if $S(\alpha)$ is either the variable β or a negation,disjunction or conjunction where β appears.

Theorem A.2 (Guarded systems) *Let S be a system. Assume that the relation $\rightsquigarrow$ has no cycle. Then S has a unique semantic solution s, and there is an algorithm to compute a syntactic representation of s.*

Theorem A.3 *Let p be a pattern; the semantic condition in the definition of $\llparenthesis p \rrparenthesis$ is equivalent to the following guarded system of equations (the variables are the $\llparenthesis p' \rrparenthesis$ for the subterms p' of p):*

$$
\begin{array}{lcl@{\qquad}lcl}
\llparenthesis x \rrparenthesis & \equiv & \mathbf{1} & \llparenthesis p_1|p_2 \rrparenthesis & \equiv & \llparenthesis p_1 \rrparenthesis \vee \llparenthesis p_2 \rrparenthesis \\
\llparenthesis t \rrparenthesis & \equiv & t & \llparenthesis p_1 \wedge p_2 \rrparenthesis & \equiv & \llparenthesis p_1 \rrparenthesis \wedge \llparenthesis p_2 \rrparenthesis \\
\llparenthesis (x := c) \rrparenthesis & \equiv & \mathbf{1} & \llparenthesis (p_1,p_2) \rrparenthesis & \equiv & \llparenthesis p_1 \rrparenthesis \times \llparenthesis p_2 \rrparenthesis
\end{array}
$$

By combining the two previous theorems, we get an algorithm to compute $\llparenthesis p \rrparenthesis$.

Theorem A.4 (Positive systems) *A system without negation and intersection has a smallest semantic solution s ($\mathcal{P}(\mathcal{D})^V$ being ordered by pointwise inclusion), and there is an algorithm to compute a syntactic representation of s.*

Theorem A.5 *Let p be a pattern, t a type such that $t \leq \llparenthesis p \rrparenthesis$ and $x \in Var(p)$. Let $\mathcal{X}$ be a base containing t and all the $\llparenthesis p' \rrparenthesis$ for the subterms p' of p. The semantic condition in the definition of $(t/p)(x)$ corresponds to the smallest solution of the following positive system (the variables are the $(t'/p')(x)$ for the subterms p' of p and the $\mathcal{X}$-regular types t'):*

$$
\begin{array}{lcll}
(t'/x)(x) & \equiv & t' & \\
(t'/p_1|p_2)(x) & \equiv & \multicolumn{2}{l}{((t' \wedge \llparenthesis p_1 \rrparenthesis)/p_1)(x) \vee ((t' \wedge \neg \llparenthesis p_1 \rrparenthesis)/p_2)(x)} \\
(t'/p_1 \wedge p_2)(x) & \equiv & (t'/p_i)(x) & \textit{if } x \in Var(p_i) \\
(t'/(p_1,p_2))(x) & \equiv & \displaystyle\bigvee_{(t_1,t_2) \in \pi(t')} (t_1/p_1)(x) \times (t_2/p_2)(x) & \\
 & & & \textit{if } x \in Var(p_1) \cap Var(p_2) \\
(t'/(p_1,p_2))(x) & \equiv & (\pi_1(t')/p_1)(x) & \textit{if } x \in Var(p_1) \backslash Var(p_2) \\
(t'/(p_1,p_2))(x) & \equiv & (\pi_2(t')/p_2)(x) & \textit{if } x \in Var(p_2) \backslash Var(p_1) \\
(t'/(x := c))(x) & \equiv & t_c & \textit{if } t' \not\simeq \mathbf{0} \\
(t'/(x := c))(x) & \equiv & \mathbf{0} & \textit{if } t' \simeq \mathbf{0}
\end{array}
$$

(formally, we have to introduce a finite number of extra variables, because our definition of a system has only binary disjunctions or products of variables on right-hand side)

By combining the two previous theorems, we get an algorithm to compute (t/p).