Semantic Subtyping: Dealing Set-Theoretically with Function, Union, Intersection, and Negation Types

ALAIN FRISCH

LexiFi, Boulogne-Billancourt, France

GIUSEPPE CASTAGNA

CNRS, PPS - Université Paris 7, Paris, France

AND

VÉRONIQUE BENZAKEN

LRI - Université Paris Sud, Orsay, 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. This work shows how to define a subtyping relation semantically in the presence of Boolean connectives, functional types and dynamic dispatch on types, without the complexity of denotational models, and how to derive a complete subtyping algorithm.

Categories and Subject Descriptors: F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*Type structure*

General Terms: Languages, Theory

Additional Key Words and Phrases: Subtyping, union types, intersection types, negation types, higher-order functions

ACM Reference Format:

Frisch, A., Castagna, G., and Benzaken, V. 2008. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. J. ACM 55, 4, Article 19 (September 2008), 64 pages. DOI = 10.1145/1391289.1391293 http://doi.acm.org/10.1145/1391289.1391293

This work was partially supported by the French ACI project "Transformation Languages for XML: Logics and Applications"(TraLaLA)

Authors' addresses: A. Frisch, LexiFi SAS, 38 rue Vauthier, F-92100, Boulogne-Billancourt France; G. Castagna, PPS-CNRS, Université Denis Diderot, 175 rue du Chevaleret, 75013 Paris, France; V. Benzaken, LRI-CNRS, Bat. 490, Université Paris Sud, 91405 Orsay, France.

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 axiomatizing it in a formal deduction system (a set of inductive or co-inductive rules); in the semantic approach (i.e., Aiken and Wimmers [1993] and Damm [1994a]), 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 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 [Hosoya and Pierce 2001, 2003; Hosoya 2001] 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 their model of types the set of values of the language. Their notion of model cannot capture functional values (their sets of values are regular languages that, as it is well known, are not closed with respect to functional spaces). 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 a previous work [Frisch et al. 2002; Frisch 2004], we extended the work on XDuce and reframed it in a more general setting: we showed 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 Frisch et al. [2002] and Frisch [2004] was to provide a theoretical foundation on the top of which to build the language ℂDuce [Benzaken et al. 2003], an XML-oriented transformation language. The key theoretical contribution of the work is a new approach to define semantic subtyping when straightforward set-theoretic interpretation does not work, in particular for arrow types. Here we focus and expand on this aspect of the work and we get rid of many features (e.g., pattern matching and pattern variable type inference) that are not directly related to the treatment of subtyping.

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 emphasize 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 Frisch et al. [2002] and Frisch [2004] 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 generalization of lately bound overloaded functions to intersection types. The intuition that overloaded functions should be typed by intersection types was always felt but never fully formalized or understood. On the one hand, we had the longstanding research on intersection types with the seminal works by the Turin research group on Curry-style typed lambda calculus [Barendregt et al. 1983; Coppo and Dezani-Ciancaglini 1980] (and later pursued in Church-style by John Reynolds' work on coherent overloading and the language Forsythe [Reynolds 1991, 1996]). However, functions with intersection types had a uniform behavior, in the sense that even if they worked on arguments of different types they always executed the same code for all of these types. So functions with intersection types looked closer to finitely *parametric* (in the sense of Reynolds [1983]) polymorphic functions (in which we enumerate the possible input types) that cannot reconstruct values of the (intuitive) finite range parametric type,[1] rather than overloaded functions that are able to discriminate on the type of the argument, execute a different code for each different type and, as such, (re-)construct values of the type at issue. On the other hand, there was the research on overloaded functions as used in programming languages that 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 $\lambda\&$-calculus [Castagna et al. 1995], overloaded functions have types that are characterized 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 Frisch et al. [2002] 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 (parametric code) with pieces of code that are specific to only some particular input types (*ad hoc* code). Therefore, they nicely integrate both programming styles.

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 Castagna et al. [2005, 2007] for the π-calculus.

1.1. PLAN OF THE ARTICLE. The presentation is structured in four parts:

(1) In the first part (Section 2), we lengthy discuss the main ideas, the underlying intuitions, and the logical entailment of the whole approach.
(2) In the second part (Sections 3–5), we succinctly and precisely define the system: the calculus and its typing relation (Section 3), the subtyping relation (Section 4), and their properties (Section 5).
(3) The third part (Section 6) presents the technical details of the properties stated in the second part. It can be skipped in the first reading.

[1]As a universally quantified second-order type can be interpreted as a mapping from types to types, so a finite intersection of arrow types can be seen as point-wise definition of a finite mapping from types to types. This is just an intuitive analogy: this particular use of intersection types evokes the perfume of parametricity but must not be taken *strictu senso*.

(4) The last part (Sections 7–9) explains those intuitions and details that could not be given in the first part since their explanation required the technical development (Section 7), it discusses related work (Section 8), and hints to other work based on the material presented here together with some results that confirm the robustness of our approach (Section 9).

2. *Overview of the Approach*

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 co-inductive, 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 and possibly a *subsumption* rule that accounts for subtyping. 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 deduction 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. A Five Steps Recipe.

In principle, the process of defining semantic subtyping can be roughly summarized in the following five steps:

(1) Take a bunch of type *constructors* (e.g., $\rightarrow$, $\times$, $\mathtt{ch}$, ...) and extend the type algebra with the following *Boolean combinators*: union $\vee$, intersection $\wedge$, and negation $\neg$, yielding a type algebra $\mathcal{T}$.

(2) Give a *set-theoretic model* of the type algebra, namely define a function $[\![\ ]\!]_D : \mathcal{T} \rightarrow \mathcal{P}(D)$, for some domain D (where $\mathcal{P}(D)$ denotes the power-set of D). In such a model, the combinators must be interpreted in a set-theoretic way (i.e., $[\![s \wedge t]\!]_D = [\![s]\!]_D \cap [\![t]\!]_D$, $[\![s \vee t]\!]_D = [\![s]\!]_D \cup [\![t]\!]_D$, and $[\![\neg t]\!]_D = D \setminus [\![t]\!]_D$), and the definition of the model must capture the essence of the type constructors. There might be several models, and each of them induces a specific subtyping relation on the type algebra. We only need to prove that there exists at least one model and then pick one that we call the *bootstrap model*. If its associated interpretation function is $[\![\ ]\!]_{\mathcal{B}}$, then it induces the following subtyping relation:

$$s \leq_{\mathcal{B}} t \quad \stackrel{def}{\Longleftrightarrow} \quad [\![s]\!]_{\mathcal{B}} \subseteq [\![t]\!]_{\mathcal{B}} \tag{1}$$

(3) Now that we defined a subtyping relation for our types, find a subtyping algorithm that decides (or semi-decides) the relation. This step is not mandatory but highly advisable if we want to use our types in practice.

(4) Now that we have a (hopefully) suitable subtyping relation available, we can focus on the language itself, consider its typing rules, use the new subtyping relation to type the terms of the language, and deduce $\Gamma \vdash_{\mathcal{B}} e : t$. In particular,

this means to use in the subsumption rule the bootstrap subtyping relation $\leq_{\mathcal{B}}$ we defined in Step 2.

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

$$s \leq_{\mathcal{B}} t \quad \Longleftrightarrow \quad s \leq_{\mathcal{V}} t$$

and this closes the circularity. Of course, this does not imply that the definitions are "valid" in any formal sense, only that they are mutually coherent. We still need to check type soundness. In this article, we do this with standard syntactical techniques (subject reduction and progress).

While the five steps above outline a nice framework in which to fit and understand what follows, in practice, however, the starting point never is the model of types but the calculus: in particular one always starts from the calculus and its values, and tries to slightly modify these so that the values outline some model that can then be formalised. This is what we also do here: while we follow the five-steps process above to give, in the rest of this section, an overview of the approach, in Section 3, we introduce a λ-calculus with overloaded functions and dynamic dispatch, in Section 4, we introduce a model to semantically define a subtyping relation inspired from the previous calculus, and in Section 5 discuss the main results, namely, the soundness of the typing relation, the correspondence between the values of Section 3 and the model of Section 4, and the decidability of the various relations.

2.2. 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 the system to prove it. This argument is particularly important with a rich-type algebra, where type constructors interact in nontrivial ways; for instance, when considering arrow, intersection and union types, one must take into account—that is, introduce rules for—many distributivity relations such as, for instance,[2] $(t_1 \vee t_2) \to s \simeq (t_1 \to s) \wedge (t_2 \to s)$. Forgetting any of these rules yields a type system that, although sound, does not match (i.e., it is not complete with respect to) the intuitive semantics of types.

[2]We write $s \simeq t$ as a shorthand for $s \leq t$ *and* $s \geq t$.

(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-known Boolean algebra laws to decompose subtyping on simpler types (as we show in Section 6.2). Furthermore, as most of the formal effort is done with the semantic definition of subtyping, studying variations of the algorithm (e.g., optimizations or different rules) turns out to be much simpler (this is common practice 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. Other examples of properties that come easily from a semantic definition are the variance of type constructors, and distributivity laws (e.g., $t_1 \times (t_2 \vee t_3) \simeq (t_1 \times t_2) \vee (t_1 \times t_3)$).

Although these properties look 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 given independently from the typing relation; also, usually the model corresponds to an untyped denotational semantics, where types are interpreted by structures in which negative types either do not have an immediate interpretation (e.g., the complement of ideals is not an ideal, therefore one should consider to mix ideals with co-ideals), or simply are never considered (one of the JACM reviewers suggests that this may be for "ideological reasons coming from proof theory and intuitionism" rather than for technical problems). 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 relation 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 at a subtyping relation (and eventually verify that this relation is consistent with our language). Thus, in our approach, types have a strong substance even before introducing the typing relation.

2.3. 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 [Asperti and Longo 1991]—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 same 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.[3]

2.4. 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[4] 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 weak head-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.

[3]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.

[4]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).

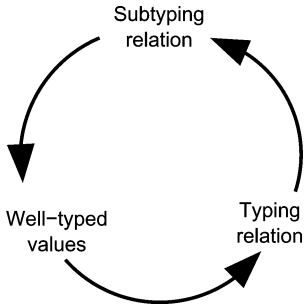

FIG. 1. Circularity.

2.5. 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 inhabiting type constructors such as products or records can be inductively defined from basic types without resorting to any typing relation (this is why the XDuce 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.

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 do so. 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; but, 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".

2.6. SET-THEORETIC MODELS. Let us then show with more details how we shall proceed. We do not need to define a particular language, the definition of types will suffice. Here, we assume that types are defined by the following syntax:

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

where $\mathbb{0}$ and $\mathbb{1}$ respectively correspond to the empty and universal types (these are sometimes denoted by the pair $\bot$, $\top$ or **Bottom**, **Top**). The formal definition of

the type algebra, which includes recursive types and basic types, will be given in Section 3.1.

The second step is to define precisely what a *set-theoretic* model for these types is. As Hindley and Longo [1980] give some general conditions that characterize 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 $\mathcal{T}$ be the set of types, D some set, and $[\![\_]\!]$ an interpretation function from $\mathcal{T}$ to $\mathcal{P}(D)$. The conditions that $[\![\_]\!]$ must satisfy to define a set-theoretic model are mostly straightforward, namely:

(1) $[\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$
(2) $[\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$
(3) $[\![\neg t]\!] = D \setminus [\![t]\!]$
(4) $[\![\mathbb{1}]\!] = D$
(5) $[\![\mathbb{0}]\!] = \varnothing$
(6) $[\![t \times s]\!] = [\![t]\!] \times [\![s]\!]$
(7*) $[\![t \to s]\!] =$???.

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 D but we prefer not to enter in such a level of detail at this point of our presentation). But the definition is not complete yet as we still have to establish the seventh condition (highlighted by a $*$) that constrains 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 condition strongly depends on the language and in particular on the kind of functions we want to implement in our language. We give detailed examples of this in Frisch [2004]. 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 D, then $[\![t \to s]\!]$ is the set of binary relations in which if the first projection is in (the interpretation of) t, then the second projection is in s, namely $\{ f \subseteq D^2 \mid \forall (d_1, d_2) \in f.\, d_1 \in [\![t]\!] \Rightarrow d_2 \in [\![s]\!] \}$. Note that this set can also be written $\mathcal{P}([\![t]\!] \times \overline{[\![s]\!]})$, where the overline denotes set complement (with respect to D or D^2). If the language is expressive enough, we can do as if every binary relation in this set was an element of $[\![t \to s]\!]$; thus, we would like to say that the seventh condition is:

$$[\![t \to s]\!] = \mathcal{P}(\overline{[\![t]\!] \times \overline{[\![s]\!]}}). \tag{2}$$

But this is completely meaningless. First, technically, this would imply that $\mathcal{P}(D^2) \subseteq D$, 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 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 the 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 $\mathcal{P}(\overline{[\![t]\!] \times \overline{[\![s]\!]}})$ as stated by (2), for what it concerns the goal we are aiming at, it is enough to require that a model must interpret functional types so that the induced subtyping relation is the same as the one the condition (2) would induce, that is:

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

and similarly for any Boolean combination of arrow types.

Formally, we associate (see Definition 4.3 in Section 4.2) to $[\![\_]\!]$ an extensional interpretation $\mathbb{E}(\_)$ that behaves as $[\![\_]\!]$ except for arrow types, for which we use the condition above as definition:

$$\mathbb{E}(t\to s) = \mathcal{P}(\overline{[\![t]\!] \times \overline{[\![s]\!]}})$$

Note that we use $[\![\_]\!]$ in the right-hand side of this equation, that is, we only re-interpret top-level arrow types. Now we can express the fact that $[\![\_]\!]$ 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) $[\![t]\!] = \varnothing \iff \mathbb{E}(t) = \varnothing$

or, equivalently, $[\![t_1]\!] \subseteq [\![t_2]\!] \iff \mathbb{E}(t_1) \subseteq \mathbb{E}(t_2)$.[5]

To put it otherwise, if we wanted an interpretation $[\![\_]\!]$ of the types that were faithful with respect to the semantics of the language, then we should require for all t that $[\![t]\!] = \mathbb{E}(t)$. 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 $[\![\_]\!]$ since all we ask to $[\![\_]\!]$ is to characterize the *containment* of types, and to that end it suffices to characterize the zeros of $[\![\_]\!]$, since

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

Therefore, instead of asking that $[\![\_]\!]$ and $\mathbb{E}(\_)$ coincide on all points, we require a weaker constraint, namely that they have the same zeros:

$$[\![t]\!] = \varnothing \iff \mathbb{E}(t) = \varnothing.$$

This is the essence of our definition of *models* of the type algebra (Definition 4.4 in Section 4.2).

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

[5]Indeed, $[\![t_1]\!] \subseteq [\![t_2]\!] \iff [\![t_1]\!] \setminus [\![t_2]\!] = \varnothing \iff [\![t_1\wedge\neg t_2]\!] = \varnothing \iff \mathbb{E}(t_1\wedge\neg t_2) = \varnothing \iff \mathbb{E}(t_1) \setminus \mathbb{E}(t_2) = \varnothing \iff \mathbb{E}(t_1) \subseteq \mathbb{E}(t_2)$.

for. Previous work [Frisch 2004] shows 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) *Nondeterministic*. 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) *Nonterminating*. 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 $t \to \mathbb{0}$; now, all the functions in such types must be always non-terminating on their domain (if they returned a value this would inhabit $\mathbb{0}$).
(3) *Overloaded*. Here, by overloading, we mean functions that can be applied to many different types, and whose results' type can depend on the type of the argument.[6] 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 $[\![(t_1 \to s_1) \wedge (t_2 \to s_2)]\!]$, (the equality instead holds in λ-calculus with union and intersection types [Barbanera et al. 1995]), 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. Then it may not be 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 returning 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 $(t_1 \vee t_2) \to (s_1 \wedge s_2)$ (this point is discussed thoroughly in Section 4.5 of our related survey [Castagna 2005]).

2.7. 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 $(D, [\![\_]\!])$ that satisfies the conditions of set-theoretic model, and choose any such pair, no matter which one. Let us call this model the *bootstrap model*. This bootstrap model defines a particular subtyping relation on our set of types $\mathcal{T}$:

$$s \leq t \quad \Longleftrightarrow \quad [\![s]\!] \subseteq [\![t]\!].$$

We can then pick any language that uses the types in $\mathcal{T}$ (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 $\leq$ we have just defined. We write $\Gamma \vdash e : t$ for the typing judgment of the language. In this article, we will consider a λ-calculus with overloaded functions and dynamic type-dispatch. See Section 3.1 for the syntax of the calculus, Section 3.3 for its

[6]This use of the term "overloading" is pretty wide since it includes for instance polymorphic functions. In this discussion, a nonoverloaded function should be thought as a function that comes with explicit input and output types.

type system and Section 3.2 for its semantics (which depends on the type system because of the dynamic type-dispatch construction).

2.8. CLOSING THE CIRCLE. In order to obtain type-safety for our calculus, we want the type system to enjoy properties such as subject reduction (Theorem 5.1) and progress (Theorem 5.2) stated in Section 5.1. Because of the subsumption rule in the type system, this can only be obtained if our definition of set-theoretic models is meaningful with respect to the operational semantics of our calculus. This is a first sanity-check for our notion of model.

But, once type-safety has been established, there is another important question: what are the relations between the bootstrap model and the calculus? And in particular, what is the relation between the bootstrap model and the values of the calculus? Have we lost all the intuition underlying the "types as sets of values" interpretation?

To answer these questions, we consider a new interpretation of types as sets of values in the calculus:

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

A second sanity-check for our notion of model is then to require that this interpretation $[\![\_]\!]_{\mathcal{V}}$ is a model. If this is the case, we can use it to define a new subtyping relation on $\mathcal{T}$:

$$s \leq_{\mathcal{V}} t \quad \Longleftrightarrow \quad [\![s]\!]_{\mathcal{V}} \subseteq [\![t]\!]_{\mathcal{V}}.$$

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 (Theorem 5.5 in Section 5.2):

$$s \leq t \qquad \Longleftrightarrow \qquad s \leq_{\mathcal{V}} t.$$

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. *The Calculus*

In this section, we define formally the syntax of types and expressions in our calculus (Section 3.1), the operational semantics (Section 3.2) and the type system (Section 3.3). A type-checking algorithm will be presented in Section 6.12.

The semantics actually depends on the type-system, which in turn depends on a subtyping relation to be defined (next section). As a consequence, we consider here the subtyping relation as a parameter of the definitions of the type system and of the semantics.

3.1. SYNTAX

3.1.1. *Expressions.* To define the calculus, we choose a set of constants $\mathcal{C}$ ranged by the meta-variable c (they will be elements of basic types).

The terms of the calculus are called ***expressions*** and are defined by the following grammar.

e	$::=$	c	constant
	$\mid$	x	variable
	$\mid$	(e, e)	pair
	$\mid$	$\pi_i(e)$	projection ($i \in \{1, 2\}$)
	$\mid$	$\mu f(t{\to}t; \ldots ; t{\to}t).\lambda x.e$	abstraction
	$\mid$	$e\ e$	application
	$\mid$	$(x = e \in t\ ?\ e \mid e)$	dynamic type dispatch
	$\mid$	$\mathtt{rnd}(t)$	nondeterministic choice,

where t ranges over types, defined in the next paragraph.

We write $\mathcal{E}$ for the set of expressions. The syntax for the calculus deserves few comments. We introduce an explicit construction for recursive functions, which combines λ-abstraction and a fix-point operator. The reason is that we want to express non-terminating expressions, but still restrict recursion to functions only. The identifiers f and x act as binders in the body of the function. The λ-abstraction comes with an nonempty sequence of function types (we call it the *interface* of the function): if more than one type is given, we are in the presence of an overloaded function. As we will see later in the type system, we adopt prescriptive Church-style for λ-abstractions: the types assigned to such expressions can be read from their signature, without considering their body. The reason, besides making type-checking feasible, is that because (closed and well-typed) λ-abstractions are values, they can be subject to dynamic type dispatch and we do not want to rely on the whole type system to decide whether a λ-abstraction has some type or not.

The nondeterministic choice construction $\mathtt{rnd}(t)$ picks an arbitrary expression of type t. We introduced this operator in the calculus in order to demonstrate subtle typing issues coming from nondeterminism. This operator can be used to model internal or external nondeterminism such as inputs or side effects.

The only data constructor in the calculus is the pair. General tuples and tagged values can be encoded by nested pairs and constants. Similarly, Section A.1 of Appendix A.1 shows how to encode disjoint sums with pairs and constants.

3.1.2. *Types.* Types are essentially those introduced in Section 2.6 (modulo Boolean equivalence) to which we add basic types (the types of constant expressions). In order to simplify the presentation of recursive types, we are going to consider potentially *infinite regular* terms produced by the following signature:

t	$::=$	b	basic type
	$\mid$	$t \times t$	product type
	$\mid$	$t \to t$	function type
	$\mid$	$t \vee t$	union type
	$\mid$	$\neg t$	complement type
	$\mid$	$\mathbb{0}$	empty type.

By regular, we mean that terms have only but a finite number of different subterms. The meta-variable b ranges over a fixed set of ***basic types***. We write $t_1 \setminus t_2$ as an abbreviation for $t_1 \wedge \neg t_2$, $t_1 \wedge t_2$ as an abbreviation for $\neg(\neg t_1 \vee \neg t_2)$, and $\mathbb{1}$ as an abbreviation for $\neg\mathbb{0}$. We will call *atom* the immediate applications of type constructors: basic types, product types, function types (these are the "atoms" for

Boolean combinators). Since we want types to denote sets, we need to impose some constraints to avoid ill-formed types such as a solution to $t = t \vee t$ (which does not carry any information about the set denoted by the type) or to $t = \neg t$ (which cannot represent any set). Namely, we say that a term is a ***type*** if it does not contain any infinite branch without an atom. Let us call $\mathcal{T}$ the set of types.

The conditions above say that the binary relation $\triangleright \subseteq \mathcal{T}^2$ defined by $t_1 \vee t_2 \triangleright t_i$, $\neg t \triangleright t$ is Noetherian (i.e., strongly normalizing). This gives an induction principle on $\mathcal{T}$ that we will use without any further explicit reference to the relation $\triangleright$.

3.2. OPERATIONAL SEMANTICS. Because of the dynamic-type dispatch, the semantics of the calculus depends on its type system. For now, we simply assume that a relation between expressions and types, written $\vdash e : t$ is given. It will be defined in the next section.

Definition 3.1. An expression e is a ***value*** if it is closed (no free variable), well-typed ($\vdash e : t$ for some type t), and produced by the following grammar:

$$v ::= c \mid (v, v) \mid \mu f(\cdots).\lambda x.e$$

We write $\mathcal{V}$ for the set of all values.

We define a small-step operational call-by-value semantics $\rightsquigarrow$ for the calculus. There are four basic reduction rules (we write $e[x_1 := e_1; x_2 := e_2; \cdots]$ for the expression obtained from e by a capture-avoiding simultaneous substitution of x_i by e_i):

$$\begin{array}{l} ev \rightsquigarrow e[f := e'; x := v] \quad \text{if } e = \mu f(\cdots).\lambda x.e' \\ (x = v \in t \ ? \ e_1 | e_2) \rightsquigarrow \begin{cases} e_1[x := v] & \text{if } \vdash v : t \\ e_2[x := v] & \text{if } \vdash v : \neg t \end{cases} \\ \pi_i(v_1, v_2) \rightsquigarrow v_i \\ \texttt{rnd}(t) \rightsquigarrow e \quad \text{if } \vdash e : t \end{array}$$

The relation $\rightsquigarrow$ is further extended by an inductive context rule:

$$C[e] \rightsquigarrow C[e'] \quad \text{if } e \rightsquigarrow e',$$

where the notion of (immediate) context is defined by:

$$\begin{array}{rcl} C[] & ::= & ([], e) \mid (e, []) \\ & \mid & []e \mid e[] \\ & \mid & (x = [] \in t \ ? \ e | e) \mid (x = e \in t \ ? \ [] | e) \mid (x = e \in t \ ? \ e | []) \\ & \mid & \pi_i([]) \\ & \mid & \mu f(\cdots).\lambda x.[]. \end{array}$$

As usual, a type safety result will be obtained by a combination of two lemmas: subject reduction (or type preservation) and progress (closed and well-typed expressions that are not values can be reduced).

The reduction rule for application requires the argument to be a value (call-by-value). In order to understand why, let us consider the application $(\mu f(t \to t \times t; s \to s \times s).\lambda x.(x, x))(\texttt{rnd}(t \vee s))$. The type system will assign to the abstraction the type $(t \to t \times t) \wedge (s \to s \times s)$. A set-theoretic reasoning shows that this type is a subtype of $(t \vee s) \to ((t \times t) \vee (s \times s))$. The type system also assigns to the argument $\texttt{rnd}(t \vee s)$ the type $t \vee s$. It will thus also assign the type $(t \times t) \vee (s \times s)$ to the application. If the semantics permits to reduce this application, we would get as a result the expression $(\texttt{rnd}(t \vee s), \texttt{rnd}(t \vee s))$ whose most precise static type is

$(t\vee s)\times(t\vee s)$. Clearly, this type is (in general) a strict supertype of $(t\times t)\vee(s\times s)$. So, if the semantics does not force the argument to be a value in order to reduce an application, we could not obtain the subject reduction lemma.

Similarly, the reduction rule for projection requires its argument to be a value. To understand why, consider the expression $e = \pi_1(e_1, e_2)$ where e_1 is an expression of type t_1 and e_2 is a looping expression of type $\mathbb{0}$ (e.g., $(\mu f(\mathbb{1}\to\mathbb{0}).\lambda x.\, f\, x)c$). The type system will assign the type $t_1\times\mathbb{0}$ to e, but in our system $t_1\times\mathbb{0}$ is an empty type because, intuitively, a set-theoretic Cartesian product with an empty component is itself empty. If e could be reduced to e_1, it would be a violation of type preservation.

The same argument applies to the dynamic type dispatch. If we allowed to reduce $(x = e \in t\ ?\ e_1|e_2)$ to $e_1[x := e]$ when $\vdash e : t$, even if e is not a value, we could break type preservation. Consider for instance the case where $\vdash e : \mathbb{0}$. In this case, the type system does not check anything about the branches e_1 and e_2 (the reason for this is explained in details later on) and so e_1 could be ill-typed. Note that when e is a value, then the dynamic type dispatch can always be reduced. Indeed, because our type connectives will be interpreted in a set-theoretic way, we always have $\vdash v : t$ or $\vdash v : \neg t$ (for any value v and any type t).

3.3. TYPE SYSTEM. The semantics we just introduced depends on the typing judgment $\Gamma \vdash e : t$ where Γ is a finite mapping from variables to types (we write $\vdash e : t$ when Γ is empty). This judgment, in turn, depends on a subtyping relation $\leq$ between types that we are going to introduce later on. For now, we assume it is a parameter of the type system.

For each constant c, we assume given a basic type b_c. The rules are:

$$\frac{\Gamma\vdash e : t_1 \quad t_1 \leq t_2}{\Gamma\vdash e : t_2}\ (subsum) \qquad \frac{}{\Gamma\vdash c : b_c}\ (const) \qquad \frac{}{\Gamma\vdash x : \Gamma(x)}\ (var)$$

$$\frac{\Gamma\vdash e : t_1\times t_2}{\Gamma\vdash \pi_i(e) : t_i}\ (proj) \qquad \frac{\Gamma\vdash e_1 : t_1\to t_2 \quad \Gamma\vdash e_2 : t_1}{\Gamma\vdash e_1 e_2 : t_2}\ (appl)$$

$$\frac{\begin{array}{l} t = \bigwedge_{i=1..n}(t_i\to s_i)\wedge \bigwedge_{j=1..m}\neg(t'_j\to s'_j) \quad t\not\simeq\mathbb{0} \\ \forall i = 1\cdot\cdot n.\Gamma, (f : t), (x : t_i)\vdash e : s_i \end{array}}{\Gamma\vdash \mu f(t_1\to s_1;\cdots;t_n\to s_n).\lambda x.e : t}\ (abstr)$$

$$\frac{\Gamma\vdash e : t_0 \qquad \begin{cases} t_0\not\leq\neg t \ \Rightarrow\ \Gamma, (x : t_0\wedge t)\vdash e_1 : s \\ t_0\not\leq t \ \Rightarrow\ \Gamma, (x : t_0\backslash t)\vdash e_2 : s \end{cases}}{\Gamma\vdash (x = e\in t\ ?\ e_1|e_2) : s}\ (case)$$

The rule (*subsum*) causes the type system to depend on the subtyping relation to be defined. The rules (*const*), (*pair*), (*var*), (*proj*), (*rnd*), and (*appl*) are standard or straightforward.

The rule (*abstr*) is a little bit tricky. Each arrow type $t_i\to s_i$ in the function interface is interpreted as a constraint to be checked. The body of the abstraction is thus type-checked once for each such constraint. When considering the type $t_i\to s_i$, the variable x is assumed to have type t_i and the body is checked against type s_i. Also, the variable f is assumed to have type t, which is also the type given to the whole function. Quite intuitively, this type is obtained by taking the

intersection of all the types $t_i \to s_i$. But we also add to this intersection any finite number of complement of arrow types, provided the type t does not become empty. This might sound surprising, but the reason is actually simple: we want types to be interpreted as sets of values in such a way that Boolean connectives behave as their set-theoretic counterpart. In particular, the union of t and $\neg t$ must always be equivalent to $\mathbb{1}$, that is, we need to have the following property: $\forall v.\forall t.(\vdash v : t)$ or $(\vdash v : \neg t)$. In particular, since a (closed and well-typed) abstraction is value, it must have type $(t \to s)$ or type $\neg(t \to s)$ for any choice of t and s. If $(t \to s)$ is a supertype of the intersection $\bigwedge t_i \to s_i$, the abstraction is known, thanks to the subsumption rule, to have type $(t \to s)$. Otherwise, the abstraction must have type $\neg(t \to s)$, but since we cannot use subsumption to prove it, then we need to provide a way to prove it has type $\neg(t \to s)$. This is why we introduce such complements of arrow types in the rule (*abstr*). More comments about this rule can be found in Section 7.3.

The rule (*case*) is easier to read. First, we need to find a type t_0 for the expression whose result will be dynamically type-checked. If this type has a nonempty intersection with t ($t_0 \not\leq \neg t$), then the first branch might be used. In this case, in order for the whole expression to have type s, we need to check that e_1 has also type s, assuming that x has type $t \wedge t_0$. Indeed, at runtime, the variable x will be bound to a value resulting from the evaluation of e_0. Because of subject reduction, this value is necessarily of type t_0. But in order to type-check e_1, we can also assume that the value has type t. If $t_0 \leq \neg t$, then the first branch cannot be used, and we don't need to type-check e_1. Similarly for e_2, replacing t with $\neg t$. The ability to ignore e_1 and/or e_2 when computing the type for $(e \in t \ ?\ e_1 \mid e_2)$ is important to type-check overloaded function. As an example, consider the abstraction $\mu f(b_1 \to b_1; b_2 \to b_2).\lambda x.(x \in b_1 \ ?\ c_1 \mid c_2)$ where b_1 and b_2 are two nonintersecting basic types and c_1 (respectively, c_2) is a constant of type b_1 (respectively, b_2). The rule (*abstr*), when it considers the arrow type $b_1 \to b_1$, checks that the body has type b_1 assuming that x has type b_1. Clearly, the typing rule for the dynamic type dispatch must discard in this case the type of the second branch.

As an aside note that the use of the *ex falso quodlibet* rule ($\bot$) yields a simpler formulation of the *case* rule:

$$\frac{}{\Gamma, x : \mathbb{0} \vdash e : t}\ (\bot) \qquad \frac{\Gamma \vdash e : t_0 \quad \Gamma, (x : t_0 \wedge t) \vdash e_1 : s \quad \Gamma, (x : t_0 \setminus t) \vdash e_2 : s}{\Gamma \vdash (x = e \in t \ ?\ e_1 | e_2) : s}\ (case).$$

The reason why we preferred the previous formulation is that it permits a stronger and simpler substitution lemma. A second reason to prefer the previous formulation is that simpler (*case*) rule above does not easily extend to the full version of $\mathbb{C}$Duce with general pattern matching, since it would need special treatment for patterns without any capture variable (since these would not produce any $x : \mathbb{0}$ hypothesis in the environment).

4. *Subtyping*

At this point, we have given the calculus, an operational semantics that depends on its type system, which, in turn, depends on a subtyping relation still to be defined. The last missing step to complete the definition of our system is the subtyping

relation. This will be defined by formalizing the ideas we outlined in Sections 2.6–2.8.

4.1. Set-Theoretic Interpretations of Types

Definition 4.1. A ***set-theoretic interpretation*** of $\mathcal{T}$ is given by a set D and a function $[\![\_]\!] : \mathcal{T} \to \mathcal{P}(D)$ such that, for any types t_1, t_2, t:

— $[\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$
— $[\![\neg t]\!] = D \backslash [\![t]\!]$
— $[\![\mathbb{0}]\!] = \varnothing$

(A consequence of the conditions is that $[\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$, $[\![t_1 \backslash t_2]\!] = [\![t_1]\!] \backslash [\![t_2]\!]$, and $[\![\mathbb{1}]\!] = D$.)

This definition does not say anything about the interpretation of atoms. Actually, using an induction on types, we see that set-theoretic interpretations with domain D correspond univocally to functions from atoms to $\mathcal{P}(D)$.

A set-theoretic interpretation $[\![\_]\!] : \mathcal{T} \to \mathcal{P}(D)$ induces a binary relation $\leq_{[\![]\!]} \subseteq \mathcal{T}^2$ defined by:

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

This relation actually only depends on the set of empty types. Indeed, we have: $[\![t_1]\!] \subseteq [\![t_2]\!] \iff [\![t_1]\!] \cap (D \backslash [\![t_2]\!]) = \varnothing \iff [\![t_1 \wedge \neg t_2]\!] = \varnothing$. We also get properties of the relation $\leq_{[\![]\!]}$ ≪for free≫, such as its transitivity, or the monotonicity of the $\vee$ and $\wedge$ constructors, and so on.

4.2. Models of Types.

We are going to define a notion of model of the type algebra. Intuitively, a model is a set-theoretic interpretation such that type constructors are interpreted in such as way that the induced relation $\leq_{[\![]\!]}$ capture their essence (in the type system of the calculus), at least as far as subtyping is concerned.

As we explained in Section 2.6, the way to formalize it consists in associating to the interpretation $[\![\_]\!]$ another interpretation $\mathbb{E}(\_)$, called extensional, and then to require, for $[\![\_]\!]$ to be a model, that $[\![\_]\!]$ and $\mathbb{E}(\_)$ behave the same as long as subtyping is concerned (that is: $[\![t]\!] \subseteq [\![s]\!] \iff \mathbb{E}(t) \subseteq \mathbb{E}(s)$ or, equivalently, $[\![t]\!] = \varnothing \iff \mathbb{E}(t) = \varnothing$).

For each basic type b, we assume there is a fixed set of constants $\mathbb{B}[\![b]\!] \subseteq \mathcal{C}$ whose elements are called constants of type b. Note that for two basic types b_1, b_2, the sets $\mathbb{B}[\![b_i]\!]$ can have a nonempty intersection. For any constant c, we assume that the type b_c is a singleton: $\mathbb{B}[\![b_c]\!] = \{c\}$.

A product type $t_1 \times t_2$ will of course be interpreted extensionally as the Cartesian product $[\![t_1]\!] \times [\![t_2]\!]$.

Things are more complicated for a function type $t_1 \to t_2$. Its extensional interpretation should be the set of set-theoretic functions (i.e., functional graphs) f such that $\forall d.\ d \in [\![t_1]\!] \Rightarrow f(d) \in [\![t_2]\!]$. However, the calculus we have in mind can express nonterminating and/or nondeterministic functions as well. This suggests to consider arbitrary binary relations instead of just functional graphs. Also, the calculus has a notion of type error: it is not possible to apply an arbitrary function to an arbitrary value. We are going to take Ω as a special element to denote this type error. Following this discussion, we interpret the function type $t_1 \to t_2$ as the set of binary relations $f \subseteq D \times D_\Omega$ (where $D_\Omega = D + \{\Omega\}$) such that $\forall (d, d') \in f.\ d \in [\![t_1]\!] \Rightarrow d' \in [\![t_2]\!]$.

Definition 4.2. If D is a set and X, Y are subsets of D, we write D_Ω for $D+\{\Omega\}$ and define $X \to Y$ as:

$$X \to Y = \{f \subseteq D \times D_\Omega \mid \forall (d, d') \in f.\, d \in X \Rightarrow d' \in Y\}.$$

Note that if we replace D_Ω with D in this definition, then $X \to Y$ is always a subset of $D \to D$. As we will see shortly, this would imply that any arrow type is a subtype of $\mathbb{1}\to\mathbb{1}$. Thanks to the subsumption rule, the application of any well-typed function to any well typed argument would then be itself well-typed. Clearly, this would break type-safety of the calculus. With Definition 4.2, instead, we have $X \to Y \subseteq D \to D$ if and only if $D = X$.

We can now give the formal definition of the extensional interpretation associated to a set-theoretic interpretation.

Definition 4.3. Let $[\![\_]\!] : \mathcal{T} \to \mathcal{P}(D)$ be a set-theoretic interpretation. We define its associated ***extensional interpretation*** as the unique set-theoretic interpretation $\mathbb{E}(\_) : \mathcal{T} \to \mathcal{P}(\mathbb{E}D)$ (where $\mathbb{E}D = \mathcal{C} + D^2 + \mathcal{P}(D \times D_\Omega)$) such that:

$$\begin{array}{lll} \mathbb{E}(b) & = \mathbb{B}[\![b]\!] & \subseteq \mathcal{C} \\ \mathbb{E}(t_1 \times t_2) & = [\![t_1]\!] \times [\![t_2]\!] & \subseteq D^2 \\ \mathbb{E}(t_1 \to t_2) & = [\![t_1]\!] \to [\![t_2]\!] & \subseteq \mathcal{P}(D \times D_\Omega) \end{array}$$

Finally, we can formalize the fact that a set-theoretic interpretation induces the same subtyping relation as if the type constructors were interpreted in an extensional way.

Definition 4.4. A set-theoretic interpretation $[\![\_]\!] : \mathcal{T} \to \mathcal{P}(D)$ is a ***model*** if it induces the same subtyping relation as its associated extensional interpretation:

$$\forall t_1, t_2 \in \mathcal{T}.\, [\![t_1]\!] \subseteq [\![t_2]\!] \iff \mathbb{E}(t_1) \subseteq \mathbb{E}(t_2).$$

Thanks to a remark in Section 4.1, the condition for a set-theoretic interpretation to be a model can be reduced to:

$$\forall t \in \mathcal{T}.\, [\![t]\!] = \varnothing \iff \mathbb{E}(t) = \varnothing.$$

At this point, we can derive many properties about $\leq_{[\![\_]\!]}$, which directly follow from the fact that it is induced by a model. For instance, the co-/contra-variance of the arrow type constructor, and equivalences such as $(t_1 \to s) \wedge (t_2 \to s) \simeq (t_1 \vee t_2) \to s$, can be immediately derived from the definition of the extensional interpretation. The meta-theoretic study of the system relies in a crucial way on many of such properties. With a more axiomatic approach for defining the subtyping relation, for example, by a system of inductive or coinductive rules, we would probably need much more work to establish these properties, and we would not have the same level of trust that we did not forget any rule.

4.3. WELL-FOUNDEDNESS. The notion of model captures the intended local behavior of type constructors with respect to subtyping. However, it fails to capture a global property of the calculus, namely that values are *finite* binary trees (where leaves are either constants or abstractions). For instance, let us consider the recursive type $t = t \times t$. Intuitively, a value v has this type if and only if it is a pair (v_1, v_2) where v_1 and v_2 also have type t. To build such a value, we would need to consider an infinite tree, which is ruled out. As a consequence, the type t contains no value.

We will introduce a new criterion to capture this property of finite decomposition of pairs.

Definition 4.5. A set-theoretic interpretation $[\![\_]\!] : \mathcal{T} \to \mathcal{P}(D)$ is ***structural*** if:

—$D^2 \subseteq D$;
—for any types t_1,t_2: $[\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!]$;
—the binary relation on D induced by $(d_1, d_2) \triangleright d_i$ is Noetherian.

Definition 4.6. A model $[\![\_]\!] : \mathcal{T} \to \mathcal{P}(D)$ is ***well-founded*** if it induces the same subtyping relation as a structural set-theoretic interpretation.

5. *Main Results*

Let us fix an arbitrary model $[\![\_]\!] : \mathcal{T} \to \mathcal{P}(D)$, which we call the ***bootstrap model***. It induces a subtyping relation, which we simply write $\leq$. In turn, this subtyping relation defines a typing judgment $\Gamma \vdash e : t$ for the calculus and thus also a notion of value and a reduction relation $e \rightsquigarrow e'$. We can now state four groups of theoretical results about our system. This first group (Section 5.1) expresses the fact that our notion of models implies that the type system and the semantics are mutually coherent. The second group (Section 5.2) justifies our approach for defining the subtyping relation with a detour through the notion of models: indeed, we can in fine re-interpret types as sets of values, and this creates a new model equivalent to the bootstrap model (if it is well founded). The third group of results (Section 5.3) shows that the notion of model is not void, by expressing the existence of (several different) models satisfying the various conditions. Finally, we focus (Section 5.4) on the effectiveness of the subtyping and typing relations and devise simple subtyping algorithms.

5.1. Type Soundness.

As announced earlier, we have the two classical lemmas that entail type soundness (proofs in Section 6.6).

Theorem 5.1 (Subject Reduction). *Let e be an expression and t a type. If $(\Gamma \vdash e : t)$ and $(e \rightsquigarrow e')$, then $(\Gamma \vdash e' : t)$.*

Theorem 5.2 (Progress). *Let e be a well-typed closed expression. If e is not a value, then there exists an expression e' such that $e \rightsquigarrow e'$.*

It is worth noticing that the proof of Theorem 5.2 (given in Section 6.6) does not use reductions under abstractions or inside the branches of dynamic type dispatch, therefore Progress still holds if we disallow such reductions. Of course, subject reduction also holds in that case. This means that a weak reduction strategy (as implemented typically in programming languages) enjoys type soundness, too. In the setting of programming languages, proving the subject reduction property also for a semantics that includes strong reduction rules is useful because these rules correspond to possible compile-time optimizations.

Theorem 5.3. *For every types t and t_1 such that $t \leq t_1 \to \mathbb{1}$, there exists a type t_2 such that, for every value v:*

$$\vdash v : t_2 \iff \exists v_f, v_x.\ (v_f\, v_x \overset{\star}{\rightsquigarrow} v) \wedge (\vdash v_f : t) \wedge (\vdash v_x : t_1)$$

This type is the smallest solution to the equation $t \leq t_1 \to s$.

This result is proved in Section 6.11. The type t_2 in the statement of the theorem above represents exactly all the possible results (i.e., is the set of all values that) we may get when applying a closed expression e_1 of type t to a closed expression e_2 of type t_1. Since $t \leq t_1 \to t_2$, the type system allows us to derive type t_2 for the application $e_1 e_2$. In other words, the typing rule (*appl*) is *locally exact*: it does not introduce any new approximation to those already made when typing its arguments.

5.2. CLOSING THE LOOP. The type system naturally defines a new interpretation of types as sets of values:

$$[\![\_]\!]_{\mathcal{V}} : \mathcal{T} \to \mathcal{P}(\mathcal{V}), t \mapsto \{v \mid \vdash v : t\}.$$

It turns out that this interpretation satisfies the conditions of Definitions 4.1 and 4.5 (proof in Section 6.4):

THEOREM 5.4. *The function $[\![\_]\!]_{\mathcal{V}}$ is a structural set-theoretic interpretation.*

A natural question is whether this set-theoretic interpretation is a model. If this is the case, we would like to compare the subtyping relation it induces with the one used to define the type system (which was induced by the bootstrap model). The following theorem answers both questions (proof in Section 6.5):

THEOREM 5.5. *The following properties are equivalent:*

(1) *The interpretation $[\![\_]\!]_{\mathcal{V}}$ is a model.*
(2) *The interpretations $[\![\_]\!]_{\mathcal{V}}$ and $[\![\_]\!]$ induce the same subtyping relation.*
(3) *The bootstrap model $[\![\_]\!]$ is well founded.*

When the interpretation $[\![\_]\!]_{\mathcal{V}}$ is a model, we could use it as a new bootstrap model, define a new type system, and so on. The theorem says that this iteration is useless, because the old and the new bootstrap model already induce the same subtyping relation.

Note that the type soundness results do not depend on the fact that the interpretation $[\![\_]\!]_{\mathcal{V}}$ is a model. It holds even if the bootstrap model is not well founded.

5.3. CONSTRUCTION OF MODELS. All the results above would be void if we could not build a model. In this section, we claim the existence of models with specific properties (proofs in Section 6.8 and Section 6.10). Models can be compared by the amount of subtyping they allow. If $[\![\_]\!]_1$ and $[\![\_]\!]_2$ are two models, we write $[\![\_]\!]_1 \preceq [\![\_]\!]_2$ if:

$$\forall t, s \in \mathcal{T}. [\![t]\!]_1 \leq [\![s]\!]_1 \Rightarrow [\![t]\!]_2 \leq [\![s]\!]_2.$$

A model $[\![\_]\!]_2$ is ***universal*** if $[\![\_]\!]_1 \preceq [\![\_]\!]_2$ for any other model $[\![\_]\!]_1$. In other words, a model is universal if the subtyping relation it induces is the largest possible one. Clearly, two universal models induce the same subtyping relation.

THEOREM 5.6. *There exists a well founded and universal model.*

The next theorem shows that the notions of universality and well foundedness are not automatic.

THEOREM 5.7. *There exists a model that is not well founded. There exists a well founded model that is not universal.*

5.4. DECIDABILITY RESULTS. Finally, our system would be of little practical use if we were not able to decide the subtyping and typing relations. Fortunately, the decidability of the inclusion of basic types implies the following theorem:

THEOREM 5.8. *The subtyping relation induced by universal models is decidable.*

The proof of decidability (Section 6.9) essentially relies on three components: (i) the regularity of types, (ii) some algebraic properties of universal models, and (iii) the equivalence between subtyping and type emptiness problems (remember that $s \leq t \iff s\setminus t \simeq \mathbb{0}$.). The algebraic properties of the model can be used to decompose a type t into a set of types t_i's such that: (i) $t \simeq \mathbb{0}$ if and only if all $t_i \simeq \mathbb{0}$ and (ii) the t_i's are Boolean combinations of subterms of t (Section 6.2). We also introduce the concept of *simulation* (Definition 6.9) which characterizes sets of types that are closed with respect to the previous decomposition. By construction a type is equivalent to $\mathbb{0}$ if and only if there exists a simulation containing it (in that case, the simulation represents a co-inductive proof of its emptiness). A regular type has only a finite number of unique subterms, therefore it suffices to enumerate all the possible sets of Boolean combinations of its subterms and test whether any of them is a simulation (which is decidable for finite sets, and more efficient algorithms exist).

Decidability of subtyping does not immediately yield decidability of the typing relation, the problem being that the use of the negated arrows in the typing rule (*abstr*) makes the minimum typing property fail. Therefore, we need to introduce a new syntactic category, type schemes: a type-scheme represents the set of all valid types for a well-typed expression (Section 6.12). This technical construction allows us to state the decidability of the type-checking problem.

THEOREM 5.9. *When the subtyping relation is decidable, the type checking problem (deciding whether* $\Gamma \vdash e : t$ *for given* Γ*, e, t) is decidable.*

6. *Formal Development*

In this section, we establish the theorems stated in the previous section and other intermediate lemmas. It can be skipped in the first reading.

6.1. DISJUNCTIVE NORMAL FORMS FOR TYPES. We write $\mathcal{A}$ for atoms and we use the meta-variable a to range over atoms. There are three kinds of atoms (and values), which we denote by the meta-variable u ranging over the set $U = \{\mathbf{prod}, \mathbf{fun}, \mathbf{basic}\}$.

We write $\mathcal{A}_{\mathbf{fun}}$ for atoms of the form $t_1 \to t_2$, $\mathcal{A}_{\mathbf{prod}}$ for atoms of the form $t_1 \times t_2$, and $\mathcal{A}_{\mathbf{basic}}$ for basic types. We have $\mathcal{A} = \mathcal{A}_{\mathbf{fun}} + \mathcal{A}_{\mathbf{prod}} + \mathcal{A}_{\mathbf{basic}}$. For what concerns values, their kinding too is straightforward: values of the form c, (v_1, v_2), and $\mu f(\cdots).\lambda x.\mathrm{e}$ have respectively kind **basic**, **prod**, and **fun**.

Every type can be seen as a finite Boolean combination of atoms. It is convenient to work with disjunctive normal forms.

Definition 6.1. A (disjunctive) ***normal form*** τ is a finite set of pairs of finite sets of atoms, that is, an element of $\mathcal{P}_f(\mathcal{P}_f(\mathcal{A}) \times \mathcal{P}_f(\mathcal{A}))$ (where $\mathcal{P}_f$ denotes the finite powerset).

If $[\![\_]\!] : \mathcal{T} \to \mathcal{P}(D)$ is an arbitrary set-theoretic interpretation and τ a normal form, we define $[\![\tau]\!]$ as:

$$[\![\tau]\!] = \bigcup_{(P,N)\in\tau} \bigcap_{a\in P} [\![a]\!] \cap \bigcap_{a\in N} (D\backslash[\![a]\!]).$$

(Note that, with the convention that an intersection over an empty set is taken to be D, $[\![\tau]\!] \subseteq D$.)

LEMMA 6.2. *For every type $t \in \mathcal{T}$, it is possible to compute a normal form $\mathcal{N}(t)$ such that for every set-theoretic interpretation $[\![\_]\!]$, $[\![t]\!] = [\![\mathcal{N}(t)]\!]$.*

PROOF. We will actually define two functions $\mathcal{N}$ and $\mathcal{N}'$, both from types to $\mathcal{P}_f(\mathcal{P}_f(\mathcal{A}) \times \mathcal{P}_f(\mathcal{A}))$, by mutual induction over types.

$$\begin{array}{lll}
\mathcal{N}(\mathbb{0}) & = & \varnothing \\
\mathcal{N}(a) & = & \{(\{a\}, \varnothing)\} \\
\mathcal{N}(t_1 \vee t_2) & = & \mathcal{N}(t_1) \cup \mathcal{N}(t_2) \\
\mathcal{N}(\neg t) & = & \mathcal{N}'(t) \\
\mathcal{N}'(\mathbb{0}) & = & \{(\varnothing, \varnothing)\} \\
\mathcal{N}'(a) & = & \{(\varnothing, \{a\})\} \\
\mathcal{N}'(t_1 \vee t_2) & = & \{(P_1 \cup P_2, N_1 \cup N_2) \mid (P_1, N_1) \in \mathcal{N}'(t_1), (P_2, N_2) \in \mathcal{N}'(t_2)\} \\
\mathcal{N}'(\neg t) & = & \mathcal{N}(t).
\end{array}$$

We check by induction over the type t the following property:

$$[\![t]\!] = [\![\mathcal{N}(t)]\!] = D\backslash[\![\mathcal{N}'(t)]\!].$$ □

As an example, consider the type $t = a_1 \wedge (a_2 \vee \neg a_3)$ where a_1, a_2, a_3 are three atoms. Then $\mathcal{N}(t) = \{(\{a_1, a_2\}, \varnothing), (\{a_1\}, \{a_3\})\}$. This corresponds to the fact that t and $(a_1 \wedge a_2) \vee (a_1 \wedge \neg a_3)$ have the same interpretation for any set-theoretic interpretation of the type algebra.

Note that the converse result is true as well: for any normal form τ, we can find a type t such that $[\![t]\!] = [\![\tau]\!]$ for any set-theoretic interpretation. Normal forms are thus simply a different, but handy, syntax for types. In particular, we can rephrase in Definition 4.4 the condition for a set-theoretic interpretation to be a model as: for any normal form τ, $[\![\tau]\!] = \varnothing \iff \mathbb{E}(\tau) = \varnothing$.

For these reason henceforth we will often confound the notions of types and normal form, and we will often speak of the *type* τ, taking the latter as a canonical representative of all the types in $\mathcal{N}^{-1}(\tau)$.

6.2. STUDY OF THE SUBTYPING RELATION. Definition 4.4 is rather intensional. In this section, we establish a more extensional criterion for a set-theoretic interpretation to be a model.

Let $[\![\_]\!]$ be a set-theoretic interpretation. We are interested in comparing the assertions $\mathbb{E}(\tau) = \varnothing$ and $[\![\tau]\!] = \varnothing$, for a normal form τ. Clearly, $\mathbb{E}(\tau) = \varnothing$ is equivalent to:

$$\forall (P, N) \in \tau.\ \bigcap_{a\in P} \mathbb{E}(a) \subseteq \bigcup_{a\in N} \mathbb{E}(a). \tag{3}$$

Let us write $\mathbb{E}^{\mathbf{basic}}D = \mathcal{C}$, $\mathbb{E}^{\mathbf{prod}}D = D^2$, $\mathbb{E}^{\mathbf{fun}}D = \mathcal{P}(D \times D_\Omega)$. We have $\mathbb{E}D = \bigcup_{u\in U}\mathbb{E}^u D$ where $U = \{\mathbf{prod}, \mathbf{fun}, \mathbf{basic}\}$. We can thus rewrite (3) as:

$$\forall u \in U.\forall(P, N) \in \tau.\ \bigcap_{a\in P}(\mathbb{E}(a) \cap \mathbb{E}^u D) \subseteq \bigcup_{a\in N}(\mathbb{E}(a) \cap \mathbb{E}^u D). \tag{4}$$

Since $[\![a]\!] \cap \mathbb{E}^u D = \varnothing$ if $a \notin \mathcal{A}_u$ and $[\![a]\!] \cap \mathbb{E}^u D = [\![a]\!]$ if $a \in \mathcal{A}_u$, we can rewrite (4) as:

$$\forall u \in U.\forall(P, N) \in \tau.(P \subseteq \mathcal{A}_u) \Rightarrow \left(\bigcap_{a\in P}\mathbb{E}(a) \subseteq \bigcup_{a\in N\cap\mathcal{A}_u}\mathbb{E}(a)\right), \tag{5}$$

(where the intersection is taken to be $\mathbb{E}^u D$ when $P = \varnothing$.)

To further decompose these predicates, we will take advantage of the set-theoretic interpretation of the semantic subtyping and rely on two set-theoretic facts, one for product types, one for arrow types. Let us introduce some new notation that will make formulae clearer, and then start with product types, following an argument similar to the one used by Hosoya et al. [2000].

Notation 6.3. Let S_1, S_2 denote two sets such that $S_1 \subseteq S_2$. We use $\overline{S_1}^{S_2}$ to denote the complement of S_1 with respect to S_2, that is $S_2 \backslash S_1$.

LEMMA 6.4. *Let $(X_i)_{i\in P}, (X_i)_{i\in N}$ (respectively, $(Y_i)_{i\in P}, (Y_i)_{i\in N}$) be two families of subsets of D_1 (respectively, D_2). Then:*

$$\left(\bigcap_{i\in P} X_i \times Y_i\right) \setminus \left(\bigcup_{i\in N} X_i \times Y_i\right) = \bigcup_{N'\subseteq N}\left(\bigcap_{i\in P} X_i \setminus \bigcup_{i\in N'} X_i\right) \times \left(\bigcap_{i\in P} Y_i \setminus \bigcup_{i\in N\setminus N'} Y_i\right)$$

(with the conventions: $\bigcap_{i\in\varnothing} X_i \times Y_i = D_1 \times D_2$; $\bigcap_{i\in\varnothing} X_i = D_1$ and $\bigcap_{i\in\varnothing} Y_i = D_2$).

Note that we use the same notation for elements in the families $(X_i)_{i\in P}$ and $(X_i)_{i\in N}$. This is not problematic since the sets P and N can be different.

PROOF. First, we notice that:

$$\overline{X_i \times Y_i}^{D_1\times D_2} = \left(\overline{X_i}^{D_1} \times D_2\right) \cup \left(D_1 \times \overline{Y_i}^{D_2}\right).$$

By distributing intersections over unions, we get:

$$\begin{aligned}
&\bigcap_{i\in N} \overline{X_i \times Y_i}^{D_1\times D_2} \\
&\quad = \bigcup_{N'\subseteq N}\left(\bigcap_{i\in N'}\left(\overline{X_i}^{D_1} \times D_2\right) \cap \bigcap_{i\in N\setminus N'}\left(D_1 \times \overline{Y_i}^{D_2}\right)\right) \\
&\quad = \bigcup_{N'\subseteq N}\left(\bigcap_{i\in N'}\overline{X_i}^{D_1} \times \bigcap_{i\in N\setminus N'}\overline{Y_i}^{D_2}\right).
\end{aligned}$$

And finally:

$$\left(\bigcap_{i\in P} X_i \times Y_i\right) \cap \left(\bigcap_{i\in N} \overline{X_i \times Y_i}^{D_1\times D_2}\right)$$
$$= \bigcup_{N'\subseteq N} \left(\left(\bigcap_{i\in P} X_i \cap \bigcap_{i\in N'} \overline{X_i}^{D_1}\right) \times \left(\bigcap_{i\in P} Y_i \cap \bigcap_{i\in N\setminus N'} \overline{Y_i}^{D_2}\right)\right).$$

We get the expected result by applying De Morgan laws. □

We get an immediate corollary.

LEMMA 6.5. *Let P, N be two finite subsets of $\mathcal{A}_{\mathbf{prod}}$. We have:*

$$\bigcap_{a\in P} \mathbb{E}(a) \subseteq \bigcup_{a\in N} \mathbb{E}(a) \iff$$
$$\forall N' \subseteq N. \left[\!\!\left[\bigwedge_{t_1\times t_2\in P} t_1 \wedge \bigwedge_{t_1\times t_2\in N'} \neg t_1 \right]\!\!\right] = \varnothing \vee \left[\!\!\left[\bigwedge_{t_1\times t_2\in P} t_2 \wedge \bigwedge_{t_1\times t_2\in N\setminus N'} \neg t_2 \right]\!\!\right] = \varnothing$$

(with the convention $\bigcap_{a\in\varnothing} \mathbb{E}(a) = \mathbb{E}^{\mathbf{prod}} D$).

We will now establish a similar result for arrow types. We first decompose the set-theoretic $\rightarrow$ operator (Definition 4.2) into more primitive operators: powerset, complement, Cartesian product.

LEMMA 6.6. *Let $X, Y \subseteq D$. Then:*

$$X \rightarrow Y = \mathcal{P}\left(\overline{X \times \overline{Y}^{D_\Omega}}^{D\times D_\Omega}\right).$$

PROOF. The result comes from a simple computation:

$$\begin{aligned} X \rightarrow Y &= \{f \subseteq D \times D_\Omega \mid \forall (x, y) \in f.\, \neg(x \in X \wedge y \notin Y)\} \\ &= \{f \subseteq D \times D_\Omega \mid f \cap X \times \overline{Y}^{D_\Omega} = \varnothing\} \\ &= \left\{f \subseteq D \times D_\Omega \mid f \subseteq \overline{X \times \overline{Y}^{D_\Omega}}^{D\times D_\Omega}\right\} \end{aligned}$$ □

LEMMA 6.7. *Let $(X_i)_{i\in P}$ and $(X_i)_{i\in N}$ be two families of subsets of D. Then:*

$$\bigcap_{i\in P} \mathcal{P}(X_i) \subseteq \bigcup_{i\in N} \mathcal{P}(X_i) \iff \exists i_o \in N.\, \bigcap_{i\in P} X_i \subseteq X_{i_0}.$$

PROOF. The $\Leftarrow$ implication is simple: if $\bigcap_{i\in P} X_i \subseteq X_{i_0}$ with $i_0 \in N$, then $\bigcap_{i\in P} \mathcal{P}(X_i) = \mathcal{P}(\bigcap_{i\in P} X_i) \subseteq \mathcal{P}(X_{i_0}) \subseteq \bigcup_{i\in N} \mathcal{P}(X_i)$. Let us prove the opposite direction. We assume that $\bigcap_{i\in P} \mathcal{P}(X_i) \subseteq \bigcup_{i\in N} \mathcal{P}(X_i)$. The set $\bigcap_{i\in P} X_i$ belongs to all the $\mathcal{P}(X_i)$ for $i \in P$. It is thus in the union of all the $\mathcal{P}(X_i)$ for $i \in N$. We can thus find some $i_0 \in N$ such that $\bigcap_{i\in P} X_i \in \mathcal{P}(X_{i_0})$, which concludes the proof. □

LEMMA 6.8. *Let P and N be two finite subsets of $\mathcal{A}_{\mathbf{fun}}$. Then:*

$$\bigcap_{a\in P}\mathbb{E}(a)\subseteq\bigcup_{a\in N}\mathbb{E}(a)$$
$$\Longleftrightarrow$$
$$\exists(t_0\to s_0)\in N.\ \forall P'\subseteq P.\ \left[\!\!\left[t_0\setminus\left(\bigvee_{t\to s\in P'}t\right)\right]\!\!\right]=\varnothing\ \vee\ \left\{\left[\!\!\left[\left(\bigwedge_{\substack{t\to s\in P\setminus P'\\ P\neq P'}}s\right)\setminus s_0\right]\!\!\right]=\varnothing.\right.$$

(with the convention $\bigcap_{a\in\varnothing}\mathbb{E}(a)=\mathbb{E}^{\mathbf{fun}}D$).

PROOF. The result follows from Lemmas 6.6, 6.7, and 6.4, by noticing that in the condition $\bigcap_{t\to s\in P\setminus P'}[\![s]\!]\subseteq[\![s_0]\!]$ that appears, the convention is to interpret the intersection as being D_Ω if $P=P'$, which makes the inclusion impossible. □

Lemma 6.8 tells us how to decompose subtyping between arrow types. For instance, we can deduce from the lemma that $\mathbb{E}((t_1\to s_1)\wedge(t_2\to s_2))\subseteq\mathbb{E}(t\to s)$ holds if and only if the four following properties are satisfied:

—$[\![t]\!]=\varnothing$ or $[\![s_1\wedge s_2]\!]\subseteq[\![s]\!]$
—$[\![t]\!]\subseteq[\![t_1]\!]$ or $[\![s_2]\!]\subseteq[\![s]\!]$
—$[\![t]\!]\subseteq[\![t_2]\!]$ or $[\![s_1]\!]\subseteq[\![s]\!]$
—$[\![t]\!]\subseteq[\![t_1\vee t_2]\!]$.

Lemmas 6.5 and 6.8, together with the property (5), suggest the following definition and give immediately the result of Theorem 6.10 below.

Definition 6.9 (*Simulation*). Let $\mathcal{S}$ be an arbitrary set of normal forms. We define another set of normal forms $\mathbb{E}\mathcal{S}$ by:

$$\mathbb{E}\mathcal{S}=\{\tau\mid\forall u\in U.\forall(P,N)\in\tau.\ (P\subseteq\mathcal{A}_u\Rightarrow C_u^{P,N\cap\mathcal{A}_u})\},$$

where:

$$C^{P,N}_{\mathbf{basic}}\ ::=\ \mathcal{C}\cap\bigcap_{b\in P}\mathbb{B}[\![b]\!]\subseteq\bigcup_{b\in N}\mathbb{B}[\![b]\!]$$
$$C^{P,N}_{\mathbf{prod}}\ ::=\ \forall N'\subseteq N.\ \left\{\begin{array}{c}\mathcal{N}\left(\bigwedge_{t_1\times t_2\in P}t_1\wedge\bigwedge_{t_1\times t_2\in N'}\neg t_1\right)\in\mathcal{S}\\ \vee\\ \mathcal{N}\left(\bigwedge_{t_1\times t_2\in P}t_2\wedge\bigwedge_{t_1\times t_2\in N\setminus N'}\neg t_2\right)\in\mathcal{S}\end{array}\right.$$

$$C_{\mathbf{fun}}^{P,N} ::= \exists t_0 \to s_0 \in N.\, \forall P' \subseteq P. \begin{cases} \mathcal{N}\left(t_0 \wedge \bigwedge_{t \to s \in P'} \neg t\right) \in \mathcal{S} \\ \vee \\ \begin{cases} P \neq P' \\ \wedge \\ \mathcal{N}\left((\neg s_0) \wedge \bigwedge_{t \to s \in P \setminus P'} s\right) \in \mathcal{S}. \end{cases} \end{cases}$$

We say that $\mathcal{S}$ is a ***simulation*** if:

$$\mathcal{S} \subseteq \mathbb{E}\mathcal{S}.$$

The intuition is that if we consider the statements of Lemmas 6.5 and 6.8 as if they were rewriting rules (from right to left), then $\mathbb{E}\mathcal{S}$ contains all the types that we can deduce in one step reduction to be empty when we suppose that the types in $\mathcal{S}$ are empty. A simulation is thus a set that is already saturated with respect to such a rewriting. In particular, if we consider the statements of Lemmas 6.5 and 6.8 as inference rules for determining when a type is equal to $\mathbb{0}$, then $\mathbb{E}\mathcal{S}$ is the set of immediate consequences of $\mathcal{S}$, and a simulation is a *self-justifying* set, that is a co-inductive proof of the fact that all its elements are equal to $\mathbb{0}$. Of course, this latter property will play a crucial role to decide the subtyping relation (see Section 6.9).

THEOREM 6.10. *Let $[\![\_]\!] : \mathcal{T} \to \mathcal{P}(D)$ be a set-theoretic interpretation. We define a set of normal forms $\mathcal{S}$ by:*

$$\mathcal{S} = \{\tau \mid [\![\tau]\!] = \varnothing\}$$

Then:

$$\mathbb{E}\mathcal{S} = \{\tau \mid \mathbb{E}(\tau) = \varnothing\}$$

PROOF. Immediate consequence of Lemmas 6.5 and 6.8. □

COROLLARY 6.11. *Let $[\![\_]\!]$ be a set-theoretic interpretation of types. We define as above $\mathcal{S} = \{\tau \mid [\![\tau]\!] = \varnothing\}$. Then $[\![\_]\!]$ is a model if and only if $\mathcal{S} = \mathbb{E}\mathcal{S}$.*

This corollary implies that the condition for a set-theoretic interpretation to be a model depends only on the subtyping relation it induces.

COROLLARY 6.12. *Let $[\![\_]\!]_1 : \mathcal{T} \to \mathcal{P}(D_1)$ be a model and $[\![\_]\!]_2 : \mathcal{T} \to \mathcal{P}(D_2)$ be a set-theoretic interpretation. Then, the following assertions are equivalent:*

—$[\![\_]\!]_2$ is a model and it induces the same subtyping relation as $[\![\_]\!]_1$.
—for any type t, $[\![t]\!]_1 = \varnothing \iff [\![t]\!]_2 = \varnothing$.

The following lemma, which is an immediate corollary of Lemma 6.8 gives several properties about subtyping between arrow types in a model, which will be needed to study the meta-theory of the type system (see the proofs of Lemma 6.15, Lemma 6.21, and Lemma 6.37).

LEMMA 6.13 (STRONG DISJUNCTION FOR ARROWS). *Let $\leq$ be the subtyping relation induced by a model, and P,N two finite sets of arrow types. Then:*

$$\bigwedge_{a\in P} a \leq \bigvee_{a\in N} a \iff \exists a_0 \in N.\ \bigwedge_{a\in P} a \leq a_0.$$

From this we immediately deduce that:

If P,N are finite sets of arrow types and if a_0 is an arrow type, if we define t as $\bigwedge_{a\in P} a \setminus \bigvee_{a\in N} a$ and if we assume that $t \not\simeq \mathbb{0}$, then:

$$t \leq a_0 \iff \bigwedge_{a\in P} a \leq a_0.$$

If P,N_1,N_2 are finite sets of arrow types, then:

$$\left.\begin{array}{c} \bigwedge_{a\in P} a \not\leq \bigvee_{a\in N_1} a \\ \wedge \\ \bigwedge_{a\in P} a \not\leq \bigvee_{a\in N_2} a. \end{array}\right\} \iff \bigwedge_{a\in P} a \not\leq \bigvee_{a\in N_1\cup N_2} a.$$

6.3. SYNTACTICAL META-THEORY OF THE TYPE SYSTEM. In this section and in the following one, we fix a bootstrap model $[\![\_]\!] : \mathcal{T} \to \mathcal{P}(D)$, we write $\leq$ for the induced subtyping relation and $\simeq$ for the associated equivalence relation, and we study the resulting typing judgment $\Gamma \vdash e : t$.

LEMMA 6.14 (STRENGTHENING). *Let Γ_1 and Γ_2 be two typing environments such that for any x in the domain of Γ_1, we have $\Gamma_2(x) \leq \Gamma_1(x)$. If $\Gamma_1 \vdash e : t$, then $\Gamma_2 \vdash e : t$.*

PROOF. Induction on the derivation of $\Gamma_1 \vdash e : t$. We simply introduce an instance of the subsumption rule below each instance of the (var) rule. □

LEMMA 6.15 (ADMISSIBILITY OF THE INTERSECTION RULE). *If $\Gamma \vdash e : t_1$ and $\Gamma \vdash e : t_2$, then $\Gamma \vdash e : t_1 \wedge t_2$.*

PROOF. By induction on the structure of the two typing derivations.

Let us first consider the case when the last rule applied to one of the two derivations is $(subsum)$, say:

$$\frac{\dfrac{\ldots}{\Gamma \vdash e : s_1} \quad s_1 \leq t_1}{\Gamma \vdash e : t_1} \quad \frac{\ldots}{\Gamma \vdash e : t_2}.$$

The induction hypothesis gives $\Gamma \vdash e : s_1 \wedge t_2$. But $s_1 \wedge t_2 \leq t_1 \wedge t_2$ because $s_1 \leq t_1$, and a new application of $(subsum)$ gives $\Gamma \vdash e : t_1 \wedge t_2$ as expected.

In all the remaining cases, the two derivations end with an instance of the same rule (which depends on the toplevel constructor of e).

Rules $(const)$, (var), (rnd): Those rules give only one possible type t for e, and $t \wedge t \simeq t$.

Rule $(appl)$: The situation is as follows:

$$\frac{\dfrac{\ldots}{\Gamma \vdash e_1 : t_1 \to t_2} \quad \dfrac{\ldots}{\Gamma \vdash e_2 : t_1}}{\Gamma \vdash e_1 e_2 : t_2} \quad \frac{\dfrac{\ldots}{\Gamma \vdash e_1 : t_1' \to t_2'} \quad \dfrac{\ldots}{\Gamma \vdash e_2 : t_1'}}{\Gamma \vdash e_1 e_2 : t_2'}.$$

The induction hypothesis gives $\Gamma \vdash e_1 : (t_1 \to t_2) \wedge (t'_1 \to t'_2)$ and $\Gamma \vdash e_2 : t_1 \wedge t'_1$. To conclude, it is enough to check that $(t_1 \to t_2) \wedge (t'_1 \to t'_2) \leq (t_1 \wedge t'_1) \to (t_2 \wedge t'_2)$, which can be proved as follows:

$$\begin{aligned}
&\mathbb{E}((t_1 \to t_2) \wedge (t'_1 \to t'_2)) \\
&\quad = (\llbracket t_1 \rrbracket \to \llbracket t_2 \rrbracket) \cap (\llbracket t'_1 \rrbracket \to \llbracket t'_2 \rrbracket) \\
&\quad = \{f \in \mathbb{E}^{\mathbf{fun}} D \mid \forall (x, y) \in f. (x \in \llbracket t_1 \rrbracket \Rightarrow y \in \llbracket t_2 \rrbracket) \wedge (x \in \llbracket t'_1 \rrbracket \Rightarrow y \in \llbracket t'_2 \rrbracket)\} \\
&\quad \subseteq \{f \in \mathbb{E}^{\mathbf{fun}} D \mid \forall (x, y) \in f. (x \in \llbracket t_1 \rrbracket \cap \llbracket t'_1 \rrbracket \Rightarrow y \in (\llbracket t_2 \rrbracket \cap \llbracket t'_2 \rrbracket)\} \\
&\quad = \mathbb{E}((t_1 \wedge t'_1) \to (t_2 \wedge t'_2)).
\end{aligned}$$

Rule (*pair*): The situation is as follows:

$$\frac{\dfrac{\cdots}{\Gamma \vdash e_1 : t_1} \quad \dfrac{\cdots}{\Gamma \vdash e_2 : t_2}}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2} \quad \frac{\dfrac{\cdots}{\Gamma \vdash e_1 : t'_1} \quad \dfrac{\cdots}{\Gamma \vdash e_2 : t'_2}}{\Gamma \vdash (e_1, e_2) : t'_1 \times t'_2}.$$

Let $t''_1 = t_1 \wedge t'_1$ and $t''_2 = t_2 \wedge t'_2$. By applying the induction hypothesis twice, we get $\Gamma \vdash e_1 : t''_1$ and $\Gamma \vdash e_2 : t''_2$. The rule (*pair*) gives $\Gamma \vdash (e_1, e_2) : t''_1 \times t''_2$. To conclude, it is enough to see that $t''_1 \times t''_2 \simeq (t_1 \times t_2) \wedge (t'_1 \times t'_2)$. Indeed:

$$\begin{aligned}
\mathbb{E}(t''_1 \times t''_2) &= (\llbracket t_1 \rrbracket \cap \llbracket t'_1 \rrbracket) \times (\llbracket t_2 \rrbracket \cap \llbracket t'_2 \rrbracket) = \llbracket t_1 \wedge t_2 \rrbracket \cap \llbracket t'_1 \wedge t'_2 \rrbracket \\
&= \mathbb{E}((t_1 \times t_2) \wedge (t'_1 \times t'_2))
\end{aligned}$$

Rule (*case*): Let us consider this situation:

$$\frac{\dfrac{\cdots}{\Gamma \vdash e : t_0} \quad \dfrac{\cdots}{(x : t_i), \Gamma \vdash e_i : s}}{\Gamma \vdash (x = e \in t \ ? \ e_1 | e_2) : s} \quad \frac{\dfrac{\cdots}{\Gamma \vdash e : t'_0} \quad \dfrac{\cdots}{(x : t'_i), \Gamma \vdash e_i : s'}}{\Gamma \vdash (x = e \in t \ ? \ e_1 | e_2) : s'}.$$

with $t_1 = t_0 \wedge t$, $t_2 = t_0 \backslash t$, $t'_1 = t'_0 \wedge t$, $t'_2 = t'_0 \backslash t$. The induction hypothesis gives: $\Gamma \vdash e : t''_0$ with $t''_0 = t_0 \wedge t'_0$. Let us define $t''_1 = t''_0 \wedge t$ and $t''_2 = t''_0 \backslash t$. Let $i \in \{1, 2\}$. We have $t''_i \leq t_i$ and thus, according to Lemma 6.14, $(x : t''_i), \Gamma \vdash e_i : s$. Similarly, we get $(x : t''_i), \Gamma \vdash e_i : s'$, and thus, applying again the induction hypothesis $(x : t''_i), \Gamma \vdash e_i : s''$ where $s'' = s \wedge s'$. Then, with the (*case*) rule, we establish $\Gamma \vdash (x = e \in t \ ? \ e_1 | e_2) : s''$ as expected.

The special cases (where $t_i \simeq \mathbb{0}$ or $t'_i \simeq \mathbb{0}$) are similar.

Rule (*abstr*): Let us consider two applications of the rule (*abstr*) to the same abstraction $\mu f (t_1 \to s_1; \cdots; t_n \to s_n).\lambda x.e$ with the following types:

$$\begin{aligned}
t &= \bigwedge_{i=1..n} (t_i \to s_i) \wedge \bigwedge_{j=1..m} \neg(t'_j \to s'_j) \\
t' &= \bigwedge_{i=1..n} (t_i \to s_i) \wedge \bigwedge_{j=m+1..m'} \neg(t'_j \to s'_j),
\end{aligned}$$

where $t \not\simeq \mathbb{0}$ and $t' \not\simeq \mathbb{0}$. We define:

$$t'' = \bigwedge_{i=1..n} (t_i \to s_i) \wedge \bigwedge_{j=1..m'} \neg(t'_j \to s'_j)$$

We have $t'' \simeq t \wedge t'$. We only need to verify that some instance of the rule (*abstr*) allows us to deduce the type t'' for the abstraction. For $i = 1..n$, we have, by hypothesis $(f : t), (x : t_i), \Gamma \vdash e : s_i$, and thus, in accordance with Lemma 6.14, $(f : t''), (x : t_i), \Gamma \vdash e : s_i$. Then, we check that $t'' \not\simeq \mathbb{0}$, which results immediately from Lemma 6.13. In this case, we have not used the induction hypothesis. □

COROLLARY 6.16. *Let Γ be a typing environment and e an expression that is well typed under Γ. Then the set $\{t \in \mathcal{T} \mid (\Gamma \vdash e : t) \vee (\Gamma \vdash e : \neg t)\}$ contains $\mathbb{0}$ and is closed under $\vee$ and $\neg$ (and thus $\wedge$).*

PROOF. Let E be the set introduced in the statement. It is clearly closed under $\neg$ and invariant under the equivalence $\simeq$. We have $\Gamma \vdash e : \mathbb{1} = \neg\mathbb{0}$ because of the subsumption rule, and thus $\mathbb{0} \in E$. What remains is to prove that E is closed under $\vee$. So let us take two elements t_1 and t_2 in E. If $\Gamma \not\vdash e : t_1 \vee t_2$, then because of $(subsum)$, we get $\Gamma \not\vdash e : t_1$ and $\Gamma \not\vdash e : t_2$. Because t_1 and t_2 are in E, we thus have $\Gamma \vdash e : \neg t_1$ and $\Gamma \vdash e : \neg t_2$. Lemma 6.15 then gives $\Gamma \vdash e : \neg t_1 \wedge \neg t_2$. And $\neg t_1 \wedge \neg t_2 \simeq \neg(t_1 \vee t_2)$. We have thus proved that $\Gamma \vdash e : t_1 \vee t_2$ or $\Gamma \vdash e : \neg(t_1 \vee t_2)$. □

LEMMA 6.17 (SUBSTITUTION). *Let $e, e_1, \ldots, e_n$ be expressions, $x_1, \ldots, x_n$ distinct variables, $t, t_1, \ldots, t_n$ types, and Γ a typing environment. Then:*

$$\begin{cases} (x_1 : t_1), \ldots, (x_n : t_n), \Gamma \vdash e : t \\ \forall i = 1..n.\ \Gamma \vdash e_i : t_i \end{cases} \Rightarrow \Gamma \vdash e[x_1 := e_1; \ldots; x_n := e_n] : t$$

PROOF. By induction on the typing derivation for $(x_1 : t_1), \ldots, (x_n : t_n), \Gamma \vdash e : t$. We simply "plug" a copy of the derivation for $\Gamma \vdash e_i : t_i$ wherever the rule (var) is used for variable x_i. □

6.4. INTERPRETING TYPES AS SETS OF VALUES. The syntactical properties obtained in the previous section are used here to prove some properties about the interpretation of types as sets of values, as defined in Section 5.2: $[\![t]\!]_{\mathcal{V}} = \{v \mid \vdash v : t\}$

LEMMA 6.18. *If $t \leq s$, then $[\![t]\!]_{\mathcal{V}} \subseteq [\![s]\!]_{\mathcal{V}}$. In particular, if $t \simeq s$, then $[\![t]\!]_{\mathcal{V}} = [\![s]\!]_{\mathcal{V}}$.*

PROOF. Consequence of the subsumption rule. □

LEMMA 6.19. $[\![\mathbb{0}]\!]_{\mathcal{V}} = \varnothing$.

PROOF. We prove that $(\vdash v : t) \Rightarrow t \not\simeq \mathbb{0}$ by induction on the typing derivation. There are four cases to consider (one per value constructor, one for the subsumption rule). All of them are trivial. □

LEMMA 6.20. $[\![t_1 \wedge t_2]\!]_{\mathcal{V}} = [\![t_1]\!]_{\mathcal{V}} \cap [\![t_2]\!]_{\mathcal{V}}$.

PROOF. Lemma 6.18 gives $[\![t_1 \wedge t_2]\!]_{\mathcal{V}} \subseteq [\![t_i]\!]_{\mathcal{V}}$ for $i \in \{1, 2\}$, and thus $[\![t_1 \wedge t_2]\!]_{\mathcal{V}} \subseteq [\![t_1]\!]_{\mathcal{V}} \cap [\![t_2]\!]_{\mathcal{V}}$. Lemma 6.15 gives the opposite inclusion. □

LEMMA 6.21 (INVERSION).

$$\begin{aligned} [\![t_1 \times t_2]\!]_{\mathcal{V}} &= \{(v_1, v_2) \mid \vdash v_1 : t_1, \vdash v_2 : t_2\} \\ [\![b]\!]_{\mathcal{V}} &= \{c \mid b_c \leq b\} \\ [\![t \to s]\!]_{\mathcal{V}} &= \left\{ (\mu f(t_1 \to s_1; \cdots; t_n \to s_n).\lambda x.e) \in \mathcal{V}. \mid \bigwedge_{i=1..n} t_i \to s_i \leq t \to s \right\} \end{aligned}$$

Moreover, if v is a value and a is an atom of a different kind, then $\vdash v : \neg a$.

PROOF. For the three equalities, the $\supseteq$ inclusion is straightforward.

To prove the three opposite inclusions, let us start with a general remark. A derivation for $\vdash v : t$ can always be described as an instance of the rule corresponding

to the kind of v (rule (*const*) for constants, (*pair*) for pairs, and (*abstr*) for abstractions), followed by zero or more instance of (*subsum*). That is, we can always find another type $t' \leq t$ such that $\vdash v : t'$ is obtained by a direct application of the typing rule corresponding to v. If t is an atom a, then v is necessarily of the same kind as a. Indeed, if v is a pair, then t' is a product type; if v is a constant, t' is a basic type; if v is an abstraction, t' is an intersection of one or more arrow types (and maybe of zero or more negation of arrow types). In all cases, $t' \wedge a \simeq \mathbb{0}$ if a and v do not have the same kind, but since $t' \leq a$, this means that $t' \simeq \mathbb{0}$, which is impossible by Lemma 6.19. We also have proved the final remark in the statement of the Lemma (because if a and v does not have the same kind, then $t' \leq \neg a$, and thus $\vdash v : \neg a$).

Case $\vdash v : t_1 \times t_2$: The value is necessarily a pair (v_1, v_2) such that $\vdash v_1 : t_1', \vdash v_2 : t_2'$, and $t_1' \times t_2' \leq t_1 \times t_2$. But $t_1' \not\simeq \mathbb{0}$ and $t_2' \not\simeq \mathbb{0}$ because of Lemma 6.19, and thus $t_1' \leq t_1$ and $t_2' \leq t_2$. By subsumption, we get $\vdash v_1 : t_1$ and $\vdash v_2 : t_2$.

Case $\vdash v : b$: The value is necessarily a constant c such that $b_c \leq b$.

Case $\vdash v : t \to s$: The value is necessarily an abstraction $\mu f(t_1 \to s_1; \cdots; t_n \to s_n).\lambda x.e$. Here, the type t' has the form:

$$\bigwedge_{i=1..n} (t_i \to s_i) \wedge \bigwedge_{j=1..m} \neg(t_j' \to s_j')$$

with $t' \not\simeq \mathbb{0}$ and $t' \leq t \to s$. We can therefore apply the second point of Lemma 6.13 and deduce:

$$\bigwedge_{i=1..n} (t_i \to s_i) \leq t \to s. \qquad \square$$

LEMMA 6.22. $[\![\neg t]\!]_{\mathcal{V}} = \mathcal{V} \backslash [\![t]\!]_{\mathcal{V}}$.

PROOF. We have $(t \wedge \neg t) \simeq \mathbb{0}$ and, thus, $[\![t]\!]_{\mathcal{V}} \cap [\![\neg t]\!]_{\mathcal{V}} = [\![t \wedge \neg t]\!]_{\mathcal{V}} = [\![\mathbb{0}]\!]_{\mathcal{V}} = \varnothing$. So it remains to prove that $[\![t]\!]_{\mathcal{V}} \cup [\![\neg t]\!]_{\mathcal{V}} = \mathcal{V}$, that is:

$$\forall v. \forall t.\ (\vdash v : t) \vee (\vdash v : \neg t).$$

We proceed by induction over the pair (v, t). Thanks to Corollary 6.16, we can assume that t is an atom a. Lemma 6.21 gives $\vdash v : \neg a$ if a and v do not have the same kind. Now, we assume they have the same kind.

Case $v = c$: We have $\vdash c : b_c$. The set $\mathbb{E}(b_c)$ is a singleton (namely $\{c\}$), and thus $\mathbb{E}(b_c) \subseteq \mathbb{E}(a)$ or $\mathbb{E}(b_c) \subseteq \mathbb{E}(\neg a)$, that is: $b_c \leq a$ or $b_c \leq \neg a$. By subsumption, we get $\vdash b_c : a$ or $\vdash b_c : \neg a$.

Case $v = (v_1, v_2)$, $a = t_1 \times t_2$: If $\vdash v_1 : t_1$ and $\vdash v_2 : t_2$, we get $\vdash v : a$. Otherwise, say $\not\vdash v_1 : t_1$, we get $\vdash v_1 : \neg t_1$ by the induction hypothesis, and $\vdash v_2 : \mathbb{1}$ always holds, and thus we get $\vdash v : (\neg t_1) \times \mathbb{1}$. We conclude this case by the observation that $(\neg t_1) \times \mathbb{1} \leq \neg(t_1 \times t_2)$.

Case $v = \mu f(t_1 \to s_1; \cdots; t_n \to s_n).\lambda x.e$, $a = t \to s$: It is easy to see that $\vdash v : a$ if $\bigwedge_{i=1..n} t_i \to s_i \leq a$ and $\vdash v : \neg a$ otherwise. $\square$

LEMMA 6.23. $[\![t_1 \vee t_2]\!]_{\mathcal{V}} = [\![t_1]\!]_{\mathcal{V}} \cup [\![t_2]\!]_{\mathcal{V}}$.

PROOF. Using Lemmas 6.22, 6.20 and 6.18, we get: $[\![t_1 \vee t_2]\!]_{\mathcal{V}} = [\![\neg((\neg t_1) \wedge (\neg t_2))]\!]_{\mathcal{V}} = \mathcal{V} \backslash ([\![\neg t_1]\!]_{\mathcal{V}} \cap [\![\neg t_2]\!]_{\mathcal{V}}) = \mathcal{V} \backslash (\mathcal{V} \backslash [\![t_1]\!]_{\mathcal{V}} \backslash [\![t_2]\!]_{\mathcal{V}}) = [\![t_1]\!]_{\mathcal{V}} \cup [\![t_2]\!]_{\mathcal{V}}$. $\square$

From Lemmas 6.22, 6.23, and 6.19, we get that $[\![\_]\!]_{\mathcal{V}}$ is a set-theoretic interpretation.

To conclude the proof of Theorem 5.4, we need to check that it is structural. Clearly, $\mathcal{V}^2 \subseteq \mathcal{V}$ and Lemma 6.21 gives $[\![t_1 \times t_2]\!]_{\mathcal{V}} = [\![t_1]\!]_{\mathcal{V}} \times [\![t_2]\!]_{\mathcal{V}}$. Also, the relation induced by $(v_1, v_2) \triangleright v_i$ is clearly Noetherian.

6.5. CLOSING THE LOOP. In this section, we detail the proof of Theorem 5.5. We start with a lemma that shows that for an arbitrary finite set of arrow types, we can always find a well-typed and closed abstraction (hence, a value) having exactly this set of types in its interface. This fact will be used in the proof of Lemma 6.26.

LEMMA 6.24. *For every nonempty and finite family of arrow types $t_1 \to s_1, \ldots, t_n \to s_n$, the expression $\mu f(t_1 \to s_1; \cdots; t_n \to s_n).\lambda x.fx$ is a value.*

PROOF. Direct application of the typing rules and from the definition of values. □

LEMMA 6.25. *In every model, $[\![t]\!] = \varnothing \iff [\![\mathbb{1} \to t]\!] \subseteq [\![\mathbb{1} \to \mathbb{0}]\!]$ holds true.*

PROOF. Lemma 6.8 tells us that, in a model, $[\![\mathbb{1} \to t]\!] \subseteq [\![\mathbb{1} \to \mathbb{0}]\!]$ is equivalent to $([\![\mathbb{1}]\!] \subseteq [\![\mathbb{0}]\!] \vee [\![t]\!] \subseteq [\![\mathbb{0}]\!]) \wedge ([\![\mathbb{1}]\!] \subseteq [\![\mathbb{1}]\!])$, which is itself equivalent to $[\![t]\!] = \varnothing$. □

LEMMA 6.26. *The set-theoretic interpretation $[\![\_]\!]_{\mathcal{V}}$ is a model if and only if it induces the same subtyping relation as $[\![\_]\!]$.*

PROOF. The $\Leftarrow$ implication is given by Corollary 6.12. Let us assume that $[\![\_]\!]_{\mathcal{V}}$ is a model and prove that $[\![t]\!]_{\mathcal{V}} = \varnothing \iff t \simeq \mathbb{0}$ for any type t. The $\Leftarrow$ implication is given by Lemma 6.19. Let t be a type such that $[\![t]\!]_{\mathcal{V}} = \varnothing$. Because $[\![\_]\!]_{\mathcal{V}}$ is a model, Lemma 6.25 gives: $[\![\mathbb{1} \to t]\!]_{\mathcal{V}} \subseteq [\![\mathbb{1} \to \mathbb{0}]\!]_{\mathcal{V}}$. Now we consider the expression $v = \mu f(\mathbb{1} \to t).\lambda x.fx$. In accordance with Lemma 6.24, it is a value. In accordance with Lemma 6.21, it is an element of $[\![\mathbb{1} \to t]\!]_{\mathcal{V}}$, and thus also of $[\![\mathbb{1} \to \mathbb{0}]\!]_{\mathcal{V}}$, which means that $\mathbb{1} \to t \leq \mathbb{1} \to \mathbb{0}$ (again Lemma 6.21), and finally that $t \simeq \mathbb{0}$ (Lemma 6.25 for the model $[\![\_]\!]$). □

LEMMA 6.27. *If the bootstrap model is well founded, then $[\![\_]\!]_{\mathcal{V}}$ is a model.*

PROOF. By definition of a well founded model, there is a *structural* set-theoretic interpretation which induces the same subtyping relation as the bootstrap model. It is thus also a model. Since the type system and $[\![\_]\!]_{\mathcal{V}}$ only depend on this subtyping relation, we can assume that the bootstrap model is not only well-founded but also structural. We will use the Noetherian relation $\triangleright$ from Definition 4.5.

We need to prove that, for every type t, $[\![t]\!]_{\mathcal{V}} = \varnothing \iff t \simeq \mathbb{0}$. The $\Leftarrow$ implication is given by Lemma 6.19 and Lemma 6.18. We actually prove by induction (using the $\triangleright$ relation) that for all $d \in D$, the following property holds: $(\forall t \in \mathcal{T}.\, d \in [\![t]\!] \Rightarrow [\![t]\!]_{\mathcal{V}} \neq \varnothing)$.

Consider a type t such that $d \in [\![t]\!]$. If $d = (d_1, d_2) \in D^2$, then it is in the set

$$[\![t]\!] \cap D^2 = \bigcup_{(P,N)\in\mathcal{N}(t)} \left(D^2 \cap \bigcap_{a\in P} [\![a]\!] \setminus \bigcup_{a\in N} [\![a]\!] \right).$$

We can thus find $(P, N) \in \mathcal{N}(t)$ such that $d \in D^2 \cap \bigcap_{a\in P} [\![a]\!] \setminus \bigcup_{a\in N} [\![a]\!]$. Note that if a is an atom that is not a product type, then $D^2 \cap [\![a]\!] = [\![\mathbb{1}\times\mathbb{1}]\!] \cap [\![a]\!] = \varnothing$,

because $\mathbb{E}(\mathbb{1}\times\mathbb{1}) \cap \mathbb{E}(a) = \varnothing$. We can thus assume that $P \subseteq \mathcal{A}_{\mathbf{prod}}$, and we have $d \in \bigcap_{t_1\times t_2\in P}([\![t_1]\!] \times [\![t_2]\!]) \setminus \bigcup_{t_1\times t_2\in N}([\![t_1]\!] \times [\![t_2]\!])$. If we write $d = (d_1, d_2)$, then Lemma 6.4 gives some $N' \subseteq N$ such that $d_1 \in [\![s_1]\!]$ and $d_2 \in [\![s_2]\!]$ for:

$$\begin{cases} s_1 = \bigwedge_{t_1\times t_2\in P} t_1 \setminus \bigvee_{t_1\times t_2\in N'} t_1 \\ s_2 = \bigwedge_{t_1\times t_2\in P} t_2 \setminus \bigvee_{t_1\times t_2\in N\setminus N'} t_2. \end{cases}$$

The induction hypothesis applied to d_1 and d_2 gives $[\![s_1]\!]_{\mathcal{V}} \neq \varnothing$ and $[\![s_2]\!]_{\mathcal{V}} \neq \varnothing$, and thus $[\![s_1\times s_2]\!]_{\mathcal{V}} \neq \varnothing$. To conclude this case, we observe that $s_1\times s_2 \leq t$, using again Lemma 6.4.

Now, we assume that $d \notin D^2 = [\![\mathbb{1}\times\mathbb{1}]\!]$. We thus have $d \in [\![t\setminus\mathbb{1}\times\mathbb{1}]\!]$, which implies that $t\setminus\mathbb{1}\times\mathbb{1} \not\simeq \mathbb{0}$. As a consequence $\mathbb{E}(t\setminus\mathbb{1}\times\mathbb{1}) \neq \varnothing$, and thus $\mathbb{E}(t) \cap (\mathbb{E}D \setminus \mathbb{E}^{\mathbf{prod}} D) \neq \varnothing$. We are in at least one of the two cases:

$\mathbb{E}(t) \cap \mathcal{C} \neq \varnothing$: let $c \in \mathbb{E}(t) \cap \mathcal{C}$. We have $\mathbb{E}(b_c) = \{c\} \subseteq \mathbb{E}(t)$, and thus $b_c \leq t$. We conclude that $\vdash c : t$.

$\mathbb{E}(t) \cap \mathbb{E}^{\mathbf{fun}} D \neq \varnothing$: we have:

$$\mathbb{E}(t) \cap \mathbb{E}^{\mathbf{fun}} D = \bigcup_{(P,N)\in\mathcal{N}(t) \text{ s.t. } P\subseteq\mathcal{A}_{\mathbf{fun}}} \left(\mathbb{E}^{\mathbf{fun}} D \cap \bigcap_{a\in P} \mathbb{E}(a) \setminus \bigcup_{a\in N} \mathbb{E}(a) \right).$$

This set is not empty. We can thus find an element (P, N) in $\mathcal{N}(t)$ such that $P = \{t_1\to s_1, \ldots, t_n\to s_n\}$, $N \cap \mathcal{A}_{\mathbf{fun}} = \{t'_1\to s'_1, \ldots, t'_m\to s'_m\}$, and $t' = \bigwedge_{i=1..n} t_i\to s_i \setminus \bigvee_{j=1..m} t'_j\to s'_j \not\simeq \mathbb{0}$. We have $t' \leq t$ and the value $v = \mu f(t_1\to s_1; \ldots; t_n\to s_n).\lambda x.fx$ has type t' (direct application the typing rule for abstractions). By subsumption, we get $\vdash v : t$. □

Lemmas 6.27 and 6.26 entail Theorem 5.5.

6.6. TYPE SOUNDNESS. Here is the proof of the subject reduction property, Theorem 5.1 in Section 5.

PROOF. If $(\Gamma \vdash e : t)$, then we prove by induction on the derivation for $\Gamma \vdash e : t$ that $\forall e'.(e \rightsquigarrow e') \Rightarrow (\Gamma \vdash e' : t)$. We consider the last rule used in the derivation of $\Gamma \vdash e : t$.

Rule (*subsum*): We have $\Gamma \vdash e : s \leq t$ and $e \rightsquigarrow e'$. The induction hypothesis gives $\Gamma \vdash e' : s$, and by subsumption we get $\Gamma \vdash e' : t$.

Rules (*const*),(*var*): The expression e is a constant or a variable. It cannot be reduced.

Rule (*proj*): We have $e = \pi_i(e_0)$, $t = t_i$, $\Gamma \vdash e_0 : t_1\times t_2$. If e' is obtained by reducing e_0, that is, $e_0 \rightsquigarrow e'_0$ and $e' = \pi_i(e'_0)$, we get, by the induction hypothesis: $\Gamma \vdash e'_0 : t_1\times t_2$ and thus $\Gamma \vdash e' : t_i$. If e' is obtained by reducing the toplevel π_i in e, then necessarily e_0 is a value (v_1, v_2) (and thus, by Lemma 6.21: $\Gamma \vdash v_i : t_i$), and $e' = v_i$. We get $\Gamma \vdash e' : t_i$.

Rule (*rnd*): We have $e = \mathtt{rnd}(t)$. The reduction rule for this expression gives $\vdash e' : t$, which implies $\Gamma \vdash e' : t$ by Lemma 6.14.

Rule (*pair*): We have $e = (e_1, e_2)$, $t = t_1 \times t_2$, and $\Gamma \vdash e_i : t_i$ for $i = 1..2$. The only possible way to reduce e is to reduce one of the e_i, say $e' = (e'_1, e_2)$ where $e_1 \rightsquigarrow e'_1$. The induction hypothesis gives $\Gamma \vdash e'_1 : t_1$, and we get $\Gamma \vdash e' : t_1 \times t_2$.

Rule (*appl*): We have $e = e_1 e_2$, $\Gamma \vdash e_1 : s \to t$ and $\Gamma \vdash e_2 : s$. If e' is obtained by reducing e_1 or e_2, we proceed as in the case for the (*pair*) rule. Otherwise, we have necessarily $e_1 = \mu f(s_1 \to t_1; \cdots; s_n \to t_n).\lambda x.e_0$, $e' = e_0[f := e_1; x := e_2]$ and e_2 is a value v_2. We have $\bigwedge_{i \in I} s_i \to t_i \leq s \to t$, where $I = \{1, \ldots, n\}$ (Lemma 6.21). In accordance with Lemma 6.8, this means that $s \leq \bigvee_{i \in I} s_i$ and that for any nonempty $I' \subseteq I$ such that $s \not\leq \bigvee_{i \in I \setminus I'} s_i$, we have $\bigwedge_{i \in I'} t_i \leq t$. We take $I' = \{i \in I \mid \vdash v_2 : s_i\}$. This set is not empty. Indeed, since $\vdash v_2 : s$ and $s \leq \bigvee_{i \in I} s_i$, we have at least one i such that $\vdash v_2 : s_i$ (Lemma 6.23). Now, we claim that $s \not\leq \bigvee_{i \in I \setminus I'} s_i$. Otherwise, we would find some $i \notin I'$ such that $\vdash v_2 : s_i$, which contradicts the definition for I'. As a consequence, we get $\bigwedge_{i \in I'} t_i \leq t$. We claim that $\Gamma \vdash e' : \bigwedge_{i \in I'} t_i$ (which, by subsumption, yields $\Gamma \vdash e' : t$, i.e., the result). To prove our claim, we show that for every $i \in I'$ we have $\Gamma \vdash e' : t_i$, which, thanks to Lemma 6.15, yields our claim. So, let us consider any $i \in I'$, that is, any i such that $\vdash v_2 : s_i$. The abstraction e_1 is well typed under Γ; therefore, in its derivation, there is an instance of the (*abstr*) rule (possibly followed by several applications of the subsumption rule), which infers for e_1 a type t' under Γ. One of the premises of this rule is $(f : t'), (x : t_i), \Gamma \vdash e_0 : t_i$. We also have $\Gamma \vdash e_1 : t'$ and $\Gamma \vdash v_2 : s_i$ (Lemma 6.14), and thus $\Gamma \vdash e' : t_i$ (Lemma 6.17) as expected.

Rule (*abstr*): The expression e is an abstraction, and the reduction can only occur within its body. We proceed as in the case for the (*pair*) rule.

Rule (*case*): We have $e = (x = e_0 \in s \ ? \ e_1 \mid e_2)$. If the reduction occurs within one of the subexpressions e_0,e_1,e_2, we proceed as in the case for the (*pair*) rule. Otherwise, the expression e_0 is necessarily a value v, and we have either $(\vdash v : s) \wedge (e' = e_1[x := v])$ or $(\vdash v : \neg s) \wedge (e' = e_2[x := v])$. Let us consider for instance the first case. The typing rule gives: $\Gamma \vdash v : s_0$. Thanks to Lemma 6.15, we get $\Gamma \vdash v : s_0 \wedge s$. Because of Lemma 6.19, we know that $s_0 \wedge s \not\simeq \mathbb{0}$, that is $s_0 \not\leq \neg s$. So the typing rule (*case*) under consideration has a premise for e_1, namely $(x : s_0 \wedge s), \Gamma \vdash e_1 : t$. Lemma 6.17 gives $\Gamma \vdash e' : t$ as expected. □

And here is the proof of the progress property, Theorem 5.2 in Section 5. Note that this proof is relatively standard.

PROOF. We write $e \not\rightsquigarrow$ if e cannot be reduced ($\nexists e'.e \rightsquigarrow e'$). Suppose that $\vdash e : t$; we prove on induction on the derivation of $\vdash e : t$ that either e is a value or it can be reduced. We consider the last rule used in this derivation.

Rule (*subsum*): Straightforward application of the induction hypothesis.

Rule (*var*): A variable cannot be welltyped in an empty environment. This case is thus impossible.

Rule (*const*): The expression e is a constant. It is thus a value.

Rule (*abstr*): The expression e is an abstraction that is well typed under the empty environment. It is thus a value.

Rule (*proj*): We have $e = \pi_i(e_0)$, $t = t_i$, $\vdash e_0 : t_1 \times t_2$. If e_0 can be reduced to, say, e'_0, then $e \rightsquigarrow \pi_i(e'_0)$. Otherwise, if $e_0 \not\rightsquigarrow$, then by the induction hypothesis e_0 is a value. By Lemma 6.21, we get $e_0 = (v_1, v_2)$, and thus $e \rightsquigarrow v_i$.

Rule (*rnd*): We have $e = \mathtt{rnd}(t)$ and thus $e \rightsquigarrow e'$ for any e' of type t (e.g., we can take for e' an expression of type $\mathbb{0}$, which exists).

Rule (*pair*): We have $e = (e_1, e_2)$, $t = t_1 \times t_2$, and $\vdash e_i : t_i$ for $i = 1..2$. If one of the e_i can be reduced, then e can also be reduced. Otherwise, by the induction hypothesis, we obtain that both e_1 and e_2 are values, and so is e.

Rule (*appl*): We have $e = e_1 e_2$, $\vdash e_1 : s \to t$ and $\vdash e_2 : s$. If one of the e_i can be reduced, then e can also be reduced. Otherwise, by the induction hypothesis, we obtain that both e_1 and e_2 are values. By Lemma 6.21, we get $e_1 = \mu f(s_1 \to t_1; \cdots; s_n \to t_n).\lambda x.e_0$. Then, $e \rightsquigarrow e_0[f := e_1; x := e_2]$.

Rule (*case*): We have $e = (x = e_0 \in s \; ? \; e_1 \mid e_2)$. If e_0 can be reduced, then e can also be reduced. Otherwise, by the induction hypothesis, we obtain that e_0 is a value v. Because of Lemma 6.23, we have $\vdash v : s$ or $\vdash v : \neg s$, and thus $e \rightsquigarrow e_1[x := v]$ or $e \rightsquigarrow e_2[x := v]$. □

6.7. CONSTRUCTION OF MODELS. A naive idea to build a model would be to look for an interpretation domain D such that $D = \mathbb{E}D$. Of course such a set cannot exist, since the cardinality of $\mathbb{E}^{\mathbf{fun}}D$, and thus of $\mathbb{E}D$, is strictly larger than the cardinality of D. This cardinality problem can be avoided by considering only finite graphs to interpret functions. As we will show below, this does not affect the subtyping relation.

For any set D, we write $\mathbb{E}_f D = \mathcal{C} + D^2 + \mathcal{P}_f(D \times D_\Omega)$ where $\mathcal{P}_f$ denotes the restriction of the powerset to finite subsets.

Definition 6.28. A set-theoretic interpretation $[\![\_]\!] : \mathcal{T} \to \mathcal{P}(D)$ is ***finitely extensional*** if:

(1) $D = \mathbb{E}_f D$

(2) $[\![a]\!] = \mathbb{E}(a) \cap D$ for any atom a.

LEMMA 6.29. *If $[\![\_]\!]$ is a finitely extensional set-theoretic interpretation, then $[\![t]\!] = \mathbb{E}(t) \cap D$ for any type t, and $[\![\tau]\!] = \mathbb{E}(\tau) \cap D$ for any normal formal τ.*

PROOF. Induction on t. □

The next lemma shows that taking finite sets as extensional models for functions does not change the subtyping relation between arrow types (compare it with Lemma 6.7).

LEMMA 6.30. *Let $(X_i)_{i \in P}$ and $(X_i)_{i \in N}$ be two finite families of subsets of D. Then:*

$$\bigcap_{i \in P} \mathcal{P}_f(X_i) \subseteq \bigcup_{i \in N} \mathcal{P}_f(X_i) \iff \exists i_o \in N. \bigcap_{i \in P} X_i \subseteq X_{i_0}$$

PROOF. The $\Leftarrow$ implication is straightforward. Let us prove $\Rightarrow$. We assume that any finite subset of $X = \bigcap_{i \in P} X_i$ is a subset of one of the X_{i_0} with $i_0 \in N$. We need to prove that the same holds for X itself. Otherwise, we could find for each $i_0 \in N$ an element $x_{i_0} \in X \backslash X_{i_0}$ and we would obtain a contradiction by considering the finite set $\{x_{i_0} \mid i_0 \in N\}$. □

LEMMA 6.31. *Let P, N be two finite sets of arrow types and $[\![\_]\!]$ an arbitrary set-theoretic interpretation. Then:*

$$\bigcap_{a\in P}\mathbb{E}(a)\subseteq\bigcup_{a\in N}\mathbb{E}(a)\iff\mathcal{P}_f(D\times D_\Omega)\cap\bigcap_{a\in P}\mathbb{E}(a)\subseteq\bigcup_{a\in N}\mathbb{E}(a)$$

(By convention $\bigcap_{a\in\varnothing}\mathbb{E}(a)=\mathcal{P}(D\times D_\Omega)$.)

PROOF. Consequence of Lemmas 6.7, 6.30, and 6.6. □

It is, then, not surprising that finitely extensional interpretations are models.

LEMMA 6.32. *Every finitely extensional interpretation is a model.*

PROOF. Since $[\![\tau]\!]=\mathbb{E}(\tau)\cap D$, we need to prove that

$$\mathbb{E}(\tau)=\varnothing\iff\mathbb{E}(\tau)\cap D=\varnothing$$

for any normal form τ. We write:

$$\mathbb{E}(\tau)=\bigcup_{u\in U}\bigcup_{(P,N)\in\tau}\left(\mathbb{E}^u D\cap\bigcap_{a\in P}\mathbb{E}(a)\setminus\bigcup_{a\in N}\mathbb{E}(a)\right).$$

So we need to prove that for any $u\in U$ and (P,N) two finite sets of atoms, we have:

$$\mathbb{E}^u D\cap\bigcap_{a\in P}\mathbb{E}(a)\subseteq\bigcup_{a\in N}\mathbb{E}(a)\iff D\cap\mathbb{E}^u D\cap\bigcap_{a\in P}\mathbb{E}(a)\subseteq\bigcup_{a\in N}\mathbb{E}(a).$$

If $u\neq\mathbf{fun}$, then $\mathbb{E}^u D\subseteq D$, and the equivalence is thus trivial. The case $u=\mathbf{fun}$ comes from Lemma 6.31. □

6.8. A UNIVERSAL MODEL. In this section, we define a structural and finitely extensional model and then show that it is universal and, in the next section, that the subtyping relation induced by this model is decidable.

We need to build a set D^0 such that $D^0=\mathbb{E}_f D^0$, that is, a solution to the equation $D^0=\mathcal{C}+D^0\times D^0+\mathcal{P}_f(D^0\times D^0_\Omega)$. We will consider the *initial* solution to this equation. Concretely, we define D^0 as the set of finite terms generated by the production d of the following grammar (c ranges over elements of $\mathcal{C}$):

$$\begin{aligned}d &::= c\mid(d,d)\mid\{(d,d'),\ldots,(d,d')\}\\ d' &::= d\mid\Omega.\end{aligned}$$

Now, we need to define a set-theoretic interpretation $[\![\_]\!]^0:\mathcal{T}\to\mathcal{P}(D^0)$ such that $[\![t]\!]^0=\mathbb{E}(a)^0\cap D^0$. Because of the inductive structure of elements of D^0, this equation actually defines the function $[\![\_]\!]^0$. To see this, we will define a binary predicate $(d:t)$ where $d\in D^0$ and $t\in\mathcal{T}$. The truth value of $(d:t)$ is defined by induction on the pair (d,t) ordered lexicographically, using the inductive structure for elements of D^0, and the induction principle we mentioned earlier for types.

Here is the definition:

$$\begin{array}{lll}
(c : b) & = c \in \mathbb{B}[\![b]\!] & \\
((d_1, d_2) : t_1 \times t_2) & = (d_1 : t_1) \wedge (d_2 : t_2) & \\
(\{(d_1, d_1'), \ldots, (d_n, d_n')\} : t_1 \to t_2) & = \forall i.\, (d_i : t_1) \Rightarrow (d_i' : t_2) & \\
(d : t_1 \vee t_2) & = (d : t_1) \vee (d : t_2) & \\
(d : \neg t) & = \neg(d : t) & \\
(d : t) & = \texttt{false} & \text{otherwise.}
\end{array}$$

Now we define $[\![t]\!]^0 = \{d \in D^0 \mid (d : t)\}$. It is straightforward from this definition to see that $[\![\_]\!]^0$ is a set-theoretic interpretation and that it is structural (and thus well founded). It is also clear that it is finitely extensional. It is thus a model. It remains to prove that this model is universal. This is a direct consequence of the next lemma.

LEMMA 6.33. *If $\mathcal{S}^0 = \{\tau \mid [\![\tau]\!]^0 = \varnothing\}$ and $\mathcal{S}$ is a simulation, then $\mathcal{S} \subseteq \mathcal{S}^0$.*

PROOF. Let $\mathcal{S}$ be a simulation. We need to prove that $\forall \tau \in \mathcal{S}.\, [\![\tau]\!]^0 = \varnothing$, that is:

$$\forall d \in D^0. \forall \tau \in \mathcal{S}.\, d \notin [\![\tau]\!]^0.$$

We will prove this property by induction on $d \in D^0$. Let's take $d \in D^0$ and $\tau \in \mathcal{S}$. Since $\mathcal{S}$ is a simulation, we also have $\tau \in \mathbb{E}\mathcal{S}$, that is:

$$\forall u \in U. \forall (P, N) \in t.\, \left(P \subseteq \mathcal{A}_u \Rightarrow C_u^{P, N \cap \mathcal{A}_u}\right), \tag{6}$$

where the conditions $C_u^{P,N}$ are as in Definition 6.9.

We need to prove that $d \notin [\![\tau]\!]^0$. The set $[\![\tau]\!]^0$ is equal to:

$$\bigcup_{(P,N)\in\tau} \bigcap_{a\in P} [\![a]\!]^0 \setminus \bigcup_{a\in N} [\![a]\!]^0.$$

We prove that d does not belong to any of the terms of this union. Let $(P, N) \in \tau$ and u be the kind of d (as for values, it is straightforward to associate a unique kind to each element of D^0). If $a \in \mathcal{A} \setminus \mathcal{A}_u$, then clearly $d \notin [\![a]\!]^0$. As a consequence, if $P \not\subseteq \mathcal{A}_u$, then $d \notin \bigcap_{a\in P} [\![a]\!]^0 \setminus \bigcup_{a\in N} [\![a]\!]^0$. We now assume that $P \subseteq \mathcal{A}_u$. We can apply (6). We obtain that $C_u^{P, N\cap\mathcal{A}_u}$ holds. It remains to prove that:

$$d \notin \bigcap_{a\in P} [\![a]\!]^0 \setminus \bigcup_{a\in N\cap\mathcal{A}_u} [\![a]\!]^0$$

$\underline{u = \textbf{basic}, d = c}$. The condition $C_u^{P, N\cap\mathcal{A}_u}$ is:

$$\mathcal{C} \cap \bigcap_{b\in P} \mathbb{B}[\![b]\!] \subseteq \bigcup_{b\in N} \mathbb{B}[\![b]\!].$$

As a consequence, we get:

$$d \notin \bigcap_{b\in P} \mathbb{B}[\![b]\!] \setminus \bigcup_{b\in N} \mathbb{B}[\![b]\!] = \bigcap_{a\in P} [\![a]\!]^0 \setminus \bigcup_{a\in N\cap\mathcal{A}_{\textbf{basic}}} [\![a]\!]^0$$

$\underline{u = \mathbf{prod}, d = (d_1, d_2)}$. The condition $C_u^{P,N\cap\mathcal{A}_u}$ is:

$$\forall N' \subseteq N \cap \mathcal{A}_{\mathbf{prod}}. \begin{cases} \mathcal{N}\left(\bigwedge_{t_1\times t_2\in P} t_1 \wedge \bigwedge_{t_1\times t_2\in N'} \neg t_1\right) \in \mathcal{S} \\ \vee \\ \mathcal{N}\left(\bigwedge_{t_1\times t_2\in P} t_2 \wedge \bigwedge_{t_1\times t_2\in N\setminus N'} \neg t_2\right) \in \mathcal{S}. \end{cases}$$

For each N', we apply the induction hypothesis to d_1 and to d_2. We get:

$$d_1 \notin \left[\!\!\left[\bigwedge_{t_1\times t_2\in P} t_1 \wedge \bigwedge_{t_1\times t_2\in N'} \neg t_1 \right]\!\!\right]^0 \vee d_2 \notin \left[\!\!\left[\bigwedge_{t_1\times t_2\in P} t_2 \wedge \bigwedge_{t_1\times t_2\in N\setminus N'} \neg t_2 \right]\!\!\right]^0$$

That is:

$$d \notin \left(\bigcap_{t_1\times t_2\in P} [\![t_1]\!]^0 \setminus \bigcup_{t_1\times t_2\in N'} [\![t_1]\!]^0 \right) \times \left(\bigcap_{t_1\times t_2\in P} [\![t_2]\!]^0 \setminus \bigcup_{t_1\times t_2\in N\setminus N'} [\![t_2]\!]^0 \right).$$

In accordance with Lemma 6.4 and to $[\![t_1]\!]^0 \times [\![t_2]\!]^0 = [\![t_1\times t_2]\!]^0$, we thus get:

$$d \notin \bigcap_{a\in P} [\![a]\!]^0 \setminus \bigcup_{a\in N\cap\mathcal{A}_{\mathbf{prod}}} [\![a]\!]^0$$

$\underline{u = \mathbf{fun}, d = \{(d_1, d_1'), \ldots, (d_n, d_n')\}}$. The condition $C_u^{P,N\cap\mathcal{A}_u}$ says that there exists $t_0\to s_0 \in N$ such that, for all $P' \subseteq P$:

$$\mathcal{N}\left(t_0 \wedge \bigwedge_{t\to s\in P'} \neg t\right) \in \mathcal{S} \vee \begin{cases} P \neq P' \\ \mathcal{N}\left((\neg s_0) \wedge \bigwedge_{t\to s\in P\setminus P'} s\right) \in \mathcal{S}. \end{cases}$$

Applying the induction hypothesis to the d_i and d_i' (note that if $d_i' = \Omega$, then $d_i' \notin [\![\tau]\!]^0$ is trivial for all τ):

$$d_i \notin \left[\!\!\left[t_0 \wedge \bigwedge_{t\to s\in P'} \neg t \right]\!\!\right]^0 \vee \begin{cases} P \neq P' \\ d_i' \notin \left[\!\!\left[(\neg s_0) \wedge \bigwedge_{t\to s\in P\setminus P'} s \right]\!\!\right]^0. \end{cases}$$

Let us first assume that $\forall i.\ (d_i \in [\![t_0]\!]^0 \Rightarrow d_i' \in [\![s_0]\!]^0)$. Then, we have $d \in [\![t_0\to s_0]\!]^0$. Otherwise, let us consider i such that $d_i \in [\![t_0]\!]^0$ and $d_i' \notin [\![s_0]\!]^0$. The formula above gives for any $P' \subseteq P$:

$$\left(d_i \in \bigcup_{t\to s\in P'} [\![t]\!]^0 \right) \vee \left(P' \neq P \wedge d_i' \in \{\Omega\} \cup \bigcup_{t\to s\in P\setminus P'} [\![\neg s]\!]^0 \right).$$

Let's take $P' = \{t\to s \in P \mid d_i \notin [\![t]\!]^0\}$. We have $d_i \notin \bigcup_{t\to s\in P'} [\![t]\!]^0$, and thus $P' \neq P$ and $d_i' \in \{\Omega\} \cup \bigcup_{t\to s\in P\setminus P'} [\![\neg s]\!]^0$. We can thus find $t\to s \in P\setminus P'$ such that

$d'_i \notin [\![s]\!]^0$, and because $t \to s \notin P'$, we also have $d_i \in [\![t]\!]^0$. We have thus proved that $d \notin [\![t \to s]\!]^0$ for some $t \to s \in P$.

In both cases, we get:

$$d \notin \bigcap_{a \in P} [\![a]\!]^0 \setminus \bigcup_{a \in N \cap \mathcal{A}_{\mathbf{fun}}} [\![a]\!]^0. \qquad \square$$

6.9. DECIDABILITY OF SUBTYPING FOR THE UNIVERSAL MODEL. We will now focus on Theorem 5.8. Let $\leq_0$ denote the subtyping relation induced by the universal model $[\![\_]\!]^0$. We have $t_1 \leq_0 t_2 \iff [\![t_1 \setminus t_2]\!]^0 = \varnothing \iff [\![\mathcal{N}(t_1 \setminus t_2)]\!]^0 = \varnothing$. Therefore, we need to show how to decide, for a given normal form τ_0, whether $[\![\tau_0]\!]^0 = \varnothing$ or not. Thanks to the lemma above, we get: $[\![\tau_0]\!]^0 = \varnothing$ if and only if there exists a simulation $\mathcal{S}$ such that $\tau_0 \in \mathcal{S}$.

Actually, we can restrict our attention to a finite number of normal forms. Indeed, let us consider the set A of all the atoms that occur in τ_0 (including atoms nested in other atoms). Thanks to the regularity of types, this set A is finite. Write $\mathcal{N}(A)$ for the set of normal forms built only on top of these atoms, that is: $\mathcal{N}(A) = \mathcal{P}(\mathcal{P}(A) \times \mathcal{P}(A))$. This set is also finite, and looking at Definition 6.9, we see that an intersection of a simulation and $\mathcal{N}(A)$ is again a simulation. As a consequence, we get: $[\![\tau_0]\!]^0 = \varnothing$ if and only if there exists a simulation $\mathcal{S} \subseteq \mathcal{N}(A)$ such that $\tau_0 \in \mathcal{S}$. A naive algorithm can simply enumerate all the subset of $\mathcal{N}(A)$ that contain τ_0 and by applying Definition 6.9 check whether one of them is a simulation.

Of course, there exist better algorithms. For instance, we can interpret the definition of a simulation as saturation rules: the algorithm starts from the set $\{\tau_0\}$ and tries to saturate it until it obtains a simulation. Because of the disjunctions in the definition of a simulation, this algorithm needs to explore different branches. A branch cannot be infinite because the algorithm will only consider the normal forms in $\mathcal{N}(A)$, which is a finite set. There exists a simulation that contains τ_0 if and only if one of the branches succeeds in reaching a simulation. The Ph.D. dissertation [Frisch 2004] describes two algorithms that improve over this simple saturation-based strategy. These algorithms are those implemented in the CDuce compiler [CDUCE] and, as such, they are tested daily on large and complex types such as the XHTML DTD.

6.10. NONUNIVERSAL MODELS. The interpretation domain D of a finitely extensional set-theoretic interpretation must be a solution to the equation $D = \mathbb{E}_f D$. In the previous section, we considered the initial solution to this equation and we obtained a universal model. In this section, we will build nonuniversal models by considering noninitial solutions to the equation $D = \mathbb{E}_f D$.

A first attempt could be to consider *infinite* (or maybe regular) terms generated by the following productions:

$$\begin{aligned} d &::= c \mid (d, d) \mid \{(d, d'), \ldots, (d, d')\} \\ d' &::= d \mid \Omega. \end{aligned}$$

But it is then impossible to build a finitely extensional interpretation on this domain D^∞. Indeed, if $[\![\_]\!]$ is such an interpretation, we consider the element $d \in D^\infty$ such that $d = (d, d)$ and the type t such that $t = (\neg t) \times (\neg t)$. Since $d \in D^\infty$ and $[\![t]\!] = \mathbb{E}(t) \cap D^\infty = (D^\infty \setminus [\![t]\!]) \times (D^\infty \setminus [\![t]\!])$, we have: $d \in [\![t]\!] \iff (d, d) \in (D^\infty \setminus [\![t]\!]) \times (D^\infty \setminus [\![t]\!]) \iff d \notin [\![t]\!]$. Contradiction.

So, we will build domains that are intermediate between D^0 and D^∞. We need to introduce some new notions.

For an arbitrary set X, we define $D^{[X]}$ as the set of finite terms generated by the production d below:

$$\begin{aligned} d &::= x \mid c \mid (d, d) \mid \{(d, d'), \ldots, (d, d')\} \\ d' &::= d \mid \Omega, \end{aligned}$$

where x ranges over elements of X. In other words, $D^{[X]}$ is the initial solution D to the equation $D = X + \mathcal{C} + D^2 + \mathcal{P}_f(D \times D_\Omega)$. We define the predicate $\Delta \vdash d : t$ for $d \in D^{[X]}$, $t \in \mathcal{T}$, $\Delta \in \mathcal{P}(\mathcal{T})^X$ by induction on the structure of d:

$$\begin{aligned} (\Delta \vdash d : t_1 \vee t_2) &= (\Delta \vdash d : t_1) \vee (\Delta \vdash d : t_2) \\ (\Delta \vdash d : \neg t) &= \neg(\Delta \vdash d : t) \\ (\Delta \vdash c : b) &= c \in \mathbb{B}[\![b]\!] \\ (\Delta \vdash (d_1, d_2) : t_1 \times t_2) &= (\Delta \vdash d_1 : t_1) \wedge (\Delta \vdash d_2 : t_2) \\ (\Delta \vdash \{(d_1, d'_1), \ldots, (d_n, d'_n)\} : t_1 \to t_2) &= \forall i.\, (\Delta \vdash d_i : t_1) \Rightarrow (\Delta \vdash d'_i : t_2) \\ (\Delta \vdash x : a) &= a \in \Delta(x) \\ (\Delta \vdash d : t) &= \texttt{false} \qquad \text{otherwise.} \end{aligned}$$

A congruence on $D^{[X]}$ is an equivalence relation $\equiv$ such that $(d_1^1 \equiv d_1^2 \wedge d_2^1 \equiv d_2^2) \Rightarrow (d_1^1, d_2^1) \equiv (d_1^2, d_2^2)$ and $(\forall i. d_i^1 \equiv d_i^2 \wedge d_i'^1 \equiv d_i'^2) \Rightarrow \{(d_1^1, d_1'^1), \ldots\} \equiv \{(d_1^2, d_1'^2), \ldots\}$. If, for all x, we choose an element $d^x \in \mathbb{E}_f(D^{[X]}) = D^{[X]} \backslash X$ and if we consider the smallest congruence $\equiv$ such that $\forall x \in X. x \equiv d^x$, then the quotient $D^{[X]}_\equiv = D^{[X]}/\equiv$ is such that $\mathbb{E}_f(D^{[X]}_\equiv) = D^{[X]}_\equiv$ (modulo an implicit bijection). Note that this quotient looks a lot like D^0, except that there are some non–well-founded elements. Let's choose some $\Delta \in \mathcal{P}(\mathcal{T})^X$. We require the predicate $(\Delta \vdash d : t)$ to be invariant under $\equiv$, that is: $d^1 \equiv d^2 \Rightarrow ((\Delta \vdash d^1 : t) \iff (\Delta \vdash d^2 : t))$. This is the case if and only if $\forall x.(\Delta \vdash x : t) \iff (\Delta \vdash d^x : t)$, that is, if and only if:

$$(*)\ \ \forall x \in X.\ \Delta(x) = \{t \mid \Delta \vdash d^x : t\}$$

When this property holds, we can define $[\![\_]\!]_\Delta : \mathcal{T} \to \mathcal{P}(D^{[X]}_\equiv)$ by $[\![t]\!]_\Delta = \{[d]_\equiv \mid (\Delta \vdash d : t)\}$, where $[d]_\equiv$ denotes the equivalence class of d modulo $\equiv$. This defines a finitely extensional set-theoretic interpretation (and thus a model).

Of course, the difficulty is now to choose X, the d^x and Δ such that $(*)$ holds. Let us consider the case where $X = \mathbb{Z}$, and each $d^k, k \in \mathbb{Z}$ is defined using only d^{k-1} in a uniform way. Formally, we consider a fixed element $\delta \in D^{\{\bullet\}}$ such that $\delta \neq \bullet$ and we define $d^k = \delta[\bullet := k - 1]$ (i.e., the element of $D^\mathbb{Z}$ obtained by substituting $\bullet$ by $k - 1$ in δ). If $\Delta \in \mathcal{P}(\mathcal{T})^\mathbb{Z}$, then $\Delta \vdash d^k : t$ is equivalent to $\Delta \vdash \delta[\bullet := k - 1] : t$, and an induction on the structure of δ shows that this is equivalent to $(\bullet \mapsto \Delta_{k-1}) \vdash \delta : t$ (from now on, we write Δ_k instead of $\Delta(k)$). If we define the operator $F : \mathcal{P}(\mathcal{T}) \to \mathcal{P}(\mathcal{T})$ by $F(T) = \{t \mid (\bullet \mapsto T) \vdash \delta : t\}$, then the condition $(*)$ can be rewritten as:

$$\forall k \in \mathbb{Z}.\ \Delta_k = F(\Delta_{k-1}).$$

Building such a sequence is not straightforward. We will rely on a technical lemma.

LEMMA 6.34. *Let A be a finite set, $f : A \to A$, and $a^0 \in A$. There exists a unique periodic sequence $(a_k)_{k\in\mathbb{Z}} \in A^{\mathbb{Z}}$ such that:*

$$\exists n_0 \in \mathbb{N}.\forall k \geq n_0.a_k = f^k(a^0)$$

(where f^n denotes the nth iterated composition of f with itself). This sequence is such that:

$$\forall k.\ a_{k+1} = f(a_k).$$

PROOF. We consider the sequence $(a^n)_{n\in\mathbb{N}}$ defined by $a^n = f^n(a^0)$. Since A is finite, this sequence cannot be injective. We can find $n_0 < n_1$ such that $a^{n_0} = a^{n_1}$. A recurrence gives $a^n = a^{n+(n_1-n_0)}$ for any $n \geq n_0$: the sequence $(a^n)_{n\in\mathbb{N}}$ is ultimately periodic. As a consequence, there exists a unique sequence $(a_k)_{k\in\mathbb{Z}}$ that coincides ultimately with $(a^n)_{n\in\mathbb{N}}$.

Clearly, the property $a_{k+1} = f(a_k)$ holds for k large enough, and because $(a_k)_{k\in\mathbb{Z}}$ is periodic, it holds for any k. □

THEOREM 6.35. *Let T^0 be a set of types. There exists a sequence $(\Delta_k)_{k\in\mathbb{Z}}$ such that:*

- $\forall k \in \mathbb{Z}.\Delta_{k+1} = F(\Delta_k)$
- *For any type t, the sequence of the truth values of $(t \in \Delta_k)_{k\in\mathbb{Z}}$ is periodic and $\exists n_0 \in \mathbb{N}.\forall k \geq n_0.(t \in \Delta_k \iff t \in F^k(T^0))$.*

PROOF. Since the set $\mathcal{P}(\mathcal{T})$ is not finite, we cannot use the lemma directly. The regularity of types will come to the rescue. We define a cone as a *finite* set of types that is closed under subterms decomposition (i.e., if the set contains a type, it also contains all its subterms). Any type belongs to some cone because a type is a regular term. For a cone C, we can define the function $F_C : \mathcal{P}(C) \to \mathcal{P}(C)$ by $F_C(T) = F(T) \cap C$. We can apply the lemma to this function, because $\mathcal{P}(C)$ is finite. We write $(T_k^C)_{k\in\mathbb{Z}}$ for the sequence we obtain. Now, we observe on the definition of the $\vdash$ predicate that for $t \in C$, the assertion $(\bullet \mapsto T) \vdash \delta : t$ holds if and only if $(\bullet \mapsto (T \cap C)) \vdash \delta : t$ holds. This gives immediately the following property:

$$\forall T \subseteq \mathcal{T}.\ C \cap F(T \cap C) = C \cap F(T)$$

From that, a recurrence gives $F_C^n(T^0) = F^n(T^0) \cap C$. So, for $t \in C$, we have $t \in T_k^C \iff t \in F^k(T_0)$ when k is large enough. Since the sequence $(t \in T_k^C)_{k\in\mathbb{Z}}$ is periodic, it does not depend on the choice of the cone C which contains t. We can thus define Δ_k as the set of types t such that $t \in T_k^C$ for some/any cone C that contains t. We have $T_k^C = \Delta_k \cap C$. It remains to check that $\Delta_{k+1} = F(\Delta_k)$ for all k. Let t be a type and C a cone that contains t. We have $t \in \Delta_{k+1} \iff t \in T_{k+1}^C$ and, in accordance with the lemma, we have $T_{k+1}^C = F(T_k^C) \cap C = F(\Delta_k) \cap C$. So: $t \in \Delta_{k+1} \iff t \in F(\Delta_k)$. Since this property holds for an arbitrary t, we get $\Delta_{k+1} = F(\Delta_k)$ as expected. □

We will give two examples of constructions based on this theorem. First, we will build a model that is not well founded. In a well founded model, the recursive type $t_0 = t_0 \times t_0$ is empty. We will build a model where this type is not empty. We take $\delta = (\bullet, \bullet)$ and we build $(\Delta_k)_{k\in\mathbb{Z}}$ as given by the theorem. We thus get a finitely extensional set-theoretic interpretation $[\![\_]\!]_\Delta : \mathcal{T} \to \mathcal{P}(D_{\equiv}^{\mathbb{Z}})$. For any set of types

T, we have $t_0 \in F(T) \iff (\bullet \mapsto T) \vdash \delta : t_0 \iff (\bullet \mapsto T) \vdash (\bullet, \bullet) : t_0 \times t_0 \iff (\bullet \mapsto T) \vdash \bullet : t_0 \iff t_0 \in T$. So if we choose T^0 such that $t_0 \in T^0$, we have $t_0 \in \Delta_k$ for all k, from which we conclude that $[\![t_0]\!]_\Delta$ contains the $[k]_\equiv$ for $k \in \mathbb{Z}$. In particular, it is not empty. To better understand our construction, we can consider the type $t_1 = (\neg t_1) \times (\neg t_1)$. We find that $t_1 \in F(T) \iff t_1 \notin T$ and we deduce that $[\![t_1]\!]_\Delta$ contains the $[k]_\equiv$ for all even $k \in \mathbb{Z}$ (if $t_1 \in T^0$) or for all $k \in \mathbb{Z}$ (if $t_1 \notin T^0$). For more complex recursive types, we might see other periods than 2.

Now, we will build a structural (and thus well-founded) model that is not universal. We consider the recursive type $t_0 = (\mathbb{0} \to \mathbb{0}) \setminus (t_0 \to \mathbb{0})$. If $[\![\_]\!]$ is a finitely extensional set-theoretic interpretation, a simple computation gives:

$$[\![t_0]\!] = \{\{(d_i, d_i') \mid \exists i.\ d_i \in [\![t_0]\!]\}\}.$$

In particular, this set is empty for the universal model built in the previous section (because its elements are finite trees). We take $\delta = \{(\bullet, \Omega)\}$ and we proceed as above, with the following computation: $t_0 \in F(T) \iff (\bullet \mapsto T) \vdash \delta : t_0 \iff (\bullet \mapsto T) \vdash \{(\bullet, \Omega)\} : (\mathbb{0} \to \mathbb{0}) \setminus (t_0 \times \mathbb{0}) \iff (\bullet \mapsto T) \vdash \bullet : t_0 \iff t_0 \in T$. We conclude by taking T^0 such that $t_0 \in T^0$ that the model $[\![\_]\!]_\Delta$ is not universal. It remains to see that it is structural. The decomposition relation $\triangleright$ is defined by $([d_1]_\equiv, [d_2]_\equiv) \triangleright [d_i]_\equiv$. Because of the definition of δ, if $[d]_\equiv \triangleright [d']_\equiv$, then d must be a pair (d_1, d_2) in $D^Z \times D^Z$. As a consequence, the relation $\triangleright$ is Noetherian.

6.11. TOWARDS TYPE-CHECKING. In this section, we introduce notions that will be useful to derive a type-checking algorithm. We also give the proof of Theorem 5.3 (local exactness of the application rule). The existence results in this section are *effective* (viz. it is possible to compute the objects whose existence is asserted) provided that the subtyping relation is decidable.

LEMMA 6.36. *Let t be a type such that $t \leq \mathbb{1} \times \mathbb{1}$. There exists a finite set of pairs of types $\pi(t) \in \mathcal{P}_f(\mathcal{T}^2)$ such that:*

—$t \simeq \bigvee_{(t_1,t_2)\in\pi(t)} t_1 \times t_2$

—$\forall (t_1, t_2) \in \pi(t).\ t_1 \not\simeq \mathbb{0} \wedge t_2 \not\simeq \mathbb{0}$

PROOF. We can write:

$$t \simeq \bigvee_{(P,N)\in\mathcal{N}(t) \text{ such that } P\subseteq\mathcal{A}_{\mathbf{prod}}} (\mathbb{1}\times\mathbb{1}) \wedge \bigwedge_{a\in P} a \setminus \bigvee_{a\in N\cap\mathcal{A}_{\mathbf{prod}}} a$$

Using Lemma 6.4, we can rewrite any intersection of product types and complement of product types as a union of product types $P' \subseteq \mathcal{A}_{\mathbf{prod}}$:

$$t \simeq \bigvee_{a\in P'} a$$

We simply define $\pi(t)$ as $\{(t_1, t_2) \mid t_1 \times t_2 \in P' \wedge t_1 \not\simeq \mathbb{0} \wedge t_2 \not\simeq \mathbb{0}\}$. □

LEMMA 6.37. *Let t be a type such that $t \leq \mathbb{0} \to \mathbb{1}$. Then there exists a finite set of pairs of types $\rho(t) \in \mathcal{P}_f(\mathcal{T}^2)$ and a type $\mathtt{Dom}(t)$ such that:*

$$\forall t_1, t_2.\ (t \leq t_1 \to t_2) \iff \begin{cases} t_1 \leq \mathtt{Dom}(t) \\ \forall (s_1, s_2) \in \rho(t).\ (t_1 \leq s_1) \vee (s_2 \leq t_2) \end{cases}$$

PROOF. We can write:

$$t \simeq \bigvee_{(P,N)\in\mathcal{N}(t) \text{ such that } P\subseteq\mathcal{A}_{\mathbf{fun}}} (\mathbb{0}\to\mathbb{1})\wedge\bigwedge_{a\in P} a\setminus\bigvee_{a\in N\cap\mathcal{A}_{\mathbf{fun}}} a.$$

Clearly, the Lemma is true for $t \simeq \mathbb{0}$ (with $\mathtt{Dom}(t) = \mathbb{1}$ and $\rho(t) = \varnothing$), and if it holds for t and t', then it also holds for $t\vee t'$ (with $\mathtt{Dom}(t\vee t') = \mathtt{Dom}(t)\wedge\mathtt{Dom}(t')$ and $\rho(t\vee t') = \rho(t)\cup\rho(t')$). We can thus assume without loss of generality that t has the form:

$$t = \bigwedge_{a\in P} a\setminus\bigvee_{a\in N} a$$

with $P, N \subseteq \mathcal{A}_{\mathbf{fun}}$, $P \neq \varnothing$, and $t \not\simeq \mathbb{0}$. Lemma 6.13 gives: $t \leq t_1\to t_2 \iff \bigwedge_{a\in P} a \leq t_1\to t_2$ and Lemma 6.8 tells us how to decompose this subtyping into:

$$\forall P'\subseteq P.\ \left(t_1 \leq \bigvee_{s_1\to s_2\in P'} s_1\right) \vee \left(P \neq P' \wedge \bigwedge_{s_1\to s_2\in P\setminus P'} s_2 \leq t_2\right).$$

We can thus define:

$$\mathtt{Dom}(t) = \bigvee_{s_1\to s_2\in P} s_1$$

$$\rho(t) = \left\{\left(\bigvee_{s_1\to s_2\in P'} s_1,\ \bigwedge_{s_1\to s_2\in P\setminus P'} s_2\right) \mid P'\subsetneq P\right\}. \qquad \square$$

COROLLARY 6.38. *Let t and t_1 be two types. If $t \leq t_1\to\mathbb{1}$, then $t \leq t_1\to t_2$ has a smallest solution t_2 which we write $t \bullet t_1$.*

PROOF. Since $t \leq t_1\to\mathbb{1}$, we have $t_1 \leq \mathtt{Dom}(t)$. The assertion $t \leq t_1\to t_2$ is thus equivalent to:

$$\forall(s_1, s_2) \in \rho(t).\ (t_1 \leq s_1) \vee (s_2 \leq t_2),$$

that is:

$$\left(\bigvee_{(s_1,s_2)\in\rho(t) \text{ s.t. } (t_1\not\leq s_1)} s_2\right) \leq t_2.$$

We write $t \bullet t_1$ for the left-hand side of this equation. $\square$

We can now prove Theorem 5.3.

PROOF. Let t, t_1 be two types such that $t \leq t_1\to\mathbb{1}$. Clearly, if $\vdash v_f : t$ and $\vdash v_x : t_1$, then $\vdash v_f v_x : t \bullet t_1$, and thus, subject reduction gives $\vdash v : t \bullet t_1$ if $v_f v_x \overset{\star}{\rightsquigarrow} v$.

Let us prove the opposite implication:

$$\forall v.\ \vdash v : t \bullet t_1 \Rightarrow \exists v_f, v_x.\ (v_f v_x \overset{\star}{\rightsquigarrow} v) \wedge (\vdash v_f : t) \wedge (\vdash v_x : t_1)$$

This property is clearly true for $t \simeq \mathbb{0}$, and if it is true for t and t', then it is true for $t\vee t'$ (because $\mathbb{0}\bullet t_1 \simeq \mathbb{0}$ and $(t\vee t')\bullet t_1 \simeq (t\bullet t_1)\vee(t'\bullet t_1)$). We can thus assume, as

in the proof of Lemma 6.37, that t has the form:

$$t = \bigwedge_{a\in P} a \setminus \bigvee_{a\in N} a$$

with $P, N \subseteq \mathcal{A}_{\mathbf{fun}}$, $P \neq \varnothing$, and $t \not\simeq \mathbb{0}$. Following the same argument as in the proof of Lemma 6.37, we get:

$$t \bullet t_1 = \bigvee_{P'\subsetneq P \text{ s.t. } t_1 \not\leq \bigvee_{t'_1\to t'_2\in P'} t'_1} \left(\bigwedge_{t'_1\to t'_2\in P\setminus P'} t'_2 \right)$$

and

$$t_1 \leq \bigvee_{t'_1\to t'_2\in P} t'_1.$$

Let v be a value of type $t \bullet t_1$. We can find $P' \subsetneq P$ such that $t_1 \not\leq \bigvee_{t'_1\to t'_2\in P'} t'_1$ and $\vdash v : \bigwedge_{t'_1\to t'_2\in P\setminus P'} t'_2$. Let v_x be a value of type $t_1 \setminus \bigvee_{t'_1\to t'_2\in P'} t'_1$ and v_f the abstraction

$$\mu f(P).\lambda x. \left(y = x \in \bigvee_{t'_1\to t'_2\in P'} t'_1 \;?\; fy \mid v \right).$$

It is then easy to check that $\vdash v_f : t$ and $v_f v_x \overset{\star}{\rightsquigarrow} v$. □

6.12. TYPE-CHECKING ALGORITHM. In this section, we assume that the subtyping relation $\leq$ is decidable and we give a type-checking algorithm for our type system.

The key difficulty to overcome is that the set of types t such that $\Gamma \vdash e : t$, for a given environment Γ and a given expression e has no smallest element in general. Indeed, consider the case where e is a well-typed abstraction. The (*abstr*) rule allows us to choose an arbitrary number of incomparable arrow types.

We will thus introduce a new syntactic category, called ***type scheme*** to denote such sets of types. The syntax for type schemes is given by the following productions:

$$\begin{array}{rll} \mathbb{t} ::= & t & t \in \mathcal{T} \\ \mid & [t_1\to s_1;\cdots;t_n\to s_n] & n \geq 1; t_i, s_i \in \mathcal{T} \\ \mid & \mathbb{t}_1 \otimes \mathbb{t}_2 & \\ \mid & \mathbb{t}_1 \mathbin{\mathbb{V}} \mathbb{t}_2 & \\ \mid & \Omega. & \end{array}$$

We will write $[t_i\to s_i]_{i=1..n}$ for $[t_1\to s_1;\cdots;t_n\to s_n]$. We define the function $\{\_\}$ which maps schemes to sets of types:

$$\begin{array}{lcl} \{t\} & = & \{s \mid t \leq s\} \\ \{[t_i\to s_i]_{i=1..n}\} & = & \{s \mid \exists s_0 = \bigwedge_{i=1..n} (t_i\to s_i) \wedge \bigwedge_{j=1..m} \neg(t'_j\to s'_j).\ \mathbb{0} \not\simeq s_0 \leq s\} \\ \{\mathbb{t}_1 \otimes \mathbb{t}_2\} & = & \{s \mid \exists t_1 \in \{\mathbb{t}_1\}, t_2 \in \{\mathbb{t}_2\}.\ t_1 \times t_2 \leq s\} \\ \{\mathbb{t}_1 \mathbin{\mathbb{V}} \mathbb{t}_2\} & = & \{s \mid \exists t_1 \in \{\mathbb{t}_1\}, t_2 \in \{\mathbb{t}_2\}.\ t_1 \vee t_2 \leq s\} \\ \{\Omega\} & = & \varnothing. \end{array}$$

LEMMA 6.39. *Let $\mathbb{t}$ be a type schema. Then $\{\mathbb{t}\} = \varnothing$ if and only if Ω appears in $\mathbb{t}$. Moreover, $\{\mathbb{t}\}$ is closed under subsumption ($t \in \{\mathbb{t}\} \wedge t \leq t' \Rightarrow t' \in \{\mathbb{t}\}$) and intersection ($t \in \{\mathbb{t}\} \wedge t' \in \{\mathbb{t}\} \Rightarrow t \wedge t' \in \{\mathbb{t}\}$).*

PROOF. Straightforward induction on the structure of $\mathbb{t}$. □

LEMMA 6.40. *Let $\mathbb{t}$ be a type scheme and t_0 a type. We can compute a type scheme, written $t_0 \oslash \mathbb{t}$, such that:*

$$\{t_0 \oslash \mathbb{t}\} = \{s \mid \exists t \in \{\mathbb{t}\}.\ t_0 \wedge t \leq s\}.$$

PROOF. We define $t_0 \oslash \mathbb{t}$ by induction on $\mathbb{t}$. If $\mathbb{t}$ is a type t, we take $t_0 \oslash \mathbb{t} = t_0 \wedge t$. If $\mathbb{t}$ is a union $\mathbb{t}_1 \vee \mathbb{t}_2$, we distribute: $t_0 \oslash \mathbb{t} = (t_0 \oslash \mathbb{t}_1) \oslash (t_0 \oslash \mathbb{t}_2)$. If $\mathbb{t}$ is Ω, or if $\{\mathbb{t}\} = \varnothing$, we take $t_0 \oslash \mathbb{t} = \Omega$. For the two remaining cases, we assume that $\{\mathbb{t}\} \neq \varnothing$, and we observe that:

$$t_0 \simeq \bigvee_{(P,N) \in \mathcal{N}(t)} \bigwedge_{a \in P} a \wedge \bigwedge_{a \in N} \neg a.$$

We can thus see t_0 as a Boolean combination built with $\mathbb{0}$, $\mathbb{1}$, $\vee$, $\wedge$, atoms and complement of atoms. For $t_0 \simeq \mathbb{0}$, we take $t_0 \oslash \mathbb{t} = \mathbb{0}$. For $t_0 \simeq \mathbb{1}$, we take $t_0 \oslash \mathbb{t} = \mathbb{t}$. For $t_0 \simeq t_1 \vee t_2$, we take $t_0 \oslash \mathbb{t} = (t_1 \oslash \mathbb{t}) \oslash (t_2 \oslash \mathbb{t})$. For $t_0 \simeq t_1 \wedge t_2$, we take $t_0 \oslash \mathbb{t} = t_1 \oslash (t_2 \oslash \mathbb{t})$. It remains to deal with the case of an atom or a complement of an atom.

For the case $\mathbb{t} = \mathbb{t}_1 \otimes \mathbb{t}_2$, we take:

$$(t_1 \times t_2) \oslash (\mathbb{t}_1 \otimes \mathbb{t}_2) = (t_1 \oslash \mathbb{t}_1) \otimes (t_2 \oslash \mathbb{t}_2)$$

$$\neg(t_1 \times t_2) \oslash (\mathbb{t}_1 \otimes \mathbb{t}_2) = ((\neg t_1 \oslash \mathbb{t}_1) \otimes \mathbb{t}_2) \oslash (\mathbb{t}_1 \otimes (\neg t_2 \oslash \mathbb{t}_2))$$

and if $a \in \mathcal{A} \backslash \mathcal{A}_{\mathbf{prod}}$:

$$a \oslash (\mathbb{t}_1 \otimes \mathbb{t}_2) = \mathbb{0}$$

$$\neg a \oslash (\mathbb{t}_1 \otimes \mathbb{t}_2) = (\mathbb{t}_1 \otimes \mathbb{t}_2).$$

For the case $\mathbb{t} = [t_i \to s_i]_{i=1..n}$, we take:

$$(t \to s) \oslash [t_i \to s_i]_{i=1..n} = \begin{cases} [t_i \to s_i]_{i=1..n} & \text{if } \bigwedge_{i=1..n} t_i \to s_i \leq t \to s \\ \mathbb{0} & \text{if } \bigwedge_{i=1..n} t_i \to s_i \not\leq t \to s \end{cases}$$

$$\neg(t \to s) \oslash [t_i \to s_i]_{i=1..n} = \begin{cases} \mathbb{0} & \text{if } \bigwedge_{i=1..n} t_i \to s_i \leq t \to s \\ [t_i \to s_i]_{i=1..n} & \text{if } \bigwedge_{i=1..n} t_i \to s_i \not\leq t \to s \end{cases}$$

and if $a \in \mathcal{A} \backslash \mathcal{A}_{\mathbf{fun}}$:

$$a \oslash [t_i \to s_i]_{i=1..n} = \mathbb{0}$$

$$\neg a \oslash [t_i \to s_i]_{i=1..n} = [t_i \to s_i]_{i=1..n}.$$ □

LEMMA 6.41. *Let $\mathbb{t}$ be a type scheme and t a type. We can decide the assertion $t \in \{\mathbb{t}\}$, which we also write $\mathbb{t} \leq t$.*

PROOF. First, we make the observation that $t \in \{\mathbb{t}\}$ if and only if $\mathbb{0} \in \{(\neg t) \mathbin{\mathbb{\bigcirc}} \mathbb{t}\}$. Indeed: $\mathbb{0} \in \{(\neg t) \mathbin{\mathbb{\bigcirc}} \mathbb{t}\} \iff \exists s \in \{\mathbb{t}\}.\ (\neg t) \wedge s \leq \mathbb{0} \iff \exists s \in \{\mathbb{t}\}.\ s \leq t \iff t \in \{\mathbb{t}\}$. As a consequence, we only need to deal with the case $t = \mathbb{0}$. If $\{\mathbb{t}\} = \varnothing$, then $\mathbb{0} \in \{\mathbb{t}\}$ does not hold. Otherwise, we conclude by induction over the structure of $\mathbb{t}$:

$$\begin{array}{lll} \mathbb{0} \in \{t\} & \iff & t \simeq \mathbb{0} \\ \mathbb{0} \notin \{[t_i \to s_i]_{i=1..n}\} & & \\ \mathbb{0} \in \{\mathbb{t}_1 \otimes \mathbb{t}_2\} & \iff & (\mathbb{0} \in \{\mathbb{t}_1\}) \vee (\mathbb{0} \in \{\mathbb{t}_2\}) \\ \mathbb{0} \in \{\mathbb{t}_1 \mathbin{\mathbb{\bigcirc}} \mathbb{t}_2\} & \iff & (\mathbb{0} \in \{\mathbb{t}_1\}) \wedge (\mathbb{0} \in \{\mathbb{t}_2\}) \\ \mathbb{0} \notin \{\Omega\}. & & \end{array}$$

□

LEMMA 6.42. *Let $\mathbb{t}$ be a type scheme and $i \in \{1, 2\}$. We can compute a type scheme $\pi_i(\mathbb{t})$ such that*

$$\{\pi_i(\mathbb{t})\} = \{s \mid \exists t_1 \times t_2 \in \{\mathbb{t}\}.t_i \leq s\}$$

PROOF. Let's take for instance $i = 1$. Note that $\exists t_1 \times t_2 \in \{\mathbb{t}\}.t_1 \leq s$ is equivalent to $s \times \mathbb{1} \in \{\mathbb{t}\}$.

If $\mathbb{t} \not\leq \mathbb{1} \times \mathbb{1}$, then we take $\{\pi_1(\mathbb{t})\} = \Omega$. Otherwise, we proceed by induction over the structure of $\mathbb{t}$. For $\mathbb{t} = \mathbb{t}_1 \mathbin{\mathbb{\bigcirc}} \mathbb{t}_2$, we take $\pi_1(\mathbb{t}) = \pi_1(\mathbb{t}_1) \mathbin{\mathbb{\bigcirc}} \pi_1(\mathbb{t}_2)$. For $\mathbb{t} = \mathbb{t}_1 \otimes \mathbb{t}_2$, we take $\pi_1(\mathbb{t}) = \mathbb{t}_1$. For $\mathbb{t} = t$, we take $\pi_1(\mathbb{t}) = \bigvee_{(t_1,t_2)\in\pi(t)} t_1$. The other cases are impossible. □

LEMMA 6.43. *Let $\mathbb{t}$ and $\mathbb{t}_1$ be two type schemes. We can compute a type scheme $\mathbb{t} \bullet \mathbb{t}_1$ such that*

$$\{\mathbb{t} \bullet \mathbb{t}_1\} = \{s \mid \exists t_1 \to t_2 \in \{\mathbb{t}\}.t_1 \in \{\mathbb{t}_1\} \wedge t_2 \leq s\}.$$

PROOF. We proceed by induction over the structure of $\mathbb{t}$. For $\mathbb{t} = \mathbb{t}^1 \mathbin{\mathbb{\bigcirc}} \mathbb{t}^2$, we take $\mathbb{t} \bullet \mathbb{t}_1 = \mathbb{t}^1 \bullet \mathbb{t}_1 \mathbin{\mathbb{\bigcirc}} \mathbb{t}^2 \bullet \mathbb{t}_1$. For $\mathbb{t} = \mathbb{t}^1 \otimes \mathbb{t}^2$ or $\mathbb{t} = \Omega$, we take $\mathbb{t} \bullet \mathbb{t}_1 = \Omega$. For $\mathbb{t} = [t'_i \to s'_i]_{i=1..n}$, we take $\mathbb{t} \bullet \mathbb{t}_1 = (\bigwedge_{i=1..n}(t'_i \to s'_i)) \bullet \mathbb{t}_1$, so the only remaining case is $\mathbb{t} = t$. We observe that $\exists t_1 \to t_2 \in \{\mathbb{t}\}.t_1 \in \{\mathbb{t}_1\} \wedge t_2 \leq s$ is equivalent to $\exists t_1 \in \{\mathbb{t}_1\}.t \leq t_1 \to s$. In accordance with Lemma 6.37, this is equivalent to: $\exists t_1 \in \{\mathbb{t}_1\}.t_1 \leq \mathtt{Dom}(t) \wedge \forall (s_1, s_2) \in \rho(t).\ (t_1 \leq s_1) \vee (s_2 \leq s)$. We now prove that this is equivalent to $\mathbb{t}_1 \leq \mathtt{Dom}(t) \wedge \forall (s_1, s_2) \in \rho(t).\ (\mathbb{t}_1 \leq s_1) \vee (s_2 \leq s)$. The $\Rightarrow$ implication is immediate. Let us check the $\Leftarrow$ implication. For every $(s_1, s_2) \in \rho(t)$ such that $s_2 \not\leq s$, we have $\mathbb{t}_1 \leq s_1$ and it is thus possible to find a type $t'_1 \in \{\mathbb{t}_1\}$ such that $t'_1 \leq s_1$. We define t_1 as the intersection of all these t'_1 and of $\mathtt{Dom}(t)$, and we thus have $t_1 \in \{\mathbb{t}_1\} \wedge t_1 \leq \mathtt{Dom}(t) \wedge \forall (s_1, s_2) \in \rho(t).\ (t_1 \leq s_1) \vee (s_2 \leq s)$. To conclude, we define $t \bullet \mathbb{t}_1$ as Ω if $\mathbb{t}_1 \not\leq \mathtt{Dom}(t)$, and otherwise as:

$$\bigvee_{(s_1,s_2)\in\rho(t) \text{ s.t. } (\mathbb{t}_1 \not\leq s_1)} s_2.$$

□

We can now describe a type-checking algorithm. We define a ***scheme environment*** as a finite mapping $\mathbb{\Gamma}$ from variables to type schemes such that $\{\mathbb{\Gamma}(x)\} \neq \varnothing$ for every x in the domain of $\mathbb{\Gamma}$. The type-checking algorithm is formalized as a total function that maps a scheme environment $\mathbb{\Gamma}$ and an expression e to a scheme written $\mathbb{\Gamma}[e]$. This

function is defined by induction on the structure of e by the following equations:

$$\begin{cases}
\mathbb{\Gamma}[c] = b_c \\
\mathbb{\Gamma}[(e_1, e_2)] = \mathbb{\Gamma}[e_1] \otimes \mathbb{\Gamma}[e_2] \\
\mathbb{\Gamma}[\mu f(t_1 \to s_1; \cdots; t_n \to s_n).\lambda x.e] = \begin{cases} \mathbb{t} & \text{if } \forall i = 1..n.\ \mathbb{s}_i \leq s_i \\ \Omega & \text{otherwise} \end{cases} \\
\qquad\qquad \text{where } \begin{cases} \mathbb{t} = [t_i \to s_i]_{i=1..n} \\ \mathbb{s}_i = ((f : \mathbb{t}), (x : t_i), \mathbb{\Gamma})[e] \quad (i = 1..n) \end{cases} \\
\mathbb{\Gamma}[x] = \begin{cases} \mathbb{\Gamma}(x) & \text{if } \mathbb{\Gamma}(x) \text{ is defined} \\ \Omega & \text{otherwise} \end{cases} \\
\mathbb{\Gamma}[\pi_i(e)] = \pi_i(\mathbb{\Gamma}[e]) \\
\mathbb{\Gamma}[e_1 e_2] = \mathbb{\Gamma}[e_1] \bullet \mathbb{\Gamma}[e_2] \\
\mathbb{\Gamma}[(x = e \in t\ ?\ e_1 | e_2)] = \mathbb{s}_1 \mathbin{\mathbb{\oslash}} \mathbb{s}_2, \\
\qquad \text{where } \begin{cases} \mathbb{t}_0 = \mathbb{\Gamma}[e] \\ \mathbb{t}_1 = t \mathbin{\mathbb{\oslash}} \mathbb{t}_0 \\ \mathbb{t}_2 = (\neg t) \mathbin{\mathbb{\oslash}} \mathbb{t}_0 \\ \mathbb{s}_i = \begin{cases} ((x : \mathbb{t}_i), \mathbb{\Gamma})[e_i] & \text{if } \mathbb{t}_i \not\leq \mathbb{0}, \{\mathbb{t}_i\} \neq \varnothing \\ \mathbb{0} & \text{if } \mathbb{t}_i \leq \mathbb{0} \\ \Omega & \text{if } \{\mathbb{t}_i\} = \varnothing \end{cases} \end{cases} \qquad (i = 1..2).
\end{cases}$$

We are now going to prove soundness and completeness of the algorithm. If $\mathbb{\Gamma}$ is a scheme environment and Γ is a typing environment, we write $\mathbb{\Gamma} \leq \Gamma$ when $\mathbb{\Gamma}$ and Γ have the same domain and for all x in this domain $\mathbb{\Gamma}(x) \leq \Gamma(x)$. If Γ_1 and Γ_2 are two typing environment, we define $\Gamma_1 \wedge \Gamma_2$ by $(\Gamma_1 \wedge \Gamma_2)(x) = \Gamma_1(x) \wedge \Gamma_2(x)$ (undefined when one of the $\Gamma_i(x)$ is not defined). Note that if $\mathbb{\Gamma} \leq \Gamma_1$ and $\mathbb{\Gamma} \leq \Gamma_2$, then $\mathbb{\Gamma} \leq \Gamma_1 \wedge \Gamma_2$.

LEMMA 6.44 (CORRECTNESS). *If $\mathbb{\Gamma}[e] \leq t$, then there exists $\Gamma \geq \mathbb{\Gamma}$ such that $\Gamma \vdash e : t$.*

PROOF. By induction over the structure of e.

$\underline{e = c}$. We have $b_c \leq t$, and thus $\vdash c : t$. We can take for Γ an arbitrary typing environment such that $\Gamma \geq \mathbb{\Gamma}$. We use the $\wedge$ operator on typing environments and Lemma 6.14 to reconcile different Γ's given by several uses of the induction hypothesis.

$\underline{e = x}$. We have $\Gamma(x) \leq t$. We can choose $\Gamma \geq \mathbb{\Gamma}$ such that $\Gamma(x) = t$.

$\underline{e = (e_1, e_2)}$. We have $\mathbb{\Gamma}[e_1] \otimes \mathbb{\Gamma}[e_2] \leq t$. We can thus find $t_1 \geq \mathbb{\Gamma}[e_1]$ and $t_2 \geq \mathbb{\Gamma}[e_2]$ such that $t_1 \times t_2 \leq t$. The induction hypothesis gives $\Gamma_1 \geq \mathbb{\Gamma}$ such that $\Gamma_1 \vdash e_1 : t_1$ and $\Gamma_2 \geq \mathbb{\Gamma}$ such that $\Gamma_2 \vdash e_2 : t_2$. We take $\Gamma = \Gamma_1 \wedge \Gamma_2$.

$\underline{e = e_1 e_2}$. We have $\mathbb{\Gamma}[e_1] \bullet \mathbb{\Gamma}[e_2] \leq t$. We can thus find t_1, t_2 such that $t_1 \to t_2 \geq \mathbb{\Gamma}[e_1]$, $t_1 \geq \mathbb{\Gamma}[e_2]$ and $t_2 \leq t$. The induction hypothesis gives $\Gamma_1 \geq \mathbb{\Gamma}$ such that $\Gamma_1 \vdash e_1 : t_1 \to t_2$ and $\Gamma_2 \geq \mathbb{\Gamma}$ such that $\Gamma_2 \vdash e_2 : t_1$. We take $\Gamma = \Gamma_1 \wedge \Gamma_2$.

$\underline{e = \pi_i(e')}$. We have $\pi_i(\mathbb{\Gamma}[e']) \leq t$. We can thus find t_1, t_2 such that $t_1 \times t_2 \geq \mathbb{\Gamma}[e']$ and $t_i \leq t$. The induction hypothesis gives $\Gamma \geq \mathbb{\Gamma}$ such that $\Gamma \vdash e' : t_1 \times t_2$. We deduce that $\Gamma \vdash e : t_i$ and by subsumption $\Gamma \vdash e : t$.

$e = (x = e' \in t' \ ?\ e_1 \mid e_2)$. We take $\mathbb{t}_0 = \mathbb{\Gamma}[e']$, $\mathbb{t}_1 = t' \mathbin{⩀} \mathbb{t}_0$ and $\mathbb{t}_2 = (\neg t') \mathbin{⩀} \mathbb{t}_0$. We also take $\mathbb{s}_1$ and $\mathbb{s}_2$ as in the corresponding case of the definition of $\mathbb{\Gamma}[e]$. We have $\mathbb{s}_1 \mathbin{⩒} \mathbb{s}_2 \leq t$. We can thus find $s_1 \geq \mathbb{s}_1$ and $s_2 \geq \mathbb{s}_2$ such that $t \geq s_1 \vee s_2$. Let's take $i \in \{1, 2\}$. We will define a type t_i. We have $\mathbb{s}_i \neq \Omega$ since $s_i \geq \mathbb{s}_i$. Two cases remain. If $\mathbb{t}_i \not\leq \mathbb{0}$, we have $\mathbb{s}_i = ((x : \mathbb{t}_i), \mathbb{\Gamma})[e_i]$. The induction hypothesis gives $\Gamma_i \geq \mathbb{\Gamma}$ and $t_i \geq \mathbb{t}_i$ such that $(x : t_i), \Gamma_i \vdash e_i : s_i$. Otherwise, we have $\mathbb{s}_i = \mathbb{0}$ and we take $t_i = 0$. In both cases, we have $t_i \geq \mathbb{t}_i$.

Let's consider the type $t_0 = (t_1 \wedge t') \vee (t_2 \wedge \neg t')$. We now prove that $t_0 \geq \mathbb{t}_0$. Since $t_1 \geq \mathbb{t}_1 = t' \mathbin{⩀} \mathbb{t}_0$, there exists $t_1' \geq \mathbb{t}_0$ such that $t' \wedge t_1' \leq t_1$. Similarly, we have $t_2' \geq \mathbb{t}_0$ such that $(\neg t') \wedge t_2' \leq t_2$. We get $t_0 \geq (t' \wedge t_1') \vee ((\neg t') \wedge t_2') \geq (t' \wedge t_1' \wedge t_2') \vee ((\neg t') \wedge t_1' \wedge t_2') \simeq t_1' \wedge t_2' \geq \mathbb{t}_0$.

Since $t_0 \geq \mathbb{t}_0$, the induction hypothesis gives $\Gamma_0 \geq \mathbb{\Gamma}$ such that $\Gamma_0 \vdash e' : t_0$. Let's consider the types $t_1'' = t_0 \wedge t \leq t_1$ and $t_2'' = t_0 \wedge (\neg t) \leq t_2$. By considering the intersection of Γ_0 and of Γ_1 and Γ_2 when they are defined, we find $\Gamma \geq \mathbb{\Gamma}$ such that $\Gamma \vdash e' : t_0$ and $(x_i : t_i''), \Gamma \vdash e_i : s_i$ when $\mathbb{t}_i \not\leq \mathbb{0}$. The rule (*case*) gives $\Gamma \vdash e : s_1 \vee s_2$. By subsumption, we get $\Gamma \vdash e : t$.

$e = \mu f(t_1 \to s_1; \ldots; t_n \to s_n).\lambda x.e'$. We take $\mathbb{t}$ and $\mathbb{s}_i$ as in the definition of the corresponding case for $\mathbb{\Gamma}[e]$. Since $\mathbb{\Gamma}[e] \neq \Omega$, we get $\mathbb{t} \leq t$ and $\mathbb{s}_i \leq s_i$ for all $i = 1..n$. The induction hypothesis gives, for each i, an environment $\Gamma_i \geq \mathbb{\Gamma}$, and two types $t^i \geq \mathbb{t}$, $t_i'' \geq t_i$ such that $(f : t^i), (x : t_i''), \Gamma_i \vdash e' : s_i$.

We define the type t' as $\bigwedge_{i=1..n} t^i \wedge t$. We have $t' \geq \mathbb{t} = [t_i \to s_i]_{i=1..n}$. We can thus find a type t'' of the form $t'' = \bigwedge_{i=1..n} t_i \to s_i \wedge \bigwedge_{j=1..m} \neg(t_j' \to s_j')$ such that $t' \geq t''$ and $t'' \not\simeq \mathbb{0}$.

If we take for Γ the intersection of all the Γ_i, we obtain $(f : t''), (x : t_i), \Gamma \vdash e' : s_i$ for all i from which we conclude $\Gamma \vdash e : t''$ and thus $\Gamma \vdash e : t$. □

LEMMA 6.45 (COMPLETENESS). *If* $\mathbb{\Gamma} \leq \Gamma$ *and* $\Gamma \vdash e : t$, *then* $\mathbb{\Gamma}[e] \leq t$.

PROOF. By induction over the derivation of $\Gamma \vdash e : t$ and case disjunction over the last rule used in this derivation. The proof is mechanical. We give the details only for the rule (*case*).

$$\frac{\Gamma \vdash e : t_0 \qquad \begin{cases} t_0 \not\leq \neg t \ \Rightarrow\ (x : t_0 \wedge t), \Gamma \vdash e_1 : s \\ t_0 \not\leq t \ \ \Rightarrow\ (x : t_0 \backslash t), \Gamma \vdash e_2 : s \end{cases}}{\Gamma \vdash (x = e \in t \ ?\ e_1 | e_2) : s}.$$

We assume that $\mathbb{\Gamma} \leq \Gamma$ and we take $\mathbb{t}_0$,$\mathbb{t}_1$, $\mathbb{t}_2$,$\mathbb{s}_1$,$\mathbb{s}_2$ as in the definition of $\mathbb{\Gamma}[(x = e \in t \ ?\ e_1 | e_2)]$. We need to prove that $\mathbb{s}_1 \mathbin{⩒} \mathbb{s}_2 \leq s$, that is $\mathbb{s}_1 \leq s$ and $\mathbb{s}_2 \leq s$. We will do the proof for $\mathbb{s}_1$ (the proof for $\mathbb{s}_2$ is similar).

The induction hypothesis gives $\mathbb{t}_0 = \mathbb{\Gamma}[e] \leq t_0$, from which we get $\mathbb{t}_1 \leq t \wedge t_0$. If $\mathbb{t}_1 \leq \mathbb{0}$, then $\mathbb{s}_1 = \mathbb{0} \leq s$. Otherwise, since $\{\mathbb{t}_1\} \neq \varnothing$, we have $\mathbb{s}_1 = ((x : \mathbb{t}_1), \mathbb{\Gamma})[e_1]$. We have $t_0 \not\leq \neg t$, otherwise $\mathbb{t}_1 \leq \mathbb{0}$. We thus have a subderivation $(x : t_0 \wedge t), \Gamma \vdash e_1 : s$. The induction hypothesis, applied to the environment $(x : \mathbb{t}_1), \mathbb{\Gamma}$ gives $\mathbb{s}_1 \leq s$. □

By combining the two previous lemmas, we get an exact characterization of the type-checking algorithm in terms of the type system.

THEOREM 6.46. *For any scheme environment* $\mathbb{\Gamma}$ *and expression* e:

$$\{\mathbb{\Gamma}[e]\} = \{t \mid \exists \Gamma \geq \mathbb{\Gamma}.\Gamma \vdash e : t\}.$$

COROLLARY 6.47. *Let* Γ *be a typing environment. It can also be seen as a scheme environment. For any expression* e *and any type* t*, we have:*

$$\Gamma \vdash e : t \iff \Gamma[e] \leq t.$$

As a special case, the expression e *is well typed under* Γ *if and only if* $\{\Gamma[e]\} \neq \varnothing$.

To conclude with the decidability of the type system, we observe that the assertion $\{\Gamma[e]\} \neq \varnothing$ is decidable (Lemma 6.39).

7. *Commentaries*

In Section 2, we described the basic intuitions and we gave an overview of our approach. In this section, we comment and explain the intuition and motivations that underlie some more technical choices we made in the formal development of the work.

7.1. WHAT DOES THE CLOSING-THE-CIRCLE THEOREM MEAN?. Theorem 5.5 is a nice and important property about our system. It means that whenever the interpretation of types as sets of values is a model, it induces the same subtyping relation as the bootstrap model; as a consequence, there is no point using this model as a new bootstrap model and iterating the whole process again. The theorem is also an indication that the typing rules are somewhat coherent with the definition of models. It is a quality check, but a limited one: we should resist the temptation to read too much from the theorem. Let us be explicit on this point: Theorem 5.5 does *not* say that the definition of models is "valid" in any way. As a matter of fact, it is possible to change the definition of models in very bogus ways and still be able to prove the theorem. If we follow closely the formal development, we see that we could actually change Definition 4.3 and take any definition for $\mathbb{E}(t_1 \to t_2)$ as long as Lemma 6.13 holds. For instance, we could even take a definition that makes arrow types covariant in their domain, for example, $\mathbb{E}(t_1 \to t_2) = \mathcal{P}(\llbracket t_1 \rrbracket) \times \mathcal{P}(\llbracket t_2 \rrbracket)$. Then, of course, the subject reduction theorem would fail to hold. We could even see purely syntactical evidences that something goes wrong (without introducing the operational semantics). With the bogus definition above, we would indeed see that:

(1) the following rule is derivable:

$$\frac{\Gamma \vdash e_1 : t_1 \to t_2 \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 e_2 : t_2} \ (appl'),$$

(which means that the type system does not check the type for the argument in function applications);

(2) Lemma 6.15 (Admissibility of the intersection rule) would fail to hold in general (but it would still hold for values). Indeed, the case for applications in its proof relies on the contravariance of arrow types in their domain.

It is interesting to look at how Theorem 5.5 could have failed with the current definition of models. The easiest way to break the theorem is the typing rule for abstractions. If we did not allow several function types to appear in λ-abstraction, or if we did not allow negation of arrow types to appear in the type assigned to the λ-abstraction, then Theorem 5.5 would not hold.

The fact that the definition of models (and thus subtyping) is "valid" with respect to our calculus is expressed by results from Section 5.1: type-safety says that subtyping is sound with respect to the semantics of the calculus, and Theorem 5.3 gives some further evidence that the whole system is coherent. As a final note about Theorem 5.5, we should emphasise here again that even if the interpretation of types is not a model (i.e., if the bootstrap model is not well-founded), then type-safety still holds.

7.2. On The Prescriptive Nature of Types for λ-Abstractions. The λ-abstractions in our calculus come with an explicit signature (a finite sequence of arrow types). This makes it possible to decide whether a functional value has type $t\to s$ or not, without looking at the function body and without relying on the typing judgment. Such a decision has to be made at run-time to reduce a dynamic type-dispatch against a type such as $t\to s$. So, the result of type-dispatch can depend on the explicit type annotations on λ-abstractions. For instance, the expression $(g = (\mu f(\text{true}\to\text{true}).\lambda x.x) \in (\text{false}\to\text{false})\ ?\ 1 \mid 0)$ evaluates to 0 (because $\text{true}\to\text{true} \not\leq \text{false}\to\text{false}$), but $(g = (\mu f(\text{false}\to\text{false}).\lambda x.x) \in (\text{false}\to\text{false})\ ?\ 1 \mid 0)$ evaluates to 1.

This observation gives a "paradox" that we would obtain if we tried to define a Curry-style type assignment for λ-abstractions, that is, if we did not include an explicit signature. Indeed, a function could check its own type and behave differently according to it. Consider for instance the value $v = \mu f.\lambda x.(g = f \in \text{true}\to\text{true}\ ?\ \text{false} \mid \text{true})$. Then, v maps true to true if and only if it does not have type true→true.

7.3. On the Typing Rule for Abstractions. The negative arrow types in the typing rule for λ-abstractions may look surprising. Indeed this rule can assign to the functional value $\mu f(\text{true}\to\text{true}).\lambda x.x$ the type $\neg(\text{false}\to\text{false})$ even if semantically, the function maps the value false to itself. We have already explained in Section 3.3 that we need these negative arrow types in order to have the property that every value has type t or $\neg t$ for any type. The previous section showed a different problem that arises if we try to get rid of the explicit signature on λ-abstractions.

If we rely on the typing judgment where the rule is modified so as to disallow negative arrow types but without changing the operational semantics, the calculus trivially remains type-safe. In this case, the reduction rule for the dynamic type dispatch must use the old judgment, so that we always have $\vdash v : t$ or $\vdash v : \neg t$.

This suggests a variation of the (*abstr*) rule that would allow negative arrow types only if the abstraction is closed (the only free variables of the body can be the function name or the argument name). This can be thought as some kind of value-restriction. With this new typing judgment, we preserve all the formal properties of our calculus, including type preservation and Theorem 5.4, because the new typing judgment coincides with the old one on values.

7.4. On the Admissibility of a Union Rule. Lemma 6.15 says that the following rule is admissible in our type system:

$$\frac{\Gamma \vdash e : t_1 \quad \Gamma \vdash e : t_2}{\Gamma \vdash e : t_1 \wedge t_2}.$$

One might consider the following dual rule for union types:

$$\frac{\Gamma, (x : t_1) \vdash e : t \quad \Gamma, (x : t_2) \vdash e : t}{\Gamma, (x : t_1 \vee t_2) \vdash e : t}\ (union).$$

Since we have adopted a call-by-value semantics, variables in the environment are meant to be substituted by values, and since a value of type $t_1 \vee t_2$ has type t_1 or type t_2, this rule is semantically sound (the substitution lemma would need to be restricted to values, though). However, this rule, which corresponds to reasoning by case disjunction, is not admissible in our system: it would allow us to derive $(x : \text{bool}) \vdash (x, x) : \text{true} \times \text{true} \vee \text{false} \times \text{false}$, while the smallest type the current system can assign to (x, x) under this typing environment is $\text{bool} \times \text{bool}$.

Therefore, the question about the opportunity of adding such a rule to our system naturally arises. We decided not to do so since we can simulate the union rule with an explicit annotation that drives the case disjunction. Let us write $\text{case}(x, t, e)$ for the expression $(y = x \in t \ ?\ e \mid e)$ (for y not free in e). Then, the following rule is admissible (and even derivable) in our system:

$$\frac{\Gamma, (x : t_1) \vdash e : t \quad \Gamma, (x : t_2) \vdash e : t}{\Gamma, (x : t_1 \vee t_2) \vdash \text{case}(x, t_1, e) : t}$$

Note that e and $\text{case}(x, t_1, e)$ are observationally equivalent (i.e., they are indistinguishable when embedded in ground contexts of basic type: see, for instance, Definition 6.4.1, page 132, in Amadio and Curien [1998]). Then, replacing e with $\text{case}(x, t_1, e)$ is just an extra hint for the type-checker and it does not break existing typing derivations. Indeed, the following rule is admissible:

$$\frac{\Gamma \vdash e : t \quad x \in \Gamma}{\Gamma \vdash \text{case}(x, t_1, e) : t}$$

So, if we have a derivation for a judgment $\Gamma \vdash e : t$ in the system extended with the rule $(union)$, it is possible to compute an expression e' observationally equivalent to e and such that $\Gamma \vdash e' : t$ is derivable in the current system (viz., without $(union)$). This expression is obtained by wrapping some subexpressions of e with the case(_) operator, in correspondence of the occurrences of the $(union)$ rule in the original derivation. Since the same subexpression can be typed several times (because of overloaded functions), it is important that case(_) does not break existing derivations.

Finally it is interesting to notice that Pierce's union elimination rule [Pierce 1991]

$$\frac{\Gamma \vdash e : u_1 \vee u_2 \quad \Gamma, x : u_1 \vdash e' : s \quad \Gamma, x : u_2 \vdash e' : s}{\Gamma \vdash \ \texttt{case}\ e\ \ \texttt{of}\ x \Rightarrow e' : s}\ \text{(UNION-E)}$$

is a special case of our $(case)$ rule given in Section 3.3 where $e_1 = e_2 = e'$, $t_0 = u_1 \vee u_2$, and t is either u_1 or u_2 (modulo an application of the Strengthening Lemma—Lemma 6.14, Section 6.3—when the intersection of u_1 and u_2 is not empty).

7.5. ON THE REASON WHY RECURSION IS RESTRICTED TO FUNCTIONS. One might wonder why recursion is restricted to functions in our calculus. Imagine we had arbitrary recursion on expressions. Then the expression $\mu x.(x, x)$ should be a (recursive) value. We can consider a recursive type $t = (\neg t) \times (\neg t)$ and look at

whether the value $v = \mu x.(x, x)$ has type t or not. Clearly, we expect to have $\vdash v : t$ if and only if $\vdash (v, v) : t$, which is equivalent to $\vdash (v, v) : (\neg t)\times(\neg t)$, and thus to $\vdash v : \neg t$. But since v is a value, this is equivalent to $\neg(\vdash v : t)$. This paradox justifies that we combine recursion and λ-abstraction in a single construction.

As an aside, note that restricting recursion to single abstractions is enough to let us encode mutually recursive functions. For instance, assume that we want to define two mutually recursive functions:

$$f_1(t_1\to s_1; \ldots ; t_n\to s_n).\lambda x.e_1$$
$$f_2(t'_1\to s'_1; \cdots ; t'_m\to s'_m).\lambda y.e_2$$

where the body of the two functions can refer to both f_1 and f_2. A possible encoding of the definition above is

$$\begin{array}{l}\mu f(\{1\}\to \bigwedge_{i=1..n} t_i\to s_i; \{2\}\to \bigwedge_{j=1..m} t'_j\to s'_j).\\ \quad \lambda c.(c = c \in \{1\}\ ?\\ \qquad \mu f_1(t_1\to s_1; \cdots ; t_n\to s_n).\lambda x.e_1\sigma \ \mid\\ \qquad \mu f_2(t'_1\to s'_1; \cdots ; t'_m\to s'_m).\lambda y.e_2\sigma)\end{array}$$

where $\{1\}$ and $\{2\}$ are two basic singleton types (with associated constants 1 and 2) and the substitution σ replaces f_1 with $(f\ 1)$ and f_2 with $(f\ 2)$. Other encodings are possible and left as an exercise to the reader.

8. *Related Work*

This work started from our desire to extend the work by Hosoya and Pierce [2003] on XDuce with first-class functions and arrow types, therefore it is natural to start this section with it. XDuce is a domain specific language specially designed to write XML transformations. Values are fragments of XML documents, which can be described by so-called regular expression types [Hosoya et al. 2000] (this notion of types generalises some widely used notions of types for XML documents such as DTD or XML-Schema). In XDuce a subtyping relation allows the programmer to use implicitly an expression of type t where an expression of type s is expected, provided that t is a subtype of s. Despite the richness of the type algebra, the definition for this subtyping relation is extremely simple: since types denote sets of values, subtyping can simply be described as the set-theoretic inclusion of interpretations. As a matter of fact, XDuce types can express exactly regular tree languages. It is well known that this class of languages is closed under all Boolean operators: the difference or the intersection of two XDuce types can be expressed by XDuce types, even if there is no explicit constructors for intersection or negation (probably, in order to keep the syntax of types as simple as possible). As far as we know, XDuce was the first type system with subtyping where types are interpreted purely set-theoretically and where sets denoted by types are closed under all Boolean operators.

XDuce also has a powerful notion of pattern matching [Hosoya and Pierce 2001], where patterns are basically types extended with capture variables. In particular, a pattern matching can perform arbitrary dispatch on types at run-time, so that XDuce semantics is actually driven by types. Because of the very rich type algebra (and in particular of the fact that it is closed under Boolean operators), the static type-checking of pattern matching results very precise.

Despite its very functional style (mutually recursive functions, structural types, pattern matching), XDuce lacks first-class functions. Our initial goal was thus to fill this gap while preserving XDuce key ingredients: (i) a rich type algebra, which supports recursive types, subtyping and a complete set of Boolean operators, and interpret them in a purely set-theoretic way (including negation); (ii) a type-driven semantics (to which we add overloaded functions so that we can reflect dynamic type dispatch on functions' interfaces). Other directions for practically embedding XDuce type system into general purpose languages have been studied independently, for example, Xtatic [Gapayev and Pierce 2003] or OCamlDuce [Frisch 2006]. In this work, though, we did not want to embed XDuce into some host type system, but to study the implications of keeping its salient features, in particular a complete set of Boolean combinators, while designing a whole language with first-class functions. The same goal was pursued by Jérôme Vouillon in a recent work [Vouillon 2006] by following an approach opposite to ours. Vouillon gives up intersection and negation types and starts from a particular model of functions in order to avoid a circularity. In particular, this is obtained by defining a subtyping relation via a deduction system that is then used to type the expressions of the language. This induces a model of values that, thanks to the absence of intersections (besides negations), is sound and complete with respects to the syntactically defined subtyping relation. The advantage of giving up intersections and negations is that besides arrow types, the system also accounts for parametric explicit polymorphism.

We already discussed in the introduction why our work fills the gap between existing work on intersection types and that on lately bound overloaded functions. More precisely, on the one hand we have the work on overloading where functions can be formed of different pieces of code stuck together, each piece of code corresponding to a different input type; however the types of these overloaded functions do not have a set-theoretic characterization as intersection types instead have [Castagna et al. 1995]. On the other hand, there is the line of research on intersection (and union) types [Barendregt et al. 1983; Coppo and Dezani-Ciancaglini 1980; Barbanera et al. 1995; Reynolds 1991, 1996], where types have a set-theoretic behavior but where different components of an intersection of arrows cannot correspond to different pieces of code: intersections stem from different repeated typings of a same code, whence a "flavor" of parametricity (where the "parameter" is the hypothesis used in each typing of a function body). We can now better pinpoint where such a "flavor" comes from. It resides in the equivalence we discussed at the end of Section 2.6, that is

$$(t_1 \to s_1) \wedge (t_2 \to s_2) \simeq (t_1 \vee t_2) \to (s_1 \wedge s_2)\,. \tag{7}$$

This, in some sense, states that it is not possible to have a function with two different behaviors that are chosen according to the type of the argument (see Appendix A.3 for a semantic interpretation of this fact). The equation above holds in the theory of union and intersection types of Barbanera et al. [1995],[7] and although it cannot be proved in the theory of Forsythe, it is not possible, in general, to write a term in Forsythe that separates the two types of Eq. (7).[8] An important piece of work

[7] *Idem*, axioms (11) and (12) of Definition 3.3.

[8] Besides, in Forsythe there is the constraint of "coherence" so that, as a concrete example, it is not possible to define an overloaded function of type $(\text{int} \to \text{int}) \wedge (\text{real} \to \text{real})$ that when applied to an integer returns zero and when applied on a non-integer real returns one.

related to this aspect of the research is the work on refinement types. When refinement types are used for logical frameworks [Pfenning 1993], then they have with respect to Eq. (7) the same behavior as the works on union and intersection types we cited above. Yet, when refinement types are coupled with datatype definitions and applied to ML, then they work better in this respect, since it is then possible to write functions with intersection types in which a particular piece of code is executed only for a given input type [Freeman and Pfenning 1991]. It is thus possible to write a term that separates two types of the same form as in Eq. (7). However, this works only for the *declared* refinements of a datatype and, therefore, it does not account for all possible subsets of a generic type. Therefore the strict containment of the types in (7) cannot be proved in general. Rather than a drawback, this is a direct consequence of using refinement types with Curry-style λ-abstractions: using Church-style abstractions, as we do, may require code-duplication, in particular in case of overloaded functions that return functions with varying types but with the same behavior. While this duplication can be avoided by uncurrifying the overloaded function, it would make it impractical to use intersections in the way they are used in the context of refinement types.

A mainstream way to deal with a complex type algebra with Boolean operators is to rely on a denotational semantics for the calculus and to interpret types as ideals in this model. There exist a rich literature that follows this approach, for instance, Aiken and Wimmers [Aiken and Wimmers 1993; Aiken et al. 1994], Damm [Damm's 1994b; Vouillon and Melliés 2004], Melliès and Vouillon [Vouillon and Melliés 2004; Melliès and Vouillon 2005]. Even Amadio and Cardelli's seminal paper on subtyping recursive types [Amadio and Cardelli 1993] proposes a "denotational" interpretation of types (as complete uniform partial-equivalence relations). The main difference between our work and this line of research is that we cannot rely on a denotational semantics either for the calculus (because of the type-driven semantics[9]) or for the types (because we want to interpret negation as set-theoretic complement and all the denotational interpretations of types we are aware of are not closed under complement). The techniques devised in our formal development are thus quite different from those used before and some aspects of our calculus might seem strange when looked from the point of view of denotational semantics. An example of "strangeness" is our treatment of negative arrow types in the typing rule for abstractions.

One way to position our article within the existing literature is to consider that we show how to introduce and study a *semantic* notion of subtyping, not only when no denotational semantics for the calculus can account for type negation, but even when a denotational semantics for the calculus is out of reach. Our use of the adjective "semantic" refers specifically to the definition of subtyping (by opposition to a syntactic/axiomatic definition), and *not* to the semantics of the calculus. Strictly speaking, we *do not* even give a semantics of types: the interpretation of types is functional to the semantics of the subtyping relation, but it is not intended to describe what types are. This is clear when considering our universal model: arrow

[9] The definition of a denotational semantics for a language with overloaded functions and dynamic dispatch—as the one studied here—is still an open problem: the attempts at creating such a definition we are aware of either put strong restrictions on dynamic dispatch [Castagna et al. 1993; Tsuiki 1994] or they impose a stratified construction of higher order types [Studer 2001] (a technique introduced for $\lambda\&$ to enforce strong normalization [Castagna et al. 1995]).

types are interpreted as sets of finite relations, but it is patent that the types of the language we presented *are not* sets of finite graph functions. The only semantics we define is the semantics of subtyping. This is perfunctory characterized by the interpretation of Boolean constructors and of the empty types. More precisely, since we require that union, intersection, and negation type constructors are interpreted as the corresponding set-theoretic operators (or, equivalently, that they obey the same laws as the corresponding set-theoretic operators), then the semantics of subtyping is univoquely identified by the set of empty types. So the core of this work sums up to identifying the set of types that are equivalent to the empty type. This is clearly less demanding than defining the entire semantics of types or, *a fortiori*, the semantics of a complete language.

In addition to the fundamental difference that we discussed above, it is interesting to compare in more details our work with Damm's [1994b]. Damm's system includes intersection and union types, and is also based on ideas from the theory of regular tree languages. Specifically, it encodes a function type as a set of sequences that represent all the possible graphs for finite approximations of functions in this type; this indirect interpretation does not give a direct and effective subtyping rule for Boolean combinations of arrow types. We could not extract from Damm's [1994b] a concrete characterization of the subtyping relation. Instead, our direct treatment gives a new and nontrivial subtyping rule for arrow types, which turned out to be useful in other contexts. In particular, a connection has been established between this rule and the minimal relevant logic $\mathbf{B}_+$ [Dezani-Ciancaglini et al. 2002].

The foundational work by Melliès and Vouillon [Vouillon and Melliés 2004; Melliès and Vouillon 2005] generalizes the model of ideals for recursive and polymorphic types proposed by MacQueen et al. [1986]. Their approach shares with our work the primacy of the types over the expressions, insofar as the latter are somehow functional to the justification of the former (in Vouillon and Melliés [2004] and Melliès and Vouillon [2005], types are sets of intentionally defined expressions, in the sense that they are defined in terms of some properties they must satisfy). Contrary to our work, Melliès and Vouillon are not interested in preserving a strictly set-theoretic interpretation of Boolean operators (e.g., their union type is an over-approximation of the set-theoretic union), they do not care about the completeness of this set of operators (negation is not accounted for, although it should be possible to add it[10]), and they do not insist of the effectiveness of the subtyping relation. Actually, in Vouillon and Melliés [2004] and Melliès and Vouillon [2005], the subtyping relation plays the role of a consistency check for their denotational semantics (only soundness of the subtyping rules is stated). Our research aims at a far more modest and practical target: we are not trying to give a denotational account for subtyping and Boolean operators, but only to define a subtyping relation. As such, we are much more in the realm of the syntax than the one of the semantics.

[10]One of the JACM reviewers suggested that negation could be interpreted as the complement of reducibility candidates for weak normalisation and conjectures that such an interpretation would be compatible with Mellies and Vouillon's approach—hence, with recursive types—as long as one adds a stratification on terms to the language as in the language interpreted by MacQueen et al. [1986] in the ideal model.

9. *Conclusion*

Our original motivation for developing the theory presented in this article was the addition of first-class functions to XDuce while preserving the set-theoretic approach to subtyping. This was the starting point of the ℂDuce project [CDUCE], aiming at developing a programming framework covering several aspects of XML programming: efficient implementation, query languages, web-services, web programming, and so on.

The reader might be surprised to face such a complex theory in the setting of an XML-oriented functional language. First, we should mention that XML plays no role in the complexity of the theory. The circularity that our bootstrapping technique addresses comes only from the combination of arrow types, recursive types and Boolean connectives. Since XDuce already had recursive types and Boolean connectives, it seemed natural to add arrow types and to fully integrate them with these features. Simpler solutions could have been possible, for example, by stratifying the type algebra so as to avoid any interaction between arrow types and existing XDuce types: this is what the first author did to integrate XDuce types into an ML-based type system [Frisch 2006].

Second, we could have presented the theory without introducing the abstract concept of models. Indeed, for the application to a specific programming language, we could have worked directly with the universal model (Section 6.8). That said, we believe that the current presentation better captures the essence of our approach. Working directly with a specific model might have seemed mysterious and ad hoc.

Although we presented our notion of model and the bootstrapping technique on a specific type algebra and for a specific calculus, our framework is quite robust. The Appendix shows how to extend our system with reference types or to modify it to deal with nonoverloaded functions. Frisch's Ph.D. dissertation [Frisch 2004] describes another variant of the system where application is always well typed (the operational semantics can return any value if the function is not prepared to deal with the argument and the type system does not give any static information about the type of the result). All these modifications are quite local and do not change the structure of the formal development nor the main properties of the system.

More importantly, our approach and the techniques we developed turned out to have much a broader application than we initially expected. What we devised is the first approach for a higher order λ-calculus in which union, intersection, *and* negation types have a set-theoretic interpretation. The logical relevance of the approach was independently confirmed by Dezani-Ciancaglini et al. [2002] who showed that the subtyping relation induced by the universal model of Section 6.8 restricted to its positive part (i.e., arrows, unions, intersections but no negations) coincides with the relevant entailment of the $\mathbf{B}_+$ logic (defined 30 years before we started our work). This same approach can be applied to paradigms other than λ-calculi: Castagna et al. [2005, 2007] use our technique to define the $\mathbb{C}\pi$-calculus, a π-calculus where Boolean combinators are added to the type constructors $\mathtt{ch}^+(t)$ and $\mathtt{ch}^-(t)$ which classify all the channels on which it is possible to read or, respectively, to write a value of type t. The technique using the extensional interpretation is still needed for cardinality reasons, however bootstrapping in $\mathbb{C}\pi$ has a different flavor, since it generates a model that is much closer to the model of values. Interestingly, this model is defined by a fix-point construction. $\mathbb{C}\pi$ features several points that are in common with or dual to ℂDuce: $\mathbb{C}\pi$ presents the same paradox one meets

when adding reference types to CDuce [Castagna and Frisch 2005]. The paradox can be avoided by restricting $\mathbb{C}\pi$ to its "local" version [Castagna et al. 2005] or by using less expressive models [Castagna et al. 2007] but in the former case the type schemes of Section 6.12 must be reintroduced, in spite of the fact that they are not needed for the full version of $\mathbb{C}\pi$. Another striking resemblance between CDuce and $\mathbb{C}\pi$ that is worth mentioning is that in order to decide the subtyping relation for $\mathbb{C}\pi$, one tackles the same difficulties as those met in deciding general subtyping for a polymorphic extension of CDuce (actually of XDuce [Hosoya et al. 2005]), namely, one must be able to decide whether a type is a singleton or not. An informal introduction to these aspects can be found in Castagna [2005], while the formal correspondence between CDuce and $\mathbb{C}\pi$ is studied in Castagna et al. [2006].

Finally, let us conclude with a more subjective remark. When we applied our approach to distinct paradigms, we often had the impression that our technique pushed the various systems to their limits: by choosing appropriate models, we could mimic the existing type systems, but by tweaking them a little bit we could reach some "semantic" limits, such as the incompatibility of recursion and some naive implementations of references and channels or the need to descend down at the atomicity of types to decide subtyping. This seems to suggest that our technique exhibits and gives us some insights about some intrinsic difficulties that appear when Boolean operators are combined with various type constructors.

Appendix

A. *Variants and Extensions*

In order to illustrate strengths and limits of our approach, we sketch in this Appendix some variants and extensions of our system.

A.1. ADDING OTHER KINDS OF DATA CONSTRUCTORS. Our system includes pairs (and product types). Other kinds of data constructors are very easy to encode in or to add to the system. For instance, assuming two basic singleton types {1} and {2}, a disjoint sum type constructor t_1+t_2 can be encoded as $(\{1\}\times t_1)\vee(\{2\}\times t_2)$; the injections $\mathtt{in}_l(e)$ and $\mathtt{in}_r(e)$ become $(1, e)$ and $(2, e)$; and the case disjunction $\mathtt{case}\ e\ \mathtt{of}\ \mathtt{in}_l(x_1) \to e_1 \mid \mathtt{in}_r(x_2) \to e_2$ becomes:

$$(x = e \in \{1\}\times\mathbb{1}\,?\, e_1[x_1 := \pi_1(x)] \mid e_2[x_2 := \pi_1(x)]).$$

If we want to extend our system with built-in sum types instead of encoding them, all changes are straightforward. For example, the definition of the extensional interpretation would be:

$$\mathbb{E}(t_1+t_2) = [\![t_1]\!] + [\![t_2]\!] \subseteq D + D$$

(where $+$ on the right-hand side denotes the set-theoretic disjoint sum).

More complex data constructors can be similarly added. For instance, Frisch's thesis [Frisch 2004] details the construction of extensible records that support concatenation and field removal. The subtyping rules that are derived from mechanical set-theoretic "arithmetic" are rather complex.

A.2. REFERENCE CELLS. Besides being a fascinating object of type theoretical study, reference types are a very useful and used programming construction. Therefore, we might want to add reference cells to our system. To this end, we would add a new kind of unary type constructor $\mathtt{ref}(t)$.

Before extending our calculus, let us describe a "paradox" that arises with reference cells in presence of a set-theoretic interpretation of Boolean connectives. Intuitively, a value of type $\mathtt{ref}(t)$ should be a cell from which we must be prepared to read any value of type t and to which we are allowed to write any value of this type. Clearly, with such an interpretation, the type $\mathtt{ref}(t)\wedge\mathtt{ref}(s)$ should be empty as soon as t and s are not equivalent; otherwise, any value in this intersection would give a way to coerce for free from one type to the other. Conversely, if $t \simeq s$, then $\mathtt{ref}(t)\wedge\mathtt{ref}(s) \simeq \mathtt{ref}(s)$, and if $s \not\simeq \mathbb{0}$, this type should not be empty (if $s \simeq \mathbb{0}$, then $\mathtt{ref}(s)$ can be empty, it suffices to disallow uninitialised references). So, intuitively for all types t,s with $s \not\simeq \mathbb{0}$:

$$\mathtt{ref}(t)\wedge\mathtt{ref}(s) \not\simeq \mathbb{0} \iff t \simeq s. \tag{8}$$

Can we define a notion of model to account for this behavior? The answer is no. To see why, consider a nonempty basic type b, and build the recursive type $t = b\vee(\mathtt{ref}(t) \wedge \mathtt{ref}(b))$. Since the basic type does not intersect reference types, then t is equivalent to b if and only if the right-hand side of the union in its definition is empty, that is:

$$t \simeq b \iff \mathtt{ref}(t) \wedge \mathtt{ref}(b) \simeq \mathbb{0}$$

and because of (8), we obtain:

$$t \simeq b \iff t \not\simeq b.$$

This negative result does not mean that it is impossible to add reference types to our system, only that we cannot do it and validate Eq. (8). This equation was obtained by the argument that whatever value we write in a reference, we must be prepared to read it back from it. So let us imagine a notion of reference cell that comes with two sets: a set X_1 of values that can be read from it, and a set X_2 of values that can be written to it. We can, for instance, design the operational semantics such that if we try to write a value v in it, it simply discards it if $v \notin X_1$ (the type system will ensure that $v \in X_2$). A reference marked (i.e., explicitly typed) with the pair (X_1, X_2) should thus have type $\mathtt{ref}(t)$ when $X_1 \subseteq [\![t]\!] \subseteq X_2$ and $X_1 \neq \varnothing$. With these intuitions in mind, the formal definitions follow. We start with the definition for the extensional interpretation:

$$\mathbb{E}(\mathtt{ref}(t)) = \mathtt{ref}([\![t]\!]) \subseteq D \times \mathcal{P}(D) \times \mathcal{P}(D)$$

where the right-hand side is defined by:

$$\mathtt{ref}(X) = \{(d, X_1, X_2) \mid d \in X_1 \subseteq X \subseteq X_2\}.$$

We also extend the calculus with the following constructions:

$$\begin{aligned} e &::= \cdots \mid\ !e \mid (e := e) \mid \mathtt{ref}_{t_1,t_2}(e) \\ v &::= \cdots \mid \mathtt{ref}_{t_1,t_2}(v). \end{aligned}$$

The first and second productions of expressions are for dereferencing and assignment. The corresponding typing rules are standard (we arbitrarily take $\mathbb{1}$ as the result type for the assignment):

$$\frac{\Gamma \vdash e : \mathtt{ref}(t)}{\Gamma \vdash\!e : t} \qquad \frac{\Gamma \vdash e_1 : \mathtt{ref}(t) \quad \Gamma \vdash e_2 : t}{\Gamma \vdash (e_1 := e_2) : \mathbb{1}}.$$

The third new construction creates a reference with the result of e as the initial value and t_1, t_2 as markers (corresponding to X_1 and X_2 in the definition of $\mathtt{ref}(X)$). Note that we consider here $\mathtt{ref}_{t_1,t_2}(v)$ as a value (when it is well typed). Of course, to define the operational semantics, we would need a notion of store and locations to account for the sharing of reference cells. Since this is standard, we do not formalise such a semantics here. It suffices to say that a reference creation must reduce to a fresh location; this reduction would extend the store to map the location to the initial value for the reference. Such a reduction should be disallowed under a λ-abstraction with several arrow types (one can, for instance, use a weak reduction semantics).

The expression $\mathtt{ref}_{t_1,t_2}(e)$ should have type $\mathtt{ref}(t)$ if and only if $t_1 \leq t \leq t_2$; otherwise, following our experience with function types, it should have type $\neg\mathtt{ref}(t)$. As a consequence, in order to preserve the admissibility of the intersection rule, we use the following typing rule:

$$\frac{\Gamma \vdash e : t_1 \quad \forall i = 1..n.\, t_1 \leq s_i \leq t_2 \quad \forall j = 1..m.\, \neg(t_1 \leq s'_j \leq t_2)}{\Gamma \vdash \mathtt{ref}_{t_1,t_2}(e) : \bigwedge_{i=1..n} \mathtt{ref}(s_i) \wedge \bigwedge_{j=1..m} \neg\mathtt{ref}(s'_j)}.$$

Although we do not formalize the operational semantics, the intuition is that at any point during run-time, a reference cell of type $\mathtt{ref}(t)$ will have the form $\mathtt{ref}_{t_1,t_2}(v)$ where v is a value of type t_1 and $t_1 \leq t \leq t_2$. Reading the content of such a reference returns v. Writing a value v' checks dynamically if v' has type t_1 and if so, replaces v with v'; otherwise, nothing happens. Our type system ensures that any value read from a reference of type $\mathtt{ref}(t)$ has type t and that any value v assigned to a reference of type $\mathtt{ref}(t)$ has type t (but if the reference is of the form $\mathtt{ref}_{t_1,t_2}(v')$ with v not in t_1 it might decide to reject this value silently). Of course, we do not really want references to reject values we assign to them. But it is clear that if the original program only contains reference expressions of the form $\mathtt{ref}_{t,t}(e)$, this will never happen. Allowing two different types t_1, t_2 is just a way to obtain the analog of Theorem 5.5 and to avoid the "paradox" implied by Eq. (8) at the beginning of this section.

All the formal definitions and results about models and the type system are easily adapted. Here, we only hint at the nontrivial points. We start with a set-theoretic lemma to study the subtyping relation induced by models:

LEMMA A.1. *Let $(X_i)_{i\in P}$ and $(Y_j)_{j\in N}$ two families of subsets of D. Then:*

$$\bigcap_{i\in P} \mathtt{ref}(X_i) \subseteq \bigcup_{j\in N} \mathtt{ref}(Y_j)$$
$$\Longleftrightarrow$$
$$\left(\bigcap_{i\in P} X_i = \varnothing\right) \text{ or } \left(\exists j \in N.\, \bigcap_{i\in P} X_i \subseteq Y_j \subseteq \bigcup_{i\in P} X_i\right)$$

PROOF. The $\Leftarrow$ implication is straightforward. For the opposite direction, we assume that $\bigcap_{i\in P} \mathtt{ref}(X_i) \subseteq \bigcup_{j\in N} \mathtt{ref}(Y_j)$ and $\bigcap_{i\in P} X_i \neq \varnothing$. We define Z_1 as $\bigcap_{i\in P} X_i$ and Z_2 as $\bigcap_{i\in P} Y_i$. We pick an element d from Z_1, which is not empty by hypothesis. The triple (d, Z_1, Z_2) is in $\bigcap_{i\in P} \mathtt{ref}(X_i)$, and thus, by hypothesis, also in $\bigcup_{j\in N} \mathtt{ref}(Y_j)$. This gives a j such that (d, Z_1, Z_2) is in $\mathtt{ref}(Y_j)$ and the rest of the proof follows easily. □

Note, in particular, that $\texttt{ref}(t)\wedge\texttt{ref}(s)$ is empty if and only if $t\wedge s$ is empty, so Eq. (8) does not hold. However, we also observe the invariance property $\texttt{ref}(t) \leq \texttt{ref}(s) \iff t \simeq s$ or $t \simeq \mathbb{0}$ which is the least we can expect from reference types.

We want the recursive type $t = \texttt{ref}(t)$ to be empty. For cardinality reason, we cannot extend the notion of structural interpretation by requiring $D \times \mathcal{P}(D) \times \mathcal{P}(D) \subseteq D$, $[\![\texttt{ref}(t)]\!] = \texttt{ref}([\![t]\!])$. We use the same trick as for function types. We define:

$$\texttt{ref}_f(X) = \{(d, X_1, X_2) \mid d \in X_1 \subseteq X \subseteq X_2\} \subseteq D \times \mathcal{P}_f(D) \times \mathcal{P}_{cf}(D),$$

where $\mathcal{P}_{cf}(D)$ denotes the set of cofinite subsets of D. We can easily check that replacing $\texttt{ref}(\_)$ by $\texttt{ref}_f(\_)$ in the Lemma above does not change anything when P and N are finite. We now take the following definition for a structural interpretation:

—$D^2 \subseteq D$ and $D \times \mathcal{P}_f(D) \times \mathcal{P}_{cf}(D) \subseteq D$

—for any types t_1,t_2: $[\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!]$

—for any type t: $[\![\texttt{ref}(t)]\!] = \texttt{ref}_f([\![t]\!])$

—The binary relation on D induced by $(d_1, d_2) \triangleright d_i$ and by $(d, X_1, X_2) \triangleright d$ is Noetherian.

The definition of the universal model is adapted accordingly: D^0 is the initial solution to the equation $D^0 = \mathcal{C} + D^0 \times D^0 + \mathcal{P}_f(D^0 \times D^0_\Omega) + (D \times \mathcal{P}_f(D) \times \mathcal{P}_{cf}(D))$. Concretely, we add a new production to this inductive definition of elements of D^0:

$$d \quad ::= \quad \cdots \quad | (d, \{d, \ldots, d\}, D^0 \backslash \{d, \ldots, d\})$$

The definition of the predicate $(d : t)$ is extended with:

$$((d, \{d_1, \ldots, d_n\}, D^0 \backslash \{d'_1, \ldots, d'_m\}) : \texttt{ref}(t)) = (d : t) \wedge \forall i.(d_i : t) \wedge \forall j.\neg(d'_j : t)$$

A.2.1. *Related Work.* Davies and Pfenning [2000] show an issue arising from the combination of references cells and intersection types. The problem appears if we allow implicitly typed (Curry-style) cell creation, like $\texttt{ref}(1)$. This reference can be given many types, like $\texttt{ref}(1)$, $\texttt{ref}(\text{int})$, $\texttt{ref}(\text{int}\vee\text{bool})$, $\texttt{ref}(\mathbb{1})$. If we allow to give it an intersection of such types, say $\texttt{ref}(\text{int})\wedge\texttt{ref}(\mathbb{1})$, it is possible to assign to it an arbitrary value (if, by subsumption, we see it with type $\texttt{ref}(\mathbb{1})$), but, when we read from it, we expect to read a value of type int (if we see it with type $\texttt{ref}(\text{int})$). In Davies and Pfenning [2000], the solution is to restrict the introduction of intersection types to values and to remove the distributivity rule $(t\to s_1)\wedge(t\to s_2) \leq (t\to s_1\wedge s_2)$. We do not follow such an approach because it has a global impact on the whole system: changing axiomatically the subtyping between function types is not possible in our system. We prefer the simpler approach that consists in having prescriptive types for reference cells. When we create a reference cell, we give enough information to infer a single unique type for the cell contents.

A.3. NONOVERLOADED FUNCTIONS. The calculus introduced in this article let specify several arrow types in λ-abstraction. In this section, we show how to restrict the calculus and the type system to allow only one arrow type. The syntax of λ-abstractions is restricted to

$$\mu f(t\to t).\lambda x.e.$$

To type this expression, we can use the same type system as for our original calculus. It is easy to check that the operational semantics will never introduce overloaded functions if the original expression does not contain any. From that we deduce that the calculus remains sound. However, the interpretation of types as sets of values changes and because of that, Theorem 5.5 no longer holds. To see why, take four types t_1, s_1, t_2, s_2 and consider the type $(t_1 \to s_1) \wedge (t_2 \to s_2)$. Values of this type are closed well-typed expressions of the form $\mu f(t \to s).\lambda x.e$ such that $t \to s \leq t_i \to s_i$ for $i = 1..2$. But $t \to s \leq t_i \to s_i$ can be decomposed into $(t \simeq \mathbb{0}) \vee (t_i \leq t \wedge s \leq s_i)$. The condition is thus equivalent to $(t \simeq \mathbb{0}) \vee (t_1 \vee t_2 \leq t \wedge s \leq s_1 \wedge s_2)$, which is again equivalent to $t \to s \leq (t_1 \vee t_2) \to (s_1 \wedge s_2)$. We have proved that in the restricted calculus, we have the following property:

$$[\![(t_1 \to s_1) \wedge (t_2 \to s_2)]\!]_{\mathcal{V}} = [\![(t_1 \vee t_2) \to (s_1 \wedge s_2),]\!]_{\mathcal{V}},$$

but it is easy to check that

$$(t_1 \to s_1) \wedge (t_2 \to s_2) \leq (t_1 \vee t_2) \to (s_1 \wedge s_2),$$

does not hold in general. This is enough to conclude that Theorem 5.5 does not hold.

To recover Theorem 5.5 and all the other formal results, we need to adapt just one definition. In the new restricted calculus, the type $t \to s$ should describe all the well-typed and closed expressions of the form $\mu f(t' \to s').\lambda x.e$, provided that $t' \to s' \leq t \to s$. This condition can be decomposed into $t \leq t' \wedge s' \leq s$. [11] Following this intuition, we adapt Definition 4.2; if X and Y are subsets of D, we define $X \to Y$ as:

$$X \to Y = \{(X', Y') \in \mathcal{P}(D) \times \mathcal{P}(D) \mid X \subseteq X' \wedge Y' \subseteq Y\}$$

and keep Definition 4.3 unchanged (with the new definition for $X \to Y$ and $\mathbb{E}D = \mathcal{C} + D^2 + \mathcal{P}(D) \times \mathcal{P}(D)$). This modification is enough to establish all the theorems from Section 5 for the restricted calculus. Let us just outline some key modifications we need to do to account for the new system

LEMMA A.2. *Let $(X_i)_{i \in P}, (X_i)_{i \in N}, (Y_i)_{i \in P}, (Y_i)_{i \in N}$ be four families of subsets of D. Then:*

$$\bigcap_{i \in P} X_i \to Y_i \subseteq \bigcup_{i \in N} X_i \to Y_i$$
$$\Longleftrightarrow$$
$$\exists i_0 \in N. X_{i_0} \subseteq \bigcup_{i \in P} X_i \wedge \bigcap_{i \in P} Y_i \subseteq Y_{i_0}.$$

PROOF. Let us prove the $\Rightarrow$ direction. We take $X = \bigcup_{i \in P} X_i$ and $Y = \bigcap_{i \in P} Y_i$. The element (X, Y) is in $\bigcap_{i \in P} X_i \to Y_i$ and so it is also in $\bigcup_{i \in N} X_i \to Y_i$. We can thus find $i_0 \in N$ such that $(X, Y) \in X_i \to Y_i$, that is, $X_{i_0} \subseteq X \wedge Y \subseteq Y_{i_0}$. The other direction is straightforward. □

From this, we learn how to adapt Lemma 6.8:

[11] We could use a more complex decompositionof $t' \to s' \leq t \to s$ as $(t \leq t' \wedge s' \leq s) \vee t \simeq \mathbb{0}$. This would make the development slightly complex without any real benefit.

LEMMA A.3. *Let P and N be two finite subsets of $\mathcal{A}_{\mathbf{fun}}$. Then:*

$$\bigcap_{a\in P}\mathbb{E}(a)\subseteq\bigcup_{a\in N}\mathbb{E}(a)$$

$$\iff$$

$$\exists(t_0{\to}s_0)\in N.\left[\!\!\left[t_0\setminus\left(\bigvee_{t\to s\in P}t\right)\right]\!\!\right]=\varnothing\wedge\left[\!\!\left[\left(\bigwedge_{t\to s\in P}s\right)\setminus s_0\right]\!\!\right]=\varnothing$$

(with the convention $\bigcap_{a\in\varnothing}\mathbb{E}(a)=\mathbb{E}^{\mathbf{fun}}D=\mathcal{P}(D)\times\mathcal{P}(D)$).

Definition 6.9 is adapted by taking:

$$C^{P,N}_{\mathbf{fun}}\ ::=\ \exists t_0{\to}s_0\in N.\begin{cases}\mathcal{N}\left(t_0\wedge\bigwedge_{t\to s\in P}\neg t\right)\in\mathcal{S}\\ \mathcal{N}\left((\neg s_0)\wedge\bigwedge_{t\to s\in P}s\right)\in\mathcal{S}\end{cases}$$

with the following results: Theorem 6.10, Corollary 6.11, Corollary 6.12, Lemma 6.13, all the results from Section 6.3 and Section 6.4, where Lemma 6.21 is modified as follows:

$$[\![t{\to}s]\!]_{\mathcal{V}}=\{(\mu f(t'{\to}s').\lambda x.e)\in\mathcal{V}.\mid t'{\to}s'\leq t{\to}s\}$$

The case for functions in the proof of Lemma 6.27 needs to be adapted as well. The value v, which is produced in this case, is now $v=\mu f(t_0{\to}s_0).\lambda x.fx$, where $t_0=\bigvee_{i=1..n}t_i$ and $s_0=\bigwedge_{i=1..n}s_i$.

Adapting the case for β-reduction in the proof of the Subject Reduction theorem is easy.

The last thing to change is the construction of the universal model (Section 6.7 and Section 6.8). We redefine $\mathbb{E}_f D$ as $\mathcal{C}+D^2+\mathcal{P}_{cf}(D)\times\mathcal{P}_f(D)$ where $\mathcal{P}_{cf}$ denotes the restriction of the powerset to *cofinite* subsets. The terms of the universal model are now generated by the following grammar:

$$d\ ::=\ c\mid(d,d)\mid(\overline{\{d,\ldots,d\}},\{d,\ldots,d\})$$

The predicate $(d:t)$ used to define the set-theoretic interpretation $[\![\_]\!]^0$ is changed with:

$$((\overline{\{d_1,\ldots,d_n\}},\{d'_1,\ldots,d'_m\}):t_1{\to}t_2)=\forall i.\ \neg(d_i:t_1)\wedge\forall j.\ (d'_j:t_2).$$

ACKNOWLEDGMENT. This work greatly benefited from suggestions from and conversation with many people. Foremost we want to say a special thank to Mariangiola Dezani, whose precious help, indefatigable encouragement, and constant appreciation of this work constituted a fundamental ingredient of its preparation. It is impossible to cite all the persons whose remarks, suggestions or simple comments contributed to the final shape of this article, nevertheless among them we want to single out Nick Benton, Luca Cardelli, Pierre-Louis Curien, Haruo Hosoya, Giuseppe Longo, Paul-André Melliès, Benjamin Pierce, Daniele Varacca, Jerôme Vouillon, Phil Wadler, and all the members of the ℂDuce group.

An important contribution was given by various anonymous referees. Foremost, those of JACM whose pertinent remarks encouraged us to write the Section 7 of this

article; we believe that this greatly improved the quality of our presentation. But an equally important role was played by the reviewers of the various conferences where several bricks that form this work were presented: LICS '02, ICFP '03, POPL '05, ICTCS '05.

REFERENCES

AIKEN, A., AND WIMMERS, E. L. 1993. Type inclusion constraints and type inference. In *Proceedings of the 7th ACM Conference on Functional Programming and Computer Architecture* (Copenhagen, Denmark). ACM, New York, 31–41.

AIKEN, A., WIMMERS, E. L., AND LAKSHMAN, T. K. 1994. Soft typing with conditional types. In *Proceedings of the 21ST ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY). ACM, New York, 163–173.

AMADIO, R. M., AND CARDELLI, L. 1993. Subtyping recursive types. *ACM Trans. Program. Lang. Syst. 15,* 4 (Sept.), 575–631.

AMADIO, R., AND CURIEN, P.-L. 1998. *Domains and Lambda Calculi*. Cambridge Tracts in Theoretical Computer Science, vol. 46. Cambridge University Press, Cambridge, MA.

ASPERTI, A., AND LONGO, G. 1991. *Categories, Types and Structures: An Introduction to Category Theory for the Working Computer Scientist*. MIT Press, Cambridge, MA.

BARBANERA, F., DEZANI-CIANCAGLINI, M., AND DE'LIGUORO, U. 1995. Intersection and union types: Syntax and semantics. *Inf. Comput. 119*, 202–230.

BARENDREGT, H., COPPO, M., AND DEZANI-CIANCAGLINI, M. 1983. A filter lambda model and the completeness of type assignment. *J. Symb. Logic 48*, 4, 931–940.

BENZAKEN, V., CASTAGNA, G., AND FRISCH, A. 2003. CDuce: An XML-friendly general purpose language. In *ICFP '03, Proceedings of the 8th ACM International Conference on Functional Programming* (Uppsala, Sweden). ACM New York, 51–63.

CASTAGNA, G. 2005. Semantic subtyping: Challenges, perspectives, and open problems. In *ICTCS 2005, Proceedings of the Italian Conference on Theoretical Computer Science*. Lecture Notes in Computer Science, vol. 3701. Springer-Verlag, New York, 1–20.

CASTAGNA, G., DE NICOLA, R., AND VARACCA, D. 2005. Semantic subtyping for the π-calculus. In *LICS '05, Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, Los Alamitos, CA.

CASTAGNA, G., DE NICOLA, R., AND VARACCA, D. 2007. Semantic subtyping for the π-calculus. *Theoretical Computer Science*. Special issue in honor of Mario Coppo, Mariangiola Dezani-Ciancaglini and Simona Ronchi della Rocca. To appear.

CASTAGNA, G., DEZANI-CIANCAGLINI, M., AND VARACCA, D. 2006. Encoding CDuce into the π-calculus. In *CONCUR 2006, Proceedings of the 17th International Conference on Concurrency Theory*. Lecture Notes in Computer Science, vol. 4137. Springer-Verlag, New York, 310–326.

CASTAGNA, G., AND FRISCH, A. 2005. A gentle introduction to semantic subtyping. In *Proceedings of PPDP '05, the 7th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming.* ACM, New York (full version) and *ICALP '05, 32nd International Colloquium on Automata, Languages and Programming.* Lecture Notes in Computer Science, vol. 3580. Springer-Verlag, New York (summary) (Joint ICALP-PPDP keynote talk).

CASTAGNA, G., GHELLI, G., AND LONGO, G. 1993. A semantics for $\lambda\&$-*early*: A calculus with overloading and early binding. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications* (Utrecht, The Netherlands), M. Bezem and J. Groote, Eds. Lecture Notes in Computer Science, vol. 664. Springer-Verlag, New York, 107–123.

CASTAGNA, G., GHELLI, G., AND LONGO, G. 1995. A calculus for overloaded functions with subtyping. *Inf. Comput. 117,* 1, 115–135.

CDUCE. The CDuce programming language. `http://www.cduce.org`.

COPPO, M., AND DEZANI-CIANCAGLINI, M. 1980. An extension of the basic functionality theory for the λ-calculus. *Notre-Dame J. Formal Logic 21,* 4 (Oct.), 685–693.

DAMM, F. 1994a. Subtyping with union types, intersection types and recursive types II. Research Report 816, IRISA.

DAMM, F. M. 1994b. Subtyping with union types, intersection types and recursive types. In *Theoretical Aspects of Computer Software*, M. Hagiya and J. C. Mitchell, Eds. Vol. 789. Springer-Verlag, New York, 687–706.

DAVIES, R., AND PFENNING, F. 2000. Intersection types and computational effects. In *Proceedings of the the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. ACM, New York.

DEZANI-CIANCAGLINI, M., FRISCH, A., GIOVANNETTI, E., AND MOTOHAMA, Y. 2002. The relevance of semantic subtyping. In *Intersection Types and Related Systems*. Electronic Notes in Theoretical Computer Science *70*, 1.

FREEMAN, T., AND PFENNING, F. 1991. Refinement types for ML. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*. *SIGPLAN Notices 26,* 6 (June), 268–277.

FRISCH, A. 2004. Théorie, conception et réalisation d'un langage de programmation fonctionnel adapté à XML. Ph.D. dissertation. Université Paris 7.

FRISCH, A. 2006. OCaml + XDuce. In *Proceedings of the the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP'06)*. ACM, New York.

FRISCH, A., CASTAGNA, G., AND BENZAKEN, V. 2002. Semantic Subtyping. In *LICS'02, Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, Los Alamitos, CA, 137–146.

GAPAYEV, V., AND PIERCE, B. 2003. Regular object types. In *Proceedings of the ECOOP '03*. Lecture Notes in Computer Science, Springer-Verlag, New York.

HINDLEY, R., AND LONGO, G. 1980. Lambda-calculus models and extensionality. *Zeit. Math. Logik Grund. Math. 26,* 2, 289–319.

HOSOYA, H. 2001. Regular expression types for XML. Ph.D. dissertation, The University of Tokyo, Tokyo, Japan.

HOSOYA, H., FRISCH, A., AND CASTAGNA, G. 2005. Parametric polymorphism for XML. In *POPL '05, Proceedings of the 32nd ACM Symposium on Principles of Programming Languages*. ACM Press, New York.

HOSOYA, H., AND PIERCE, B. 2001. Regular expression pattern matching for XML. In *POPL '01, Proceedings of the 25th ACM Symposium on Principles of Programming Languages*. ACM Press, New York.

HOSOYA, H., AND PIERCE, B. 2003. XDuce: A typed XML processing language. *ACM Trans. Internet Tech. 3,* 2, 117–148.

HOSOYA, H., VOUILLON, J., AND PIERCE, B. 2000. Regular expression types for XML. In *Proceedings of the ICFP '00. SIGPLAN Notices*, *35*, 9.

MACQUEEN, D., PLOTKIN, G., AND SETHI, R. 1986. An ideal model for recursive polymorphic types. *Inf. Cont. 71,* 1/2, 95–130.

MELLIÈS, P.-A., AND VOUILLON, J. 2005. Recursive polymorphic types and parametricity in an operational framework. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)*. IEEE Computer Society Press, Los Alamitos, CA, 82–91.

PFENNING, F. 1993. Refinement types for logical frameworks. In *Informal Proceedings of the Workshop on Types for Proofs and Programs*, H. Geuvers, Ed. Nijmegen, The Netherlands, 285–299.

PIERCE, B. 1991. Programming with intersection types, union types, and polymorphism. Tech. Rep. CMU-CS-91-106, School of Computer Sciente, Carnegie Mellon University.

REYNOLDS, J. C. 1983. Types, abstraction and parametric polymorphism. In *Information Processing*, R. E. A. Mason, Ed. Elsevier Science Publishers B. V. (North-Holland), Amsterdam, The Netherlands, 513–523.

REYNOLDS, J. C. 1991. The coherence of languages with intersection types. In *Theoretical Aspects of Computer Software*, T. Ito and A. R. Meyer, Eds. Lecture Notes in Computer Science, vol. 526. Springer-Verlag, Berlin, Germany, 675–700.

REYNOLDS, J. C. 1996. Design of the programming language Forsythe. Tech. Rep. CMU-CS96-146, Carnegie Mellon University, Pittsburgh, PA, June.

STUDER, T. 2001. A semantics for λ-{}: A calculus with overloading and late-binding. *J. Logic Computation 11,* 4, 527–544.

TSUIKI, H. 1994. A normalizing calculus with overloading and subtyping. In *TACS '94, Proceedings of the Symposium on Theoretical Aspects of Computer Science*. Lecture Notes in Computer Science, vol 789, Springer-Verlag, New York.

VOUILLON, J. 2006. Polymorphic regular tree types and patterns. In *POPL '06, Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, 103–114.

VOUILLON, J., AND MELLIÉS, P.-A. 2004. Semantic types: A fresh look at the ideal model for types. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, 52–63.

RECEIVED MAY 2006; REVISED JULY 2007 AND JUNE 2008; ACCEPTED JUNE 2008