Subtyping-Based Type-Checking for System F with Induction and Coinduction

Rodolphe Lepigre Christophe Raffalli
LAMA, UMR 5127 CNRS - Université Savoie Mont Blanc
{rodolphe.lepigre | christophe.raffalli}@univ-smb.fr

Abstract

We present a type system with subtyping for a strongly normalizing, Curry-style language. Our type constructors include sum and product types, universal and existential quantifiers, inductive and coinductive types. Soundness and strong normalization are shown semantically by constructing a realizability model. We argue that the system is suitable for practical use based on our experience with a prototype implementation. To support this claim, we consider several examples of applications. For instance, the system is able to type recursors for datatypes encoded in the pure λ-calculus, using a representation due to Dana Scott. Other examples include injections between mixed inductive and coinductive types.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory.

Keywords Hilbert's epsilon, induction and coinduction, polymorphism, realizability, strong normalization, subtyping, System F.

1. Introduction

Polymorphism and subtyping are essential for programming in a generic way. They lead to programs that are shorter, easier to understand and hence more reliable. Although polymorphism is widespread among programming languages, only limited forms of subtyping are used in practice (e.g. polymorphic variants [7], modules [18]). Overall, subtyping is useful for both product types (e.g. records or modules) and sum types (e.g. polymorphic variants). It provides canonical injections between a type and its subtypes. For example, unary natural numbers may be defined as a subtype of unary integers.

The downside of subtyping is that it is difficult to incorporate in complex systems such as Haskell or OCaml. For instance, the latter provides polymorphic variants [7] for which complex annotated types are inferred. For example, one would expect

```
fun x → match x with `T → `F | `F → `T
```

to be given type `[`T|`F] → [`T|`F]`. Indeed, the variance of the arrow type conveys enough information: the term can be applied to elements of any subtype of `[`T|`F]` (e.g. `[`T]`) to produce elements of any supertype of `[`T|`F]` (e.g. `[`T|`F|`M]`). OCaml infers the type `[<`T|`F] → [>`T|`F]` in which subtypes and supertypes are explicitly tagged. This is not very natural and hides a complex mechanism involving polymorphic type variables.

In this paper, we show that it is possible to design a practical type system based on subtyping. It can be implemented easily using standard unification techniques and following its typing and subtyping rules (Figures 1 and 2). In particular, the typing and subtyping procedures never backtrack since they are directed by the syntax of terms and types respectively.

Our motivation being the design of a practical programming languages, we consider a reasonably expressive calculus. It is based on System F extended with SML-style records, polymorphic variants, existential types, inductive types and coinductive types. Our system is Curry style (a.k.a. implicitly typed), which means that polymorphic and existential types are interpreted as intersection and union types respectively. Type annotations are provided in the implementation to guide the typing algorithm, however they are not part of the theoretical type system. Working with such a large set of complex features requires us to introduce the technical innovations that are listed and discussed below.

Local subtyping and choice operators for terms. We generalize the usual subtyping relation $A \subset B$ using a *local subtyping* relation $t \in A \subset B$. It is interpreted as "if t has type A then it also has type B". Usual subtyping is then recovered by introducing choice operators $\varepsilon_{x\in A}(t \notin B)$ inspired from Hilbert's epsilon function. Such a choice operator denotes a term of type A such that $t[x := \varepsilon_{x\in A}(t \notin B)]$ does not have type B. If no such term exists, then an arbitrary term of type A can be chosen.[1] We can then take $\varepsilon_{x\in A}(x \notin B) \in A \subset B$ as a definition of $A \subset B$.

Choice operators can be used to replace free variables, and hence suppress the need for contexts. Intuitively, $\varepsilon_{x\in A}(t \notin B)$ denotes a counterexample to the fact that $\lambda x\ t$ has type $A \to B$. Consequently, we can use the rule

$$\frac{t[x := \varepsilon_{x\in A}(t \notin B)] : B}{\lambda x\ t : A \to B}$$

for typing λ-abstractions. This rule corresponds to a proof by contradiction. Its premise is only valid when there is no term u in A such that $t[x := u]$ has type B. The axiom rule is then replaced by a typing rule for choice operators.

$$\frac{}{\varepsilon_{x\in A}(t \notin B) : A}$$

Choice operators for types. We introduce two new type constructors $\varepsilon_X(t \in A)$ and $\varepsilon_X(t \notin A)$ for handling quantifiers in local subtyping relations. They correspond to choice operators satisfying

[1] As our model is based on Girard's reducibility candidates [8, 9], the semantics of a type is never empty (it contain neutral terms).

the denoted properties. For example, $\varepsilon_X(t \notin B)$ is a type such that t does not have type $B[X := \varepsilon_X(t \notin B)]$. Intuitively, $\varepsilon_X(t \notin B)$ is a counter-example to $t : \forall X.B$. Hence, to show $t : \forall X.B$, it will be enough to show $t : B[X := \varepsilon_X(t \notin B)]$. As a consequence, the usual typing rule for universal quantifier introduction is subsumed by the following local subtyping rule.[2]

$$\frac{t \in A \subset B[X := \varepsilon_X(t \notin B)]}{t \in A \subset \forall X.B}$$

In conjunction with local subtyping, choice operators allow the derivation of valid permutations of quantifiers and connectors. For instance, Mitchell's containment axiom [4]

$$\forall X.(A \to B) \subset (\forall X.A) \to (\forall X.B)$$

can be easily derived. Consequently, our type system mostly contains typing and subtyping rules that are syntax-directed. In particular, we do not have a transitivity rule for subtyping. It is not a problem since such a rule cannot be implemented. Indeed, it would require the system to guess an intermediate type. Transitivity is generally admissible in subtyping systems. In our system however, it is an open problem whether a form of transitivity is admissible.

Uniform ordinal induction for sized types. Inductive and coinductive types are generally handled using types $\mu X.F(X)$ and $\nu X.F(X)$ denoting the least and greatest fixpoint of a covariant parametric type F. Here we use sized types $\mu_\kappa X.F(X)$ and $\nu_\kappa X.F(X)$ [2, 11, 28], where κ denotes an ordinal. Intuitively, these types correspond to κ iterations of F on the types $\bot$ and $\top$ respectively. These types can hence be unfolded to $F(\mu_\tau X.F(X))$ and $F(\nu_\tau X.F(X))$, where τ is an ordinal smaller that κ. When κ is maximal, the usual semantics is recovered.

In this paper, we introduce a uniform induction rule for local subtyping. It is able to deal with many inductive and coinductive types at once. As a consequence, our system is suitable for handling subtyping between mixed inductive and coinductive types.

Facing undecidability. Typing and subtyping are likely to be undecidable in our system. Indeed, it contains Mitchell's variant of System F [4], for which both typing and subtyping are undecidable [29]. We believe that there are no practical complete semi-algorithms for such extensions of System F. Instead, we propose an incomplete semi-algorithm that may fail or even diverge on a typable program. In practice we almost never meet non termination, but even in such an eventuality, the user can interrupt the program to obtain a standard type error message. The typing rules being syntax-directed, type-checking can only diverge when checking a local subtyping judgment. In this case, a relevant error message can be built using the last applied typing rule.

We conjecture that our system is complete (i.e. may succeed on all typable programs) provided that enough type annotations are given. In practice, the required amount of annotations correspond to what a human would need to understand the program (see Section 5). Our experiments indicate that our system provides a similar user experience to statically typed functional language like OCaml or Haskell. In fact, such languages also require type annotations for advanced features like polymorphic recursion.

Properties of the system. The language and type system defined in Section 2 have three main properties: strong normalization, type safety and logical consistency (Theorems 3, 4 and 5). They follow from the construction of a realizability model presented in Section 3, and are consequences of the adequacy lemma (Theorem 2). This central result establishes the compatibility of the model with the language and type system.

[2] Note that no freshness constraint is required thanks to choice operators.

Implementation. As a proof of concept, we implemented a toy programming language based on our system. It is called SubML and is available online [16]. Aside from a few subtleties described in Section 4, the implementation is straightforward and remains very close to the typing and subtyping rules of Figures 1 and 2. Although the system is relatively expressive, its simplicity allows for a very concise implementation. The main type-checking functions require less than 400 lines of OCaml code. The full implementation, including parsing, evaluation and LaTeX pretty printing is less that 3000 lines long.

SubML provides literate programming features (inspired by PhoX [23]) that can be used to generate LaTeX documents. In particular, the examples presented in Section 5 (including prooftrees) have been generated using SubML and are therefore machine checked. Moreover, many examples of programs (more that 1000 lines of code) are provided as part of SubML's standard library. They are not included in this paper but can be used to verify that the system is indeed usable in practice.

Applications. In addition to classical examples, our system allows for applications that we find very interesting (see Section 5). As a first example, we can program with the Church encoding of algebraic data types. Although this have little practical interest (if any), it requires the full power of System F and is a good test suite for polymorphism.

Church encoding is known for having a bad time complexity. To solve this problem, Dana Scott proposed a better encoding using a combination of polymorphism and inductive types [1]. For instance, the type of natural numbers can be defined as

$$\mathbb{N} = \mu X.\forall Y.((X \to Y) \to Y \to Y)$$

where $\mu X.F(X)$ denotes the least fixpoint of a covariant parametric type F. Contrary to Church numerals, this encoding admits a constant time predecessor of type $\mathbb{N} \to \mathbb{N}$.

In general, recursion for Scott encoding requires some specific typing rules using a recursor like in Gödel's System T. In contrast, our system is able to derive the typing of a recursor encoded as a λ-term, without extending the language. This recursor was shown to the second author by Michel Parigot. We then adapted it to other algebraic data types, showing that Scott encoding can be used to program in a strongly normalisable system with the expected asymptotic complexity. In general, recursors for Scott encoded data require a non-trivial interaction between polymorphism and inductive types.

We also discovered a surprising coiterator for streams encoded using an existentially quantified type as an internal state.

$$\mathrm{Stream}(A) = \nu X.\exists S.S \times (S \to A \times X)$$

Here $\nu X.F(X)$ denotes the greatest fixpoint of a covariant parametric type F. The type S can indeed be though of as an abstract internal state, as it must be used to progress in the computation of the stream. Contrary to Church or Scott encoding of datatypes, the product that we use in the definition of streams does not need to be encoded using polymorphism. As a consequence, our definition of streams may have practical interest.

This leads to the following open question: what are the derivable induction and coinduction schemes in our system? A first step, is to consider subtyping in the context of mixed induction and coinduction. For instance, our system is able to derive subtyping relations such as

$$\mu X.\nu Y.F(X,Y) \subset \nu Y.\mu X.F(X,Y)$$

for a given covariant type F with two parameters. When we restrict ourselves to types without universal and existential quantifiers, our experiments tend to indicate that our system can be extended to-

ward a complete subtyping relation using the size change principle [14]. This has already been implemented in SubML [16].

As another example of application, it is possible to take advantage of the presence of choice operators in types to provide type abstractions for universal types. Such a feature is required to be able to annotate subterms of polymorphic functions with their types. This enables us to write

$$\mathrm{Id} : \forall X.X \to X = \Lambda X \lambda x : X.x$$

like in Church style (a.k.a. explicitly typed) systems.

An even more interesting application of the choice operators in types is the dot notation for existential types. It allows the encoding of a module system based on records. For instance, we can encode an ML signature for isomorphisms with the following type.

$$\mathrm{Iso} = \exists T.\exists U.\{f : T \to U; g : U \to T\}$$

For any term $h : \mathrm{Iso}$, the following syntactic sugars can be defined to allow access to the abstract types T and U.

$$h.T = \varepsilon_T(h \in \exists U.\{f : T \to U; g : U \to T\})$$
$$h.U = \varepsilon_U(h \in \exists T.\{f : T \to U; g : U \to T\})$$

The first choice operator denotes a type T such that h inhabits the type $\exists U\{f : T \to U; g : U \to T\}$. As our system never infers polymorphic or existential types, we can rely on the name given to the bound variables by the user. This new approach to abstract types seems simpler than previous work like [5].

Related work. The language presented in this paper is an extension of John Mitchell's System F [4], which itself extends Jean-Yves Girard and John Reynolds's System F [8, 25] with subtyping. Our subtyping algorithm extends previous work [3, 21] to support mixed induction and coinduction with polymorphic and existential types. In particular, we improve on an unpublished work of the second author [24].

Our system is related to sized types [11] since we annotate inductive and coinductive types with ordinals. Such a technique has already been used for handling induction and coinduction [2, 28], but it is here extended with a more general induction principle. Sized-types fit very well in our system as they are closely related to subtyping. In particular, iterations of a fixpoints to related ordinals can be related via subtyping. In our context, sized types are not directly used for termination checking.

Subtyping has been extensively studied in the context of ML-like languages, starting with the work of Roberto Amadio and Luca Cardelli [3]. Recent work includes the MLsub system [6], which extends unification to handle subtyping constraints. It relies on a flow analysis distinction between input types and output types borrowed from the PhD thesis of François Pottier [22].

2. Language and Type System

Our language is formed using three syntactic entities: terms, types and ordinals. In this paper, syntactic ordinals are used to annotate inductive and coinductive types.

Definition 1. *The set of syntactic ordinals $\mathcal{O}$ is built from a set of ordinal variables $\{\alpha, \beta, \gamma \dots\}$ using the following BNF grammar.*

$$\kappa, \tau \ ::= \infty \ \mid \ \alpha_{<\kappa} \ \mid \ \alpha_{\leq\kappa}$$

The symbol ∞ denotes the largest ordinal of our semantics, while other symbols correspond to variables with inequality constraints.

Although our system is Curry-style (a.k.a. implicitly typed), terms and types need to be defined simultaneously due to the presence of choice operators in terms.

Definition 2. *The set of terms Λ and the set of types $\mathcal{F}$ are built from a set of λ-variables $\{x, y, z \dots\}$ and a set of propositional variables $\{X, Y, Z \dots\}$ using the following BNF grammars.*

$$\begin{aligned} t, u \ ::= & \ x \mid \lambda x.t \mid t\ u \mid \{(l_i = t_i)_{i\in I}\} \mid t.l \mid C\ t \\ & \mid [t \mid (C_i \to t_i)_{i\in I}] \mid \varepsilon_{x\in A}(t \notin B) \end{aligned}$$

$$\begin{aligned} A, B \ ::= & \ X \mid \{(l_i : A_i)_{i\in I}\} \mid [(C_i \ of \ A_i)_{i\in I}] \mid A \Rightarrow B \\ & \mid \forall X.A \mid \exists X.A \mid \mu_\tau X.A \mid \nu_\tau X.A \\ & \mid \varepsilon_X(t \in A) \mid \varepsilon_X(t \notin A) \end{aligned}$$

Record fields are named using labels of the set $\mathcal{L} = \{l, l_1, l_2 \dots\}$ and constructors are taken in the set $\mathcal{C} = \{C, C_1, C_2 \dots\}$. A term of the form $\varepsilon_{x\in A}(t \notin B)$ should not have any free term variables. For example the term $\lambda y.\varepsilon_{x\in A}(y\ x \notin B)$ is invalid. We also forbid types of the form $\mu_\tau X.A$ or $\nu_\tau X.A$ with negative occurrences of X in A. Occurrences of type variables in terms are neither positive nor negative, therefore they cannot be bound by fixpoints.

The term language contains the usual syntax of the pure λ-calculus extended with records, projections, constructors and pattern matching (see the reduction rules below). A term of the form $\varepsilon_{x\in A}(t \notin B)$ corresponds to a choice operator denoting a closed term u of type A such that $t[x := u]$ does not have type B. The restriction to closed choice operators is necessary for their interpretation in the semantics.

In addition to the usual types of System F, our system provides sums and products (corresponding to variants and records), existential types, inductive types and coinductive types. Our least and greatest fixpoints carry size information in the form of syntactic ordinals. Two choice operators $\varepsilon_X(t \in A)$ and $\varepsilon_X(t \notin A)$ are also provided. Like choice operators in terms, they correspond to witnesses of the property they denote and will be interpreted as such in the semantics. Contrary to choice operators in terms, they do not need to be closed.

Definition 3. *The one step reduction relation $(\succ) \subseteq \Lambda \times \Lambda$ is defined as the contextual closure of the following rules.*

$$\begin{aligned} (\lambda x.t)u &\succ t[x := u] && \\ \{(l_i : t_i)_{i\in I}\}.l_j &\succ t_j && \text{if } j \in I \\ [C_j\ u \mid (C_i \to t_i)_{i\in I}] &\succ t_j\ u && \text{if } j \in I \\ \{(l_i : t_i)_{i\in I}\}.l_j &\succ \Omega && \text{if } j \notin I \\ [C_j\ u \mid (C_i \to t_i)_{i\in I}] &\succ \Omega && \text{if } j \notin I \\ [(\lambda x.t) \mid (C_i \to t_i)_{i\in I}] &\succ \Omega && (\lambda x.t).l \succ \Omega \\ \{(l_i = t_i)_{i\in I}\}\ u &\succ \Omega && (C\ t)\ u \succ \Omega \\ [\{(l_i = t_i)_{i\in I}\} \mid (C_i \to t_i)_{i\in I}] &\succ \Omega && (C\ t).l \succ \Omega \end{aligned}$$

Note that bad terms corresponding to runtime errors are reduced to a diverging term Ω for termination to subsume type safety.

Notations and conventions. To lighten the syntax and reduce the need for parentheses, we consider binders to have the lowest priority. This means that $\lambda x.x\ x$ and $\forall X.A \to B$ denote $\lambda x.(x\ x)$ and $\forall X.(A \to B)$ respectively. Binders can be grouped, hence $\lambda x\ y.t$ denotes $\lambda x.\lambda y.t$ and $\forall X\ Y.A$ denotes $\forall X.\forall Y.A$. We use the notations $\mu X.A$ and $\nu X.A$ for the usual inductive and coinductive types defined as $\mu_\infty X.A$ and $\nu_\infty X.A$ respectively. We sometimes use the letter F to denote a type with one parameter $X \mapsto A$. We will hence write $F(\mu_\kappa F)$ for $A[X := \mu_\kappa X.A]$. Pattern matching and records can be expanded to obtain expressions of the forms $[t \mid C_1 \to t_1 \mid \dots \mid C_n \to t_n]$ and $\{l_1 = t_1; \dots; l_n = t_n\}$. Similar notations are used for the corresponding types. Labels and constructors should always be unique in such structures, consequently their order is irrelevant. In a pattern matchings we will sometimes use the notation $Cx \to t$ to denote $C \to \lambda x.t$. If C is a constructor, the notation $t.C$ corresponds to the term $[t \mid C \to \lambda x.x]$.

$$\frac{\lambda x\ t \in A \to B \subset C \qquad t[x := \varepsilon_{x\in A}(t \notin B)] : B}{\lambda x\ t : C} \to_i$$

$$\frac{t : A \to B \qquad u : A}{t\ u : B} \to_e$$

$$\frac{\{(l_i = t_i)_{i\in I}\} \in \{(l_i : A_i)_{i\in I}\} \subset B \qquad (t_i : A_i)_{i\in I}}{\{(l_i = t_i)_{i\in I}\} : B} \times_i$$

$$\frac{t : \{l : A\}}{t.l : A} \times_e$$

$$\frac{t : A \qquad C\ t \in [C \text{ of } A] \subset B}{C\ t : B} +_i$$

$$\frac{t : [(C_i \text{ of } A_i)_{i\in I}] \qquad (t_i : A_i \to B)_{i\in I}}{[t \mid (C_i \to t_i)_{i\in I}] : B} +_e$$

$$\frac{\varepsilon_{x\in A}(t \notin B) \in A \subset C}{\varepsilon_{x\in A}(t \notin B) : C}\ \varepsilon$$

Figure 1. Typing rules.

Definition 4. *The syntactic ordinals are equipped with ordering relations $\kappa \le \tau$ and $\kappa < \tau$ defined using the following rules.*

$$\frac{}{\kappa \le \infty} \qquad \frac{}{\kappa \le \kappa} \qquad \frac{\kappa \le \tau}{\alpha_{<\kappa} < \tau}$$

$$\frac{\kappa \le \tau}{\alpha_{<\kappa} \le \tau} \qquad \frac{\kappa < \tau}{\alpha_{\le\kappa} < \tau} \qquad \frac{\kappa \le \tau}{\alpha_{\le\kappa} \le \tau}$$

Definition 5. *The types are equipped with an ordering relation $A \le B$ comparing types with the same structure. It is defined using the following rule system.*

$$\frac{}{A \le A} \qquad \frac{A_2 \le A_1 \qquad B_1 \le B_2}{A_1 \to B_1 \le A_2 \to B_2}$$

$$\frac{A \le B}{\forall X.A \le \forall X.B} \qquad \frac{(A_i \le B_i)_{i\in I}}{\{(l_i : A_i)_{i\in I}\} \le \{(l_i : B_i)_{i\in I}\}}$$

$$\frac{A \le B}{\exists X.A \le \exists X.B} \qquad \frac{(A_i \le B_i)_{i\in I}}{[(C_i \text{ of } A_i)_{i\in I}] \le [(C_i \text{ of } B_i)_{i\in I}]}$$

$$\frac{\kappa \le \tau \qquad A \le B}{\mu_\kappa X.A \le \mu_\tau X.B} \qquad \frac{\tau \le \kappa \qquad A \le B}{\nu_\kappa X.A \le \nu_\tau X.B}$$

The ordering relations over ordinals and types only contain syntax-directed rules. As a consequence, they can be immediately implemented as a deterministic and terminating function. In particular, this is the case because they do not include transitivity rules. The relation $A \le B$ will be used by our general subtyping procedure for checking subtyping up to ordinals. Note the contravariance on ordinals for coinductive types.

Definition 6. *In addition to usual typing judgments of the form $t : A$, we introduce local subtyping judgements of the form $t \in A \subset B$ meaning "if t has type A, then it also has type B". Usual subtyping judgments of the form $A \subset B$ are then encoded using $\varepsilon_{x\in A}(x \notin B) \in A \subset B$. The typing and subtyping rules of the system are given in Figures 1 and 2 respectively, with the exception of the induction rule that is discussed below.*

Thanks to local subtyping judgements, quantifiers can be handled in the subtyping part of the system. Using choice operators allows for valid commutations of quantifiers with other connectors, without changing the syntax-directed nature of the system. Our judgements have the peculiarity of not carrying any context.[3] Indeed, choice operators are used to substitute free variables denoting terms or types.

Induction rule. We now give a local subtyping allowing proofs by ordinal induction in proof trees.

$$\left[\ \frac{(\tau_i \le \kappa_i)_{i\in I} \qquad (\tau_i)_{i\in I} <^* (\alpha_{i\le\kappa_i})_{i\in I}}{u \in (A \subset B)[(\kappa_i := \tau_i)_{i\in I}]}\ H_m\ \right]$$

$$\vdots$$

$$\frac{(A \subset B)[(\kappa_i := \alpha_{i\le\kappa_i})_{i\in I}]}{t \in A \subset B}\ I_m$$

For the rule to apply, the ordinals $\{\kappa_i\}_{i\in I}$ have to appear in A or in B, and I has to be non-empty. The constrained ordinal variables $\{\alpha_{i\le\kappa_i}\}_{i\in I}$ are fresh in the conclusion. In other words, they cannot appear in A, B or t. The name of the rule I_m contains a number m (unique in the branch of the proof) that maintain the link with the corresponding axioms labeled H_m. The relation $<^*$ on ordinal tuples denotes any lexicographic ordering. With this definition, correctness of the system is ensured: the axiom H_m are valid induction hypothesis on smaller tuples. But as discussed later, it is not enough for the completeness of the implementation.

As an example, the judgment $t \in \mu_\tau X.A \subset \nu_\kappa X.B$ is a valid conclusion for the induction rule. In this case, the application of the induction rule corresponds to a proof of $\mu_\alpha X.A \subset \nu_\beta X.B$ by induction on the tuple of ordinals (α, β). We can assume that these ordinals verify $\alpha \le \tau$ and $\beta \le \kappa$. This is expressed in the premise of I_m, which consists in a general subtyping judgement. It is not restricted to a local subtyping judgement over t, and hence the induction hypothesis H_m can be applied to local subtyping judgements over arbitrary terms. Figure 4 provides an example of a proof using the induction rule.

3. Realizability Semantics

In this section, we build a realizability model that is shown adequate with our type system. In particular, formulas are interpreted using sets of strongly normalizing pure terms.

Definition 7. *A term is said pure when it does not contain choice operators $\varepsilon_{x\in A}(t \notin B)$. We denote $[\![\Lambda]\!] \subseteq \Lambda$ the set of pure terms.*

Definition 8. *We say that a pure term $t \in [\![\Lambda]\!]$ is strongly normalizing if there is no infinite sequence of reduction starting from t. We denote $\mathcal{N} \subseteq [\![\Lambda]\!]$ the set of strongly normalizing pure terms.*

Definition 9. *The set $\mathcal{H}$ of head contexts (i.e. terms with a hole in head position) is generated by the following grammar.*

$$H \ ::= \ [\,] \ \mid \ H\ t \ \mid \ H.l \ \mid \ [H \mid (C_i \to t_i)_{i\in I}]$$

Given a term t and a head context H we denote $H[t]$ the term formed by plugging t into the hole of H.

Definition 10. *We say that a set of pure terms $\Phi \subseteq [\![\Lambda]\!]$ is saturated if the following conditions hold.*

- *If $H[t[x := u]] \in \Phi$ and $u \in \mathcal{N}$ then $H[(\lambda x.t)\ u] \in \Phi$.*
- *If $H[t\ u] \in \Phi$ and for all $i \in I$ we have $t_i \in \mathcal{N}$, then*
$$H[[D\ u \mid D \to t \mid (C_i \to t_i)_{i\in I}]] \in \Phi.$$
- *If $H[t] \in \Phi$ and for all $i \in I$ we have $t_i \in \mathcal{N}$, then*
$$H[\{l = t; (l_i = t_i)_{i\in I}\}.l] \in \Phi.$$

[3] However, ordinal variable constraints may be seen as a form of context.

$$\frac{A \leq B}{t \in A \subset B} = \qquad \frac{\varepsilon_{x\in A_2}(t\ x \notin B_2) \in A_2 \subset A_1 \qquad t\ \varepsilon_{x\in A_2}(t\ x \notin B_2) \in B_1 \subset B_2}{t \in A_1 \to B_1 \subset A_2 \to B_2} \to$$

$$\frac{t \in A[X := U] \subset B}{t \in \forall X.A \subset B}\ \forall_l \qquad \frac{t \in A \subset B[X := \varepsilon_X(t \notin B)]}{t \in A \subset \forall X.B}\ \forall_r \qquad \frac{I_2 \subseteq I_1 \qquad (t.l_i \in A_i \subset B_i)_{i\in I_2}}{t \in \{(l_i : A_i)_{i\in I_1}\} \subset \{(l_i : B_i)_{i\in I_2}\}} \times$$

$$\frac{t \in B \subset A[X := U]}{t \in B \subset \exists X.A}\ \exists_r \qquad \frac{t \in B[X := \varepsilon_X(t \in B)] \subset A}{t \in \exists X.B \subset A}\ \exists_l \qquad \frac{I_1 \subseteq I_2 \qquad (t.C_i \in A_i \subset B_i)_{i\in I_1}}{t \in [(C_i : A_i)_{i\in I_1}] \subset [(C_i : B_i)_{i\in I_2}]} +$$

$$\frac{t \in A \subset F(\mu F)}{t \in A \subset \mu F}\ \mu_r^\infty \qquad \frac{t \in A \subset F(\mu_\tau F) \qquad \tau < \kappa}{t \in A \subset \mu_\kappa F}\ \mu_r \qquad \frac{t \in F(\mu_{\alpha<\kappa} F) \subset B}{t \in \mu_\kappa F \subset B}\ \mu_l^*$$

$$\frac{t \in F(\nu F) \subset B}{t \in \nu F \subset B}\ \nu_l^\infty \qquad \frac{t \in F(\nu_\tau F) \subset B \qquad \tau < \kappa}{t \in \nu_\kappa F \subset B}\ \nu_l \qquad \frac{t \in A \subset F(\nu_{\alpha<\kappa} F)}{t \in A \subset \nu_\alpha F}\ \nu_r^*$$

($*$) Variable $\alpha_{<\kappa}$ must not appear in the conclusion of the μ_l and ν_r rules.

Figure 2. Subtyping rules (excluding the induction rule).

Lemma 1. *The set $\mathcal{N}$ is saturated.*

Proof. The proof can be carried out using techniques found in [13], by first showing that arbitrary infinite reductions can be transformed into infinite standard reductions. The proof of a similar result can be found in the work of Karim Nour and Khelifa Saber [20], however it is more complex due to specific reduction rules for classical logic and commutative cuts. Finally, a general framework for proofs using reducibility candidates can be found in the work of Colin Riba [26, 27]. □

Definition 11. *The set of neutral terms $\mathcal{N}_0$ is defined as the smallest set of terms such that:*

1. *For every λ-variable x we have $x \in \mathcal{N}_0$.*
2. *For every $u \in \mathcal{N}$ and $t \in \mathcal{N}_0$ we have $t\,u \in \mathcal{N}_0$.*
3. *For every $l \in \mathcal{L}$ and $t \in \mathcal{N}_0$ we have $t.l \in \mathcal{N}_0$.*
4. *For every $(C_i, t_i)_{i\in I} \in (\mathcal{C} \times \mathcal{N})^I$ and $t \in \mathcal{N}_0$ we have*
$$[t \mid (C_i \to t_i)_{i\in I}] \in \mathcal{N}_0.$$

Lemma 2. *We have $\mathcal{N}_0 \subseteq \mathcal{N}$.*

Proof. Simple induction on the definition of $\mathcal{N}_0$, using the fact that the terms of $\mathcal{N}_0$ never contain a redex in their left branch. □

Definition 12. *Given two sets $\Phi_1 \subseteq \Lambda$ and $\Phi_2 \subseteq \Lambda$, we define $(\Phi_1 \to \Phi_2) \subseteq \Lambda$ as follows.*
$$\Phi_1 \to \Phi_2 = \{t \in \Lambda \mid \forall u \in \Phi_1,\ t\,u \in \Phi_2\}$$

Lemma 3. *Let Φ_1, Φ_2, Ψ_1, $\Psi_2 \subseteq \Lambda$ be sets of terms such that $\Phi_2 \subseteq \Phi_1$ and $\Psi_1 \subseteq \Psi_2$. We have $(\Phi_1 \to \Psi_1) \subseteq (\Phi_2 \to \Psi_2)$.*

Proof. Immediate by definition. □

Lemma 4. *We have $\mathcal{N}_0 \subseteq (\mathcal{N} \to \mathcal{N}_0) \subseteq (\mathcal{N}_0 \to \mathcal{N}) \subseteq \mathcal{N}$.*

Proof. By Lemma 2 we know that $\mathcal{N}_0 \subseteq \mathcal{N}$ and hence we obtain $(\mathcal{N} \to \mathcal{N}_0) \subseteq (\mathcal{N}_0 \to \mathcal{N})$ using Lemma 3. If we take $t \in \mathcal{N}_0$, then by definition $t\,u \in \mathcal{N}_0$ for every $u \in \mathcal{N}$. Therefore we obtain $\mathcal{N}_0 \subseteq (\mathcal{N} \to \mathcal{N}_0)$. Finally, if we take $t \in (\mathcal{N}_0 \to \mathcal{N})$ then by definition $t\ x \in \mathcal{N}$ since $x \in \mathcal{N}_0$. Hence $t \in \mathcal{N}$, which gives us $(\mathcal{N}_0 \to \mathcal{N}) \subseteq \mathcal{N}$. □

To simplify the definition of the semantics, the syntax of terms, formulas and ordinals is extended with elements of their respective models (Definition 13). Hence, we first need to give the domain of interpretation for each of the three syntactic category.

- A closed term $t \in \Lambda$ is interpreted by a pure term $[\![t]\!] \in [\![\Lambda]\!]$. We therefore interpret terms with choice operators using (possibly open) pure terms.
- A formula $A \in \mathcal{F}$ is interpreted by a saturated set of pure terms $[\![A]\!]$ such that $\mathcal{N}_0 \subseteq [\![A]\!] \subseteq \mathcal{N}$. We denote $[\![\mathcal{F}]\!]$ the set of every such type interpretation.
$$[\![\mathcal{F}]\!] = \{\Phi \subseteq [\![\Lambda]\!] \mid \Phi \text{ saturated},\ \mathcal{N}_0 \subseteq \Phi \subseteq \mathcal{N}\}$$
- A syntactic ordinal $\kappa \in \mathcal{O}$ will be interpreted by an ordinal $[\![\kappa]\!] \in [\![\mathcal{O}]\!]$, where $[\![\mathcal{O}]\!]$ is the set of all the ordinals that are smaller or equal to the cardinal of $2^{[\![\mathcal{F}]\!]}$.

Definition 13. *The sets of* parametric *terms, formulas and ordinals are formed by extending the syntax of ordinals with the elements of $[\![\mathcal{O}]\!]$ and the syntax of formulas with the elements of $[\![\mathcal{F}]\!]$. The corresponding sets will be denoted Λ^*, $\mathcal{F}^*$ and $\mathcal{O}^*$. Terms do not need to be extended directly. However, the definition of $\mathcal{F}^*$ and $\mathcal{O}^*$ impacts the definition of Λ^* since terms and formulas are defined mutually inductively.*

Definition 14. *A closed parametric term (resp. formula, resp. syntactic ordinal) is a parametric term (resp. formula, resp. ordinal) with no free type variables and no free ordinal variables.*

Definition 15. *The semantics $[\![t]\!]$ (resp. $[\![\kappa]\!]$, resp. $[\![A]\!]$) of a closed parametric term $t \in \Lambda^*$ (resp. ordinal $\kappa \in \mathcal{O}^*$, resp. type $A \in \mathcal{F}^*$) is defined in Figure 3. The semantics of terms and types need to be defined mutually inductively because of the choice operators.*

Note that the semantical interpretation of syntactic ordinals is straightforward because they mostly denote variables, which are not interpreted directly. Indeed, closed parametric object are built from non-parametric ones by substituting their free variables with closed parametric objects (e.g. elements of the semantics). As ordinal variables carry constraints, we will need to restrict ourselves to substitutions satisfying these constraints (see Definition 16).

In the interpretation of a choice operators $\varepsilon_{x\in A}(t \notin B)$, it is important that no λ-variable other that x is bound in t. This is enforced by a syntactic restriction that was given in Definition 2. It is absolutely necessary to obtain a well-defined term model. Without this restriction, a term $[\![\lambda y.\varepsilon_{x\in A}(t \notin B)]\!]$ with y free in t or B would correspond to a function that is not always definable using a pure term. In this case, our model would have circular and hence invalid definitions. Note that the axiom of choice is required to interpret all the choice operators.

$$
\begin{aligned}
[\![t]\!] &= t \text{ if } t \in [\![\Lambda]\!] \\
[\![x]\!] &= x \\
[\![t\,u]\!] &= [\![t]\!]\,[\![u]\!] \\
[\![\lambda x.t]\!] &= \lambda x.[\![t]\!] \\
[\![C\,u]\!] &= C\,[\![u]\!] \\
[\![[u \mid (C_i \to t_i)_{i\in I}]]\!] &= [[\![u]\!] \mid (C_i \to [\![t_i]\!])_{i\in I}] \\
[\![\{(l_i = t_i)_{i\in I}\}]\!] &= \{(l_i = [\![t_i]\!])_{i\in I}\} \\
[\![\varepsilon_{x\in A}(t \notin B)]\!] &= \begin{cases} u \in [\![A]\!] \text{ such that } [\![t[x:=u]]\!] \notin [\![B]\!] \\ t \in \mathcal{N}_0 \text{ if there is no such } u \end{cases} \\
\\
[\![o]\!] &= o \text{ if } t \in [\![\mathcal{O}]\!] \\
[\![\infty]\!] &= 2^{[\![\mathcal{F}]\!]} = \max([\![\mathcal{O}]\!])
\end{aligned}
$$

$$
\begin{aligned}
[\![\Phi]\!] &= \Phi \text{ if } \Phi \in [\![\mathcal{F}]\!] \\
[\![A \Rightarrow B]\!] &= [\![A]\!] \to [\![B]\!] \\
[\![\{l_i : A_i\}_{i\in I\neq}]\!] &= \begin{cases} \{t \in \Lambda \mid \forall i \in I,\ t.l_i \in [\![A_i]\!]\} \text{ if } I \neq \emptyset \\ \mathcal{N} \text{ otherwise} \end{cases} \\
[\![[C_i : A_i]_{i\in I}]\!] &= \cap_{\Phi\in[\![\mathcal{F}]\!]}\{t \in \Lambda \mid \forall (t_i \in ([\![A_i]\!] \to \Phi))_{i\in I}, \\
&\qquad\qquad [t \mid (C_i \to t_i)_{i\in I}] \in \Phi\} \\
[\![\forall X A]\!] &= \cap_{\Phi\in[\![\mathcal{F}]\!]}[\![A[X:=\Phi]]\!] \\
[\![\exists X A]\!] &= \cup_{\Phi\in[\![\mathcal{F}]\!]}[\![A[X:=\Phi]]\!] \\
[\![\mu_\kappa X A]\!] &= \cup_{o<[\![\kappa]\!]}[\![A[X:=\mu_o X A]]\!] \\
[\![\nu_\kappa X A]\!] &= \cap_{o<[\![\kappa]\!]}[\![A[X:=\nu_o X A]]\!] \\
[\![\varepsilon_X(t \in A)]\!] &= \begin{cases} \Phi \in [\![\mathcal{F}]\!] \text{ s.t. } [\![t[X:=\Phi]]\!] \in [\![A[X:=\Phi]]\!] \\ \mathcal{N} \text{ if there is no such } \Phi \end{cases} \\
[\![\varepsilon_X(t \notin A)]\!] &= \begin{cases} \Phi \in [\![\mathcal{F}]\!] \text{ s.t. } [\![t[X:=\Phi]]\!] \notin [\![A[X:=\Phi]]\!] \\ \mathcal{N} \text{ if there is no such } \Phi \end{cases}
\end{aligned}
$$

Figure 3. Semantical interpretation of closed parametric terms, semantic ordinals and types.

Theorem 1. *For every closed parametric term $t \in \Lambda^*$ (resp. ordinal $\kappa \in \mathcal{O}^*$, resp. type $A \in \mathcal{F}^*$) we have $[\![t]\!] \in [\![\Lambda]\!]$ (resp. $[\![\kappa]\!] \in [\![\mathcal{O}]\!]$, resp. $[\![A]\!] \in [\![\mathcal{F}]\!]$).*

Proof. We do a proof by induction. For terms, all the cases are immediate by induction hypothesis. If $u = [\![\varepsilon_{x\in A}(t \notin B)]\!]$ then we have $u \in \mathcal{N} \subseteq [\![\Lambda]\!]$ since $\mathcal{N}_0 \subseteq \mathcal{N}$ by Lemma 2 and $[\![A]\!] \subseteq \mathcal{N}$ by induction hypothesis. For ordinals, the proof is immediate by definition. The saturation is easy for types because the closure of the definition by head context is exactly what is needed. For the inclusions, using Lemma 1 we obtain $\mathcal{N} \in [\![\mathcal{F}]\!]$, hence the proof is immediate for types of the forms Φ, $\{\}$, $\varepsilon_X(t \in A_X)$ or $\varepsilon_X(t \notin A_X)$. The remaining cases are given below.

- If it is of the form $A \Rightarrow B$, then we have $\mathcal{N}_0 \subseteq [\![A]\!] \subseteq \mathcal{N}$ and $\mathcal{N}_0 \subseteq [\![B]\!] \subseteq \mathcal{N}$ by induction hypothesis. We need to show that $\mathcal{N}_0 \subseteq [\![A \to B]\!] \subseteq \mathcal{N}$. By Lemma 4, it is enough to show $(\mathcal{N} \to \mathcal{N}_0) \subseteq [\![A \to B]\!] \subseteq (\mathcal{N}_0 \to \mathcal{N})$. We can hence conclude using Lemma 3.
- If it is of the form $\{(l_i : A_i)_{i\in I\neq\emptyset}\}$ then we know that for all $i \in I$ we have $\mathcal{N}_0 \subseteq [\![A_i]\!] \subseteq \mathcal{N}$ by induction hypothesis. Let us take $t \in \mathcal{N}_0$ and show that $t \in [\![\{(l_i : A_i)_{i\in I\neq\emptyset}\}]\!]$. We take $j \in I \neq \emptyset$ and show that $t.l_j \in [\![A_j]\!]$. It is enough to show that $t.l_j \in \mathcal{N}_0$, which follows by definition of $\mathcal{N}_0$. Let us now take $t \in [\![\{(l_i : A_i)_{i\in I\neq\emptyset}\}]\!]$ and show that $t \in \mathcal{N}$. Since $I \neq \emptyset$ we can take $j \in I$ and by definition $t.l_j \in [\![A_j]\!] \subseteq \mathcal{N}$. Since $t.l_j$ is strongly normalizing t must be strongly normalizing.
- If it is of the form $[(C_i : A_i)_{i\in I}]$ then we know that for all $i \in I$ we have $\mathcal{N}_0 \subseteq [\![A_i]\!] \subseteq \mathcal{N}$ by induction hypothesis. Let us take $t \in \mathcal{N}_0$ and show that $t \in [\![[(C_i : A_i)_{i\in I}]]\!]$. Let us now take $\Phi \in [\![\mathcal{F}]\!]$ and $(t_i \in ([\![A_i]\!] \to \Phi))_{i\in I}$ and show that $[t \mid (C_i \to t_i)_{i\in I}] \in \Phi$. It is enough to show that $[t \mid (C_i \to t_i)_{i\in I}] \in \mathcal{N}_0$ and since $t \in \mathcal{N}_0$ we only have to show $t_i \in \mathcal{N}$ for all $i \in I$, by definition of $\mathcal{N}_0$. It is the case since $([\![A_i]\!] \to \Phi) \subseteq (\mathcal{N}_0 \to \mathcal{N}) \subseteq \mathcal{N}$ by Lemmas 3 and 4. Let us now take $t \in [\![[(C_i : A_i)_{i\in I}]]\!]$ and show that $t \in \mathcal{N}$. Let us take take $\Phi \in [\![\mathcal{F}]\!]$ and $(t_i \in [\![A_i]\!] \to \Phi)_{i\in I}$. By definition $[t \mid (C_i \to t_i)_{i\in I}] \in \Phi \subseteq \mathcal{N}$, hence we must have $t \in \mathcal{N}$.
- If it is of the form $\forall X.A$ or $\exists X.A$ then for every $\Phi \in [\![\mathcal{F}]\!]$ we have $\mathcal{N}_0 \subseteq [\![A[X:=\Phi]]\!] \subseteq \mathcal{N}$ by induction hypothesis. As a consequence we obtain that $\mathcal{N}_0 \subseteq [\![\forall X.A]\!] \subseteq \mathcal{N}$ and that $\mathcal{N}_0 \subseteq [\![\exists X.A]\!] \subseteq \mathcal{N}$.
- If it is of the form $\mu_\kappa X.A$ or $\nu_\kappa X.A$ then the proof is similar to the previous case. Formally to manage ordinals, the proof is by induction on the size of the formula and the multiset of the semantics of all ordinals appearing in the formula. In all the previous cases, the size was decreasing and the multiset was constant. Here, the multiset decreases. □

Definition 16. *A substitution σ for terms, ordinals and type variables is said to be acceptable for some terms and types if it satisfies the following conditions.*

1. *After substitution, the resulting terms and types are closed parametric terms and types.*
2. *For every variable $\alpha_{<\kappa}$ (resp. $\alpha_{\leq\kappa}$) occurring in the terms or types, we have $[\![\alpha\sigma]\!] < [\![\kappa\sigma]\!]$ (resp.$[\![\alpha\sigma]\!] \leq [\![\kappa\sigma]\!]$).*

By extension, a substitution is said to be acceptable for a judgment if it is acceptable for the underlying terms and types.

Theorem 2. *Let A and B be types, t be a term and σ be an acceptable substitution for t, A and B.*

- *If $t : A$ is derivable then $[\![t\sigma]\!] \in [\![A\sigma]\!]$.*
- *If $t \in A \subset B$ is derivable and $[\![t\sigma]\!] \in [\![A\sigma]\!]$ then $[\![t\sigma]\!] \in [\![B\sigma]\!]$.*

Proof. We do a proof by induction on the derivations. Let us first note that if $\kappa < \tau$ (resp. $\kappa \leq \tau$) is derivable then $[\![\kappa\sigma]\!] < [\![\kappa\sigma]\!]$ (resp. $[\![\kappa\sigma]\!] \leq [\![\kappa\sigma]\!]$) follows from the definition of acceptable substitution, provided that σ is acceptable for κ and τ. Similarly, if $A \leq B$ is derivable and if σ is acceptable for A and B, then $[\![A\sigma]\!] \subset [\![B\sigma]\!]$. This follows from an immediate induction on the size of A, using the previous remark on ordinals for fixpoints.

We now reason by case analysis on the last typing or local subtyping rule used in the derivation of the considered typing or local subtyping judgment.

- ε Let σ be an acceptable substitution for t, A, B and C. By definition of $u = [\![\varepsilon_{x\in A\sigma}(t\sigma \notin B\sigma)]\!] \in [\![A\sigma]\!]$, and hence $u \in [\![C\sigma]\!]$ by induction hypothesis
- $\to_i$ Let σ be an acceptable substitution for $\lambda x.t \in C$. We need to show that $[\![(\lambda x t)\sigma]\!] \in [\![C\sigma]\!]$. We can assume that σ is acceptable with $A \to B$, because $[\![(\lambda x t)\sigma]\!]$ and $[\![C\sigma]\!]$ are independent of $\alpha\sigma$ for any ordinal α that occurs in $A \to B$ but not in C, nor in t. Therefore, by induction hypothesis, we have $[\![(\lambda x t)\sigma]\!] \in [\![(A \to B)\sigma]\!]$ implies $[\![(\lambda x t)\sigma]\!] \in [\![C\sigma]\!]$. It

remains to prove that $[\![(\lambda xt)\sigma]\!] \in [\![(A \to B)\sigma]\!]$. By induction hypothesis, we have $[\![t\sigma[x := \varepsilon_{x\in A\sigma}(t\sigma \notin B\sigma)]]\!] \in [\![B\sigma]\!]$ which by definition of the choice operator means that for all $u \in [\![A\sigma]\!]$ we have $[\![t\sigma[x := u]]\!] \in [\![B\sigma]\!]$.

$\forall_r$ Let σ be an acceptable substitution for $t \in A \subset \forall X.B$. We assume $[\![t\sigma]\!] \in [\![A\sigma]\!]$ and show $[\![t\sigma]\!] \in [\![(\forall X.B)\sigma]\!]$. Up to a renaming of X, we may assume that $(\forall X.B)\sigma = \forall X.B\sigma$. By induction hypothesis $[\![t\sigma]\!] \in [\![B\sigma[X := \varepsilon_X(t\sigma \notin B\sigma)]]\!]$ and hence $[\![t\sigma]\!] \in [\![A\sigma]\!]$. Consequently we $[\![t\sigma]\!] \in [\![B\sigma[X := \Phi]]\!]$ for all $\Phi \in \mathcal{M}$ by definition of the choice operator. We hence obtain $[\![t\sigma]\!] \in [\![(\forall X.B)\sigma]\!]$.

μ_l^* Let sigma be an acceptable substitution for $t \in \mu_\kappa X.A \subset B$. We assume $[\![t\sigma]\!] \in [\![(\mu_\kappa X.A)\sigma]\!]$ and $(\mu_\kappa X.A)\sigma = \mu_\kappa X.A\sigma$ (renaming X if necessary). By definition, this means that we can find $o < [\![\kappa\sigma]\!]$ such that $[\![t\sigma]\!] \in [\![A\sigma[X := \mu_o X.A\sigma]]\!]$. The substitution $\rho = \sigma[\alpha := o]$ is acceptable for t, B and $A[X := \mu_{\alpha<\kappa} X.A$. Furthermore we have $t\rho = t\sigma$, $A\rho = A\sigma$ and $B\rho = B\sigma$ because of the constraint on α. As a consequence, $[\![t\sigma]\!] \in [\![B\rho]\!] = [\![B\sigma]\!]$ by induction hypothesis.

I_m The intuition for the induction rules is given informally in the previous section. One point is missing to make the argument formal: the fact that the semantical truth of $A\sigma \subset B\sigma$ for an acceptable substitution σ indeed implies $[\![A\sigma]\!] \subset [\![B\sigma]\!]$. By definition, the truth of the judgment means that $u = [\![\varepsilon_{x\in A\sigma}(x \notin B\sigma)]\!] \in [\![A\sigma]\!]$ implies $u \in [\![B\sigma]\!]$. By definition of the semantics of the choice operator, this implication can only be true if $[\![A\sigma]\!] \subset [\![B\sigma]\!]$, otherwise we would obtain a contradiction with $u \in [\![A\sigma]\!]$ and $u \notin [\![A\sigma]\!]$.

The remaining cases can be proved in a similar way. □

The previous theorem is generally called the *adequacy lemma*. Intuitively, it establishes the compatibility between the syntax and the semantics. We will now rely on this theorem to obtain results such as strong normalization.

Theorem 3. *Let t be a pure term. If there is a type A with no free ordinal variables such that $t : A$ then t is strongly normalizable.*

Proof. We consider the substitution σ mapping all the free type variables of A to $\mathcal{N}$. It is obviously acceptable for t and A. Indeed, t cannot contain types or semantic ordinals as it is pure. Using the adequacy lemma (Theorem 2), we obtain that $[\![t\sigma]\!] \in [\![A\sigma]\!]$. We hence get $t \in [\![A\sigma]\!] \subseteq \mathcal{N}$ using Theorem 1. □

Note that the condition on free ordinal variables could be relaxed by a soundness condition forbidding cyclic constraints like $\alpha_{<\beta}$ and $\beta_{<\alpha}$. A direct consequence of strong normalization is that well-typed terms cannot produce runtime errors thanks to our definition of reduction.

Theorem 4. *We say that a data type is simple if it is closed and only contains sums, products and fixpoints. If we can derive $t : A$ for a simple data type A and a closed pure term t, then t reduces to a normal form u and $u : A$ is derivable.*

Proof. Using strong normalization (Theorem 3) we know that t reduces to a normal form u that remains closed (no free variables are introduced during reduction). Since A is a simple datatype, we can show by induction on A that $u \in [\![A]\!]$ implies $u : A$ if u is in normal form. □

Theorem 5 (logical consistency)**.** *There is no closed term t such that we can derive $t : \forall X.X$ or $t : [\,]$.*

Proof. This result follows from Theorem 2 using the fact that $[\![\forall X.X]\!] = [\![[\,]]\!] = \mathcal{N}_0$ only contains open terms. □

4. Type-checking Algorithm

Our system can be implemented by transforming the rule systems into recursive functions. This can be done easily because the system is mostly syntax-directed. For instance, there is only one typing rule for every form of term. Nonetheless, several subtle details need further discussion.

Unification variable. During type-checking, a term with an unknown type may arise (e.g. the argument of an application). To handle such situations, types are extended with a set of unification variables $\{U, V \dots\}$. Such variables can be used in the place of unknown types until their outer layer can be inferred. In our implementation [16], unification variables are handled as follows.

- If we encounter $U \leq U$ or $t \in U \subset U$ then we use reflexivity.
- If we encounter $U \leq V$ or $t \in U \subset V$, then we may decide that U and V are equal. For efficiency reasons, this can be implemented using a union-find data structure.
- If we encounter $U \leq A$, $A \leq U$, $t \in U \subset A$ or $t \in A \subset U$, then we may decide that U is equal to A provided that U does not occur in A. Note that it is essential to check occurrence inside choice operators. When U occurs only positively in A we can use $\mu X.A[U := X]$ as a definition for U.

Saturating induction. The induction rule is the only one that is not directed by the syntax. As a consequence, it cannot be implemented directly and requires special treatment. In our implementation, we try to apply the induction rule whenever possible (i.e. before every application of a local subtyping rule). As a consequence, we need to store induction hypotheses in a data structure. A branch of the proof can then be ended whenever a stored hypothesis applies. More precisely, we say that judgements are similar if they are equal up to the value of ordinals. For example, the judgment $\mu F \subset A$ and $\mu_{\alpha<\infty} F \subset A$ are similar. When a judgement that is encountered is similar to a stored induction hypothesis, we distinguish the following two cases.

- The ordinals are strictly smaller for the chosen ordering and this ends the branch.
- Typechecking fails to avoid infinite loops.

During a proof search with no polymorphic nor existential types, only finitely many distinct types up to similarity may occur. This is a consequence of the fact that no subtyping rule increases the size of a type except $\forall_l$ and $\exists_r$, if ordinals are ignored.

As a byproduct, this ensures termination of our semi-algorithm when checking a judgment containing no polymorphic nor existential types. Unfortunately, such an algorithm is not complete. However, this problem seems to be solvable using the size change principle [14] to detect valid induction instead of trying to build lexicographic combinations of ordinals compatible with use of the induction hypothesis. This is what is currently implemented in SubML, because it is both simpler and seems complete.

Avoiding the μ_r and ν_l rules. In an implementation, the μ_r and ν_l rules cannot be implemented without relying on unification on ordinals to search for suitable ordinals. This situation can in fact be avoided by enforcing the following invariant: all indexes κ of a positive μ_κ and negative ν_κ must be ∞. Following this idea, we can proceed as follows.

- We replace the μ_l rule and the ν_r rule by two new rules μ_l^∞ and ν_r^∞ dual to μ_r^∞ and ν_l^∞.
- We only consider ordinals appearing in a negative μ or a positive ν in induction rules. In fact, doing more seems useless.
- When we perform an imitation, when checking $t \in U \subset A$ (resp. $t \in A \subset U$), we can take $U = A'$ where A' is obtained

replacing by ∞ the ordinals in negative (resp. positive) μ or positive (resp. negative) ν in A. This imitation preserves the invariant and is correct because $A' \leq A$ (resp. $A \leq A'$).

5. Applications

Our new type system is quite expressive and allows for many different applications. First, it can be used for programming with inductive datatypes as usual. Several examples including unary natural numbers, lists and trees are provided with the implementation of SubML. In this section we focus on original applications.

Mixed inductive coinductive types. Our system is suitable for handling types containing both least and greatest fixpoints. As an example, let us consider the following two types S and L.

$$\mathrm{S} = \mu X \nu Y [\mathrm{A\ of\ } X | \mathrm{B\ of\ } Y] \qquad \mathrm{L} = \nu Y \mu X [\mathrm{A\ of\ } X | \mathrm{B\ of\ } Y]$$

The elements of the former can be thought of as streams of A'a and B's that only use finitely many A's. The elements of L are streams that do not use infinitely many consecutive A's. Our system is hence able to find a proof that $\mathrm{S} \subset \mathrm{L}$. It is displayed in Figure 4.

To check that the induction hypothesis H_1 is valid, we must prove that $\alpha_0 \leq \infty$, $\alpha_3 \leq \alpha_6$ and $(\alpha_0, \alpha_3) < (\alpha_5, \alpha_4)$. These inequalities are true because the constraints on ordinal variables are $\alpha_0 \leq \alpha_1 < \alpha_5 \leq \infty$ and $\alpha_3 \leq \alpha_4 \leq \alpha_6$. In this case, it is the first element of the tuple decreases. For H_2, we have to check that $\alpha_0 \leq \alpha_1$, $\alpha_2 \leq \alpha_4$ and $(\alpha_0, \alpha_2) < (\alpha_0, \alpha_3)$. This is again true because the constraints are $\alpha_0 \leq \alpha_1$ and $\alpha_2 < \alpha_3 \leq \alpha_4$. Here, it is the second element of the tuple that is decreasing. In this precise example, any choice of a lexicographic ordering would work. However, this is not always the case.

Mitchell's subtyping. In our system, the use of local subtyping together with choice operators allow many valid commutations of quantifiers with other connectives. In particular, it can derive Mitchell's containment axiom [19] and one of its variations.

$$\forall X.\mathrm{F}(X) \to \mathrm{G}(X) \subset (\forall X.\mathrm{F}(X)) \to \forall X.\mathrm{G}(X)$$
$$\forall X.\mathrm{F}(X) \to \mathrm{G}(X) \subset (\exists X.\mathrm{F}(X)) \to \exists X.\mathrm{G}(X)$$

The first derivation is given in figure 5. Note that the choice operators for terms and types are all well defined (i.e. they are not cyclic definitions). In the proof, simple imitations are enough to guess the type to use for the $\forall_l$ and $\exists_r$ rules.

Type annotations and dot notation. Our algorithm being incomplete, type annotations are required for the user to be able to help the system. However, type annotations are not completely natural in a Curry style language. Simple type coercions like $t : A$ can be added to the system without difficulty using the following rule.

$$\frac{t : A \qquad t \in A \subset B}{t : A : B}$$

However, type annotations are often required to reference bound type variables. Hence, a type abstraction constructor $\Lambda X.t$ is necessary, like in Church style languages. A very simple and natural way to extend the system with such annotations uses our choice operator in types. We define the following partial function relying on the name of bound variables.

$$W_\forall(t, X, \forall X.A) = \varepsilon_X(t \notin A)$$
$$W_\forall(t, X, \forall Y.A) = W_\forall(t, X, A[Y := \varepsilon_Y(t \notin A)]) \text{ when } X \neq Y$$

The name of the bound variables can be used because we never infer polymorphic types. Therefore, the names are always chosen by the programmer. We can hence use the following typing rule for type-checking with type abstractions.

$$\frac{t[X := W_\forall(t, X, A)] : A}{\Lambda X.t : A}$$

One of the advantages of this definition is that the order of the type abstraction is not relevant. In fact, only the type abstractions that are really needed have to be provided.

The same kind of definitions can be used to define dot notation on existential types. This time, two local subtyping rules need to be added to the system.

$$W_\exists(t, X, \exists X.A) = \varepsilon_X(t \in A)$$
$$W_\exists(t, X, \exists Y.A) = W_\exists(t, X, Y := \varepsilon_Y(t \notin A)]) \text{ when } X \neq Y$$
$$\frac{t : A \quad u \in W_\exists(t, X, A) \subset B}{u \in t.X \subset B} \qquad \frac{t : A \quad u \in B \subset W_\exists(t, X, A)}{u \in B \subset t.X}$$

As an example, we can define a type for categories using two abstract types O and M for objects and morphisms. We can then use dot notation to annotate the definition of a function dual computing the opposite category of a category.

$$\mathrm{Cat} = \exists O.\exists M. \left\{ \begin{array}{l} \mathrm{dom} : M \to O; \\ \mathrm{cod} : M \to O; \\ \mathrm{cmp} : M \to M \to M \end{array} \right\}$$

$$\mathrm{dual} : \mathrm{Cat} \to \mathrm{Cat}$$
$$= \lambda c. \left\{ \begin{array}{ll} \mathrm{dom} : c.M \to c.O & = c.\mathrm{cod}; \\ \mathrm{cod} : c.M \to c.O & = c.\mathrm{dom}; \\ \mathrm{cmp} : c.M \to c.M \to c.M & = \lambda xy.c.\mathrm{cmp}\, y\, x \end{array} \right\}$$

Typable recursor. We are now going to exhibit some short but enlightening examples showing the power of our system. The first examples are Church and Scott encodings of natural numbers. Although they have little practical interest, they provide difficult tests because they rely heavily on polymorphism and fixpoints. The type of Church natural numbers $\mathbb{N}_C$ and the type of Scott natural numbers $\mathbb{N}_S$ are defined below, together with their respective zero and successor functions.

$$\mathbb{N}_C = \forall X.(X \to X) \to X \to X$$
$$0_C = \lambda f x.x : \mathbb{N}_C$$
$$S_C = \lambda n f x.f\ (n\ f\ x) : \mathbb{N}_C \to \mathbb{N}_C$$
$$\mathbb{N}_S = \mu N \forall X.(N \to X) \to X \to X$$
$$0_S = \lambda f x.x : \mathbb{N}_S$$
$$S_S = \lambda n f x.f\ n : \mathbb{N}_S \to \mathbb{N}_S$$

Using Church encoding, our algorithm is able to typecheck the usual terms for predecessor P_C, recursor R_C, but also the less usual Maurey infimum ($\leq$), which requires inductive type [12]. It can be typechecked using few type annotations. They are defined as follows, where $\mathrm{T}, \mathrm{F} : \mathbb{B}$ denote booleans.

$$P_C = \lambda n.n\ (\lambda p x y.p\ (S_C\ x)\ x)\ (\lambda x y.y)\ 0_C\ 0_C$$
$$R_C = \lambda f a n.n\ (\lambda x p.f\ (x\ (S_C\ p))\ p)\ (\lambda p.a)\ 0_C$$
$$(\leq) = \lambda n m.n\ (\lambda f g.g\ f)\ (\lambda i.\mathrm{T})\ (m\ (\lambda f g.g\ f)\ (\lambda i.\mathrm{F}))$$

Scott numerals were introduced because they have a constant time predecessor, whereas Church numerals don't. Usually, programming using Scott numerals require the use of recursor in the style of Gödel's System T. Such a recursor can be programmed using a general fixpoint combinator, however this would imply the introduction of typable terms that are not strongly normalizing. In our type system, we can typecheck a strongly normalisable recursor found by Michel Parigot. It is displayed below together with several other terms and types that are used for Scott encoding.

$$\mathrm{pred} = \lambda n.n\ (\lambda p.p)\ 0_S : \mathbb{N}_S \to \mathbb{N}_S$$
$$\mathrm{U}(P) = \forall Y.Y \to \mathbb{N}_S \to P$$
$$\mathrm{T}(P) = \forall Y.(Y \to \mathrm{U}(P) \to Y \to \mathbb{N}_S \to P) \to Y \to \mathbb{N}_S \to P$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\nu Y[\mathrm{A\ of\ S}_{\alpha_{0\le\alpha_1}}|\mathrm{B\ of}\ Y]\subset\mu X[\mathrm{A\ of}\ X|\mathrm{B\ of\ L}_{\alpha_{2<\alpha_3}}]}\,H_2}{\nu Y[\mathrm{A\ of\ S}_{\alpha_{0\le\alpha_1}}|\mathrm{B\ of}\ Y]\subset \mathrm{L}_{\alpha_3\le\alpha_4}}\,\nu_r \qquad \dfrac{}{\mathrm{S}_{\alpha_{0\le\alpha_1}}\subset\mu X[\mathrm{A\ of}\ X|\mathrm{B\ of\ L}_{\alpha_3\le\alpha_4}]}\,H_1}{[\mathrm{A\ of\ S}_{\alpha_{0\le\alpha_1}}|\mathrm{B\ of}\ \nu Y[\mathrm{A\ of\ S}_{\alpha_{0\le\alpha_1}}|\mathrm{B\ of}\ Y]]\subset[\mathrm{A\ of}\ \mu X[\mathrm{A\ of}\ X|\mathrm{B\ of\ L}_{\alpha_3\le\alpha_4}]|\mathrm{B\ of\ L}_{\alpha_3\le\alpha_4}]}\,+}{[\mathrm{A\ of\ S}_{\alpha_{0\le\alpha_1}}|\mathrm{B\ of}\ \nu Y[\mathrm{A\ of\ S}_{\alpha_{0\le\alpha_1}}|\mathrm{B\ of}\ Y]]\subset\mu X[\mathrm{A\ of}\ X|\mathrm{B\ of\ L}_{\alpha_3\le\alpha_4}]}\,\mu_r^\infty}{\nu Y[\mathrm{A\ of\ S}_{\alpha_{0\le\alpha_1}}|\mathrm{B\ of}\ Y]\subset\mu X[\mathrm{A\ of}\ X|\mathrm{B\ of\ L}_{\alpha_3\le\alpha_4}]}\,\nu_l^\infty}{\nu Y[\mathrm{A\ of\ S}_{\alpha_{1<\alpha_5}}|\mathrm{B\ of}\ Y]\subset\mu X[\mathrm{A\ of}\ X|\mathrm{B\ of\ L}_{\alpha_4\le\alpha_6}]}\,I_2}{\mathrm{S}_{\alpha_5}\subset\mu X[\mathrm{A\ of}\ X|\mathrm{B\ of\ L}_{\alpha_4\le\alpha_6}]}\,\mu_l}{\mathrm{S}\subset\mu X[\mathrm{A\ of}\ X|\mathrm{B\ of\ L}_{\alpha_6<\infty}]}\,I_1}{\mathrm{S}\subset\mathrm{L}}\,\nu_r$$

where $S_\alpha = \mu_\alpha X \nu Y[A \text{ of } X|B \text{ of } Y]$ and $L_\alpha = \nu_\alpha Y \mu X[A \text{ of } X|B \text{ of } Y]$

Figure 4. Proof of $\mathrm{S} = \mu X \nu Y[\mathrm{A\ of}\ X|\mathrm{B\ of}\ Y] \subset \nu Y \mu X[\mathrm{A\ of}\ X|\mathrm{B\ of}\ Y] = \mathrm{L}$.

$$\dfrac{\dfrac{\dfrac{}{x_0\in\mathrm{F}(X_0)\subset\mathrm{F}(X_0)}=}{x_0\in\forall X.\mathrm{F}(X)\subset\mathrm{F}(X_0)}\,\forall_l \quad \dfrac{\dfrac{}{x_1\,x_0\in\mathrm{G}(X_0)\subset\mathrm{G}(X_0)}=}{x_1\,x_0\in\mathrm{G}(X_0)\subset\forall X.\mathrm{G}(X)}\,\forall_r}{x_1\in\mathrm{F}(X_0)\to\mathrm{G}(X_0)\subset(\forall X.\mathrm{F}(X))\to\forall X.\mathrm{G}(X)}\,\to}{x_1\in\forall X.\mathrm{F}(X)\to\mathrm{G}(X)\subset(\forall X.\mathrm{F}(X))\to\forall X.\mathrm{G}(X)}\,\forall_l$$

$$x_1 = \varepsilon_{x\in\forall X.\mathrm{F}(X)\to\mathrm{G}(X)}(x\notin(\forall X.\mathrm{F}(X))\to\forall X.\mathrm{G}(X))$$
$$x_0 = \varepsilon_{x\in\forall X.\mathrm{F}(X)}(x_1\,x\notin\forall X.\mathrm{G}(X))$$
$$X_0 = \varepsilon_X(x_1\,x_0\notin\mathrm{G}(X))$$

Figure 5. Derivation of Mitchell's containment axiom.

$$\begin{aligned}
\mathbb{N}' &= \forall P.\mathrm{T}(P)\to\mathrm{U}(P)\to\mathrm{T}(P)\to\mathbb{N}_S\to P\\
\zeta &= \lambda arq.a : \forall P.P\to\mathrm{U}(P)\\
\delta &= \lambda afprq.f\,(\mathrm{pred}\,q)\,(p\,r\,(\zeta\,a)\,r\,q)\\
&\quad : \forall P.P\to(\mathbb{N}_S\to P\to P)\to\mathrm{T}(P)\\
R_S &= \lambda afn.n{:}\mathbb{N}'\,(\delta\,a\,f)\,(\zeta\,a)\,(\delta\,a\,f)\,n\\
&\quad : \forall P.P\to(\mathbb{N}_S\to P\to P)\to\mathbb{N}_S\to P
\end{aligned}$$

It is easy to check that the term R_S is indeed a recursor for Scott numerals. As this term is typable, Theorem 3 implies that it is strongly normalizable. It can be typechecked because the subtyping relation $\mathbb{N}_S \subset \mathbb{N}'$ holds in our system. It is however not clear what are the terms of type $\mathbb{N}'$. The minimum type annotation for our algorithm to type-check the recursor are its type and the annotation $n : \mathbb{N}'$. It is not necessary to give the type of the subterms ζ nor δ.

The recursor on Scott numerals can be adapted to other algebraic data types like lists or trees. Surprisingly, it can also be adapted to some coinductive data types. For instance, it is possible to encode streams using the following definitions.

$$\begin{aligned}
\mathbb{S}(A) &= \nu K\exists S.\{\mathrm{hd} : S\to A;\mathrm{tl} : S\to K;\mathrm{st} : S\}\\
\mathrm{hd} &= \lambda s.s.\mathrm{hd}\,s.\mathrm{st}\\
\mathrm{tl} &= \lambda s.s.\mathrm{tl}\,s.\mathrm{st}\\
\mathrm{cons} &= \lambda al.\{\mathrm{hd}=\lambda x.a;\mathrm{tl}=\lambda x.l;\mathrm{st}=()\}
\end{aligned}$$

In this definition of streams, the existentially quantified type is used as an internal state of the stream. It needs to be provided in order to compute an element of the steam. The order of the fixpoint and the existential is essential to allow the typing of cons. The use of an internal state is essential to keep strong normalization and add some laziness to the data-type. In particular, a function call is required to access a new element of the stream. The following definitions are required to program a strongly normalizing co-iterator.

$$\begin{aligned}
\mathbb{S}_0(A,S) &= \nu K\{\mathrm{hd} : S\to A;\mathrm{tl} : S\to K;\mathrm{st} : S\}\\
\mathrm{T}(A,P) &= \forall Y.P{\times}Y\to\{\mathrm{hd} : P{\times}Y\to A;\mathrm{tl} : Y;\mathrm{st} : P{\times}Y\}\\
\mathbb{S}'(A,P) &= \left\{\begin{array}{ll}\mathrm{hd} & : P{\times}\mathrm{T}(A,P)\to A;\\ \mathrm{tl} & : \mathrm{T}(A,P);\\ \mathrm{st} & : P{\times}\mathrm{T}(A,P)\end{array}\right\}\\
\zeta &: \forall A.\forall P.(P\to A)\to\forall X.P{\times}X\to A\\
&= \lambda fs.f\,s.1\\
\delta &: \forall A.\forall P.(P\to A)\to(P\to P)\to\mathrm{T}(A,P)\\
&= \lambda fns.\{\mathrm{hd}=\zeta\,f;\mathrm{tl}=s.2;\mathrm{st}=(n\,s.1,s.2)\}\\
\mathrm{coiter} &: \forall A.\forall P.P\to(P\to A)\to(P\to P)\to\mathbb{S}(A)\\
&= \Lambda AP.\lambda sfn.\left\{\begin{array}{ll}\mathrm{hd} & =\zeta\,f;\\ \mathrm{tl} & =\delta\,f\,n;\\ \mathrm{st} & =(s,\delta\,f\,n)\end{array}\right\} : \mathbb{S}_0(A,P{\times}\mathrm{T}(A,P))
\end{aligned}$$

In these definitions, we use the same names as in the previous one to show the similarities. One of the main difference in that we use here the native records of our language. For Scott's numeral, it is possible to use native sums, but we have to keep some function types to be able to program a strongly normalisable recursor. For the types $\mu N[\mathrm{Z}|\mathrm{S\ of}\ N]$, we are not able program such a recursor and we conjecture that it is impossible.

The minimum type annotation for this program is to use the following subtyping relations:

$$\mathbb{S}'(\mathrm{A},\mathrm{P})\subset\mathbb{S}_0(\mathrm{A},\mathrm{P}{\times}\mathrm{T}(\mathrm{A},\mathrm{P}))\subset\mathbb{S}(\mathrm{A})$$

Here we gave the first subtypings by giving the type of ζ and δ. The two subtypings are necessary, because our simple strategy for unification variables can not simultaneously guess the type of the global state (i.e. $\mathrm{P}{\times}\mathrm{T}(\mathrm{A},\mathrm{P})$) and the instanciation of Y in $\mathrm{T}(\mathrm{A},\mathrm{P})$. The later is given by the second subtyping, which also reverse the order of the ν and $\exists$ quantifier. The first subtyping gives the subtype $\mathbb{S}'(\mathrm{A},\mathrm{P})$ of streams which does not use fixpoint and that really make the typing works.

6. Perspectives and Future Work

In this paper, we introduced a new subtyping-based framework for the design of programming languages based on system F. Our experiments show that such systems are practical and can be implemented easily. However, a lot of work remains to explore combinations of our system with several common programming features.

Higher-order and ordinal quantification. In our system, only types can be quantified over. It is however possible to allow quantification over ordinals. Such a feature is particularly useful as it allows the definition of size-preserving functions. As an example, the map function on lists can be given the type

$$\forall\alpha.(A \to B) \to \mu_\alpha X.List_X(A) \to \mu_\alpha List_X(B)$$

where $List_X(Y)$ denotes the appropriate parametric type. As a consequence, the length of the output list is guaranteed to be the same as the length of the input list. Such an information can be used for proving termination. Ordinal quantification has been implemented in a modified version of SubML [17].

In practice, many other forms of quantification are useful. For instance, quantification over values is necessary for allowing proofs of programs as in [15]. The most natural way of allowing many different kinds of quantification is to switch to higher-order types. We then use one atomic sort for every kind of objects (e.g. formulas, ordinals, values). The main difficulty for extending our system to higher order is purely practical. The handling of unification variables needs to be generalized into a form of higher-order pattern matching and variances have to be computed to ensure the validity of (co)inductive types.

Dependent types and proofs of programs. One of our motivations for this work is the integration of subtyping to the realizability models defined in a previous work of the first author [15]. To achieve this goal, the system needs to be extended with a first-order layer having terms as individuals. Two new type constructors $t \in A$ and $A \restriction t \equiv u$ are then required to encode dependent products and specifications. These two ingredients would be a first step toward program proving in our system.

Extensible sums and products. The proposed system is relatively expressive, however it lacks flexibility for records and pattern-matching. A form of inheritance allowing extensible records and sums is desirable. Moreover, features like record opening are required to recover the full power of ML modules and functors.

Completeness and size change principle. As mentioned in the introduction, our handling of induction and coinduction is not complete for the system without polymorphism and existential types. For example, the system is unable to derive the correct inclusions $\mathrm{T} \subset \mathrm{U}$ and $\mathrm{T} \subset \mathrm{V}$ if we define the following types.

$$\mathrm{T} = \mu X \nu Z \mathrm{F}(X, X, Z)$$
$$\mathrm{U} = \mu X \nu Z \mu Y \mathrm{F}(X, Y, Z)$$
$$\mathrm{V} = \mu Y \nu Z \mu X \mathrm{F}(X, Y, Z)$$

Here, our algorithm fails because the required constraints are not met. For our algorithm to succeed, a deeper proof search would be required, unfolding the fixpoints more than once.

However, using the size change principle [14] to check the well-foundedness of subtyping proofs seems to give a complete subtyping algorithm in the monomorphic fragment. Moreover, the size of the produced proof trees is reasonably small. A first implementation shows that all examples with up to five alternations of inductive and coinductive types are correctly checked. We hope to prove the completeness of this method in the near future.

References

[1] M. Abadi, L. Cardelli, and G. Plotkin. Types for the Scott numerals. URL `http://lucacardelli.name/papers/notes/scott2.ps`. 1993.

[2] A. Abel and B. Pientka. Wellfounded recursion with copatterns: a unified approach to termination and productivity. In G. Morrisett and T. Uustalu, editors, *ICFP Proceedings*, pages 185–196. ACM, 2013.

[3] R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15, 1993.

[4] L. Cardelli, S. Martini, J. C. Mitchell, and A. Scedrov. An extension of system F with subtyping. In T. Ito and A. R. Meyer, editors, *TACS Proceedings*, volume 526 of *LNCS*, pages 750–770, 1991.

[5] J. Courant. $\mathcal{MC}_2$ a module calculus for pure type systems. *Journal of Functional Programming*, 17:287–352, 2007.

[6] S. Dolan and A. Mycroft. Polymorphism, subtyping and type inference in MLsub. URL `http://www.cl.cam.ac.uk/~sd601/mlsub/`. 2015.

[7] J. Garrigue. Programming with polymorphic variants. In *ML Workshop*, 1998.

[8] J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris 7, 1972.

[9] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*. Cambridge University Press, 1989.

[10] D. Hilbert and P. Bernays. *Grundlagen der Mathematik*, volume 1 of *Grundlehren der mathematischen Wissenschaften*. 1968.

[11] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In H. Boehm and G. L. Steele, Jr., editors, *POPL Proceedings*. ACM, 1996.

[12] J.-L. Krivine. Un algorithme non typable dans le système F. *CRAS*, 304, 1987.

[13] J.-L. Krivine. *Lambda-calcul : types et modèles*. Études et recherches en informatique. Masson, 1990.

[14] C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *POPL Proceedings*, pages 81–92. ACM, 2001.

[15] R. Lepigre. A classical realizability model for a semantical value restriction. In *ESOP Proceedings (Accepted for publication)*, 2016.

[16] R. Lepigre and C. Raffalli. SubML implementation, 2015. URL `http://lama.univ-savoie.fr/subml`.

[17] R. Lepigre and C. Raffalli. SupML: extension of the implementation of subML, 2015. URL `http://lama.univ-savoie.fr/supml`.

[18] D. MacQueen. Modules for Standard ML. In *Proceedings of LFP 1984*, pages 198–207. ACM, 1984.

[19] J. C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76(2):211–249, 1988.

[20] K. Nour and K. Saber. A semantical proof of the strong normalization theorem for full propositional classical natural deduction. *Archive for Mathematical Logic*, 45, 2006.

[21] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[22] F. Pottier. *Synthèse de types en présence de sous-typage: de la théorie à la pratique*. PhD thesis, Université Paris 7, July 1998.

[23] C. Raffalli. The PhoX proof assistant, 2008. URL `http://www.lama.univ-savoie.fr/~RAFFALLI/phox.html`.

[24] C. Raffalli. Type checking in system F^η. In *Prépublication 98-05a du LAMA*, 1998.

[25] J. C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 408–423. Springer-Verlag, 1974.

[26] C. Riba. On the values of reducibility candidates. In P. Curien, editor, *TLCA Proceedings*, LNCS. Springer, 2009.

[27] C. Riba. Toward a general rewriting-based framework for reducibility. URL `https://perso.ens-lyon.fr/colin.riba/`. 2008.

[28] J. L. Sacchini. Well-founded sized types in the calculus of (co)inductive constructions. URL `http://www.qatar.cmu.edu/~sacchini/well-founded/`. 2015.

[29] J. Tiuryn and P. Urzyczyn. The subtyping problem for second-order types is undecidable. *Inf. Comput.*, 179(1):1–18, 2002.