PRACTICAL SUBTYPING FOR SYSTEM F WITH SIZED (CO-)INDUCTION

RODOLPHE LEPIGRE AND CHRISTOPHE RAFFALLI

LAMA, UMR 5127 CNRS - Université Savoie Mont Blanc
e-mail address: rodolphe.lepigre@univ-smb.fr

LAMA, UMR 5127 CNRS - Université Savoie Mont Blanc
e-mail address: christophe.raffalli@univ-smb.fr

Abstract. We present a rich type system with subtyping for an extension of System F. Our type constructors include sum and product types, universal and existential quantifiers, inductive and coinductive types. The latter two size annotations allowing the preservation of size invariants. For example it is possible to derive the termination of the quicksort by showing that partitioning a list does not increase its size. The system deals with complex programs involving mixed induction and coinduction, or even mixed (co-)induction and polymorphism (as for Scott-encoded datatypes). One of the key ideas is to completely separate the induction on sizes from the notion of recursive programs. We use the size change principle to check that the proof is well-founded, not that the program terminates. Termination is obtained by a strong normalization proof. Another key idea is the use symbolic witnesses to handle quantifiers of all sorts. To demonstrate the practicality of our system, we provide an implementation that accepts all the examples discussed in the paper and much more.

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 [10], modules [22]). 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, natural numbers may be defined as a subtype of integers.

The downside of subtyping is that it is difficult to incorporate in complex systems like Haskell or OCaml. For example, OCaml provides polymorphic variants [10] for which complex annotated types are inferred. For instance, one would expect the function

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

1998 ACM Subject Classification: D.3.1, Programming languages, Formal Definitions and Theory.
Key words and phrases: subtyping, (co-)induction, Hilbert's epsilon, size-change principle.

to be given type `[`T|`F]` $\rightarrow$ `[`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]` $\rightarrow$ `[>`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 using standard unification techniques and following its typing and subtyping rules (Figures 10 and 5). In particular, the typing and subtyping procedures need not backtrack since they are directed by the syntax of terms and types respectively. Our motivation being the design of a practical programming language, we consider a rather 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 intersections and unions 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 several technical innovations that are 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 to be interpreted as "if t has type A then it also has type B". Usual subtyping is then recovered using choice operators inspired from Hilbert's Epsilon and Tau functions. In our system, the choice operator $\varepsilon_{x\in A}(t \notin B)$ 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] The usual subtyping relation $A \subset B$ is then recovered using $\varepsilon_{x\in A}(x \notin B) \in A \subset B$. Indeed, the term $\varepsilon_{x\in A}(x \notin B)$ denotes a counterexample to $A \subset B$ if it exists. Therefore, if we can prove that $\varepsilon_{x\in A}(x \notin B)$ has type B provided that it has type A, then such a counterexample cannot exist. This exactly means that A is a subtype of B.

Choice operators can be used to replace free variables, and hence suppress the need for contexts in the usual sense.[2] Intuitively, $\varepsilon_{x\in A}(t \notin B)$ can be seen as a counterexample to the fact that $\lambda x\, t$ has type $A \rightarrow B$. Consequently, we can use the rule

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

for typing λ-abstractions. It can be read as a proof by contradiction as its premise is only valid when there is no term u in A such that $t[x := u]$ does not have type B. The axiom rule is then replaced by the following typing rule for choice operators.

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

[1] As our model is based on Girard's reducibility candidates [11, 12], the semantics of a type is never empty.

[2] We will still use contexts to store ordinals that are assumed to be positive (see Section 4).

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

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

Note that this rule does not carry a (usually required) freshness constraint on variable X. It is not required thanks to the use of choice operators.

In conjunction with local subtyping, choice operators allow the derivation of valid permutations of quantifiers and connectors. For instance, Mitchell's containment axiom [7]

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

can be easily obtained. Consequently, our type system mostly contains typing and subtyping rules that are syntax-directed. In particular, we do not have a transitivity rule for local subtyping, and we do not yet know whether such a rule is admissible in our system. In practice, transitivity is seldom required[3] and type annotations like $((t : A) : B) : C$ can be used to guide the system whenever necessary. Such annotations enforce the decomposition of a proof of $t : C$ into proofs of $t : A$, $t : A \subset B$ and $t : B \subset C$, which may help the system to guess correct instantiation of unification variables.

Well-founded ordinal induction and size change principle. 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. In this paper, such fixpoints are decorated with an ordinal κ to form sized types $\mu_\kappa X.F(X)$ and $\nu_\kappa X.F(X)$ [15, 2, 31]. Intuitively, these types correspond to κ iterations of F on the types $\bot$ and $\top$ respectively. In particular, $t : \mu_\kappa X.F(X)$ is equivalent to $t : F(\mu_\tau X.F(X))$ for some ordinal $\tau < \kappa$ and $t : \nu_\kappa X.F(X)$ is equivalent to $t : F(\nu_\tau X.F(X))$ for all ordinals $\tau < \kappa$. Moreover, there is an ordinal ∞ such that for every covariant parametric type F, $\mu_\infty X.F(X)$ and $\nu_\infty X.F(X)$ are respectively the least and greatest fixpoint of F.

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, but accepts proofs that are not well-founded. To solve this problem, we rely on the size change principle [19], which allows us to check for well-foundedness a posteriori. Our system is thus suitable for handling subtyping between mixed inductive and coinductive types. For instance, it 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 is in some sense complete. However, we failed to prove completeness in the presence of function types, the main problem being the mere definition of what it means to be complete in this setting.

[3]This tends to indicate that the transitivity of local subtyping is admissible.

In this paper, the language that is used to refer to ordinals in the syntax is very limited. It only contains ∞, a successor symbol and some choice operators, as for instance in [30]. In particular, 0 does not need to be represented. It is clear that the system could be improved by enlarging the language of ordinals with functions such as the maximum or the addition.

Totality of recursive functions. As for local subtyping judgments, it is possible to use circular proofs for typing recursive programs. General recursion is enabled by adding a fixpoint combinator $Yx.t = t[x := Yx.t]$ with the following typing rule.

$$\frac{\vdash t[x := Yx.t] : A}{\vdash Yx.t : A}$$

It is easy to see that, using this rule, a proof of $Yx.t : A$ will require a proof of $Yx.t : A$, which induces circularity. As there is no guarantee that such circular proofs are well-founded, we need to rely on the size-change principle again. Given its simplicity, the obtained system is surprisingly powerful. However, it is sometimes necessary to unfold a fixpoint several times to build a well-founded proof (see Section 8). This process can be implemented in the implementation.

Quantification over ordinals. As types can be annotated with ordinal sizes, it is natural to allow quantification over the ordinals themselves. This enables us to write type such as $\forall\alpha.\mu_\alpha X.F(X) \to \mu_\alpha X.F(X)$ to express the fact that a function returns a result which size is not greater than the size of its input. This extension can be obtained with very little effort as both the theory and the implementation already contain everything that is required to deal with sized types. However, ordinal quantification allows the system to infer the termination of more complex algorithm such as quicksort, for which it is required to show that partitioning a list does not increase its size.

Properties of the system. A first version of the language without general recursion (i.e. without the fixpoint combinator) is defined in Section 4. It has three main properties: strong normalisation, type safety and logical consistency (Theorems 6.23, 6.25 and 6.26). These properties are preserved after the introduction of the fixpoint combinator in Section 7 with only a small change. Indeed, strong normalisation is compromised by the reduction rule for the fixpoint ($Yx.t \succ t[x := Yx.t]$), which is obviously non-terminating. Nevertheless, we can still prove strong normalisation if we restrict ourselves to weak reduction strategies (i.e. those that do not reduce under λ-abstractions). These results follow from the construction of a realizability model presented in Section 6. They are consequences of the adequacy lemma (Theorem 6.22), which establishes the compatibility of the model with the language and type system.

Implementation. Typing and subtyping are likely to be undecidable in our system. Indeed, it contains Mitchell's variant of System F [7], for which both typing and subtyping are undecidable [32, 33, 34]. Moreover, we believe that there are no practical, complete semi-algorithms for extensions of System F like ours. 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 relevant error message. The typing rules being syntax-directed, type-checking can only diverge when checking a local subtyping judgment. In this case, an error message can be built using the last applied typing rule.

As a proof of concept, we implemented a toy programming language based on our system. It is called SubML and is available online [21]. Aside from a few subtleties described in Section 9, the implementation is straightforward and remains very close to the typing and subtyping rules of Figures 10 and 5. Although the system is relatively expressive, its simplicity allows for a very concise implementation. The main functions (type-checking and subtyping) require around 400 lines of OCaml code. The full implementation, including parsing, evaluation and LaTeX pretty printing is less that 6000 lines long.

We conjecture that our implementation is complete (i.e. may succeed on all typable programs), provided that enough type annotations are given. On practical instances, the required amount of annotations correspond to what a human would need to understand the program (see Section 8). Overall, our system provides a similar user experience to statically typed functional language like OCaml or Haskell. In such languages, type annotations are also required for advanced features like polymorphic recursion.

SubML provides literate programming features (inspired by PhoX [28]) that can be used to generate LaTeX documents. In particular, the examples presented in Section 8 (including proof-trees) have been generated using SubML and are therefore machine checked. Moreover, many examples of programs (around 4000 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 and 8). 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 [24]. 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.

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

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 data types, the product that we use in the definition of streams does not need to be encoded using polymorphism. As a consequence, this definition of streams may have practical interest.

As the system is incomplete, the user must be able to give type annotation. But with polymorphism and Curry style, we have no way to introduce the name of type variables in a term. In fact, because of our use of choice operators, there are no type variable. We found a solution using a syntax allowing to give names to local types by pattern matching: 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

$$\text{Id} : \forall X.X \to X = \lambda x.\ \text{let}\ X\ \text{such that}\ x : X\ \text{in}\ (x : X)$$

An 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.

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

For any term h : 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 [8].

Related work. The language presented in this paper is an extension of John Mitchell's System F_η [7], which itself extends Jean-Yves Girard and John Reynolds's System F [11, 29] with subtyping. Our subtyping algorithm extends previous work [25, 4] to support mixed induction and coinduction with polymorphic and existential types. In particular, we improve on an unpublished work of the second author [27].

Our system uses sized types [15] since we annotate inductive and coinductive types with ordinals. Such a technique is now wide spread for handling induction [5, 6, 17, 13, 31] and even coinduction [3, 2, 30].

A first difference with our work is the use of Curry-style, but this is probably not a very important difference for the termination itself, but as more consequence in the meaning of subtyping (see below).

More importantly, previous solutions uses rule which specifically check relation between the size of the ordinal/size parameters, when doing (recursive) function calls. Termination is ensured in all previous work known to the authors by rule dealing with induction and co-induction.

Our system completely avoid this. All relation between sizes are in the subtyping rule for inductive and co-inductive rule. And the termination check using the size change principle is used to prove that the proof is well-founded. This allows for inductive proof of subtyping relation between mixed inductive and co-inductive types which seems completely novel in this work.

Antoher consequence is that we use a trivial fixpoint rule that simply unfold the fixpoint. This means that the type constructors μ and ν, the termination and the fixpoint rule are treated by rules and concept that are in some sence orthogonal.

Here is an informal account of some other ingredients we think to be novel in our treatment of sized types:

– We have ordinal contexts to assume the positiveness of ordinals.
– The subtyping rule to prove $\mu_\kappa F \subset A$ and $A \subset \nu_\kappa F$ use a choice operator for ordinal $\varepsilon_{\alpha<\kappa}(...)$ and the positivity of κ is assumed in the premise.
– This positivity context allows to prove that $\varepsilon_{\alpha<\kappa}(...) < \kappa$ provided that κ is in the context. Apart from the construction of the call-graph for the size change principle, only the subtyping rules for $A \subset \mu_\kappa F$ and $\nu_\kappa F \subset A$ require to compare ordinals.
– To obtain termination in presence of fixpoint and treat enough programs, we introduce two new connectives $A \wedge \gamma$ and $A \vee \gamma$. They allow to transport some positiveness hypotheses from subtyping to typing. In the type $A \vee \gamma$, γ is a set of ordinals and when they are not all positive, $A \vee \gamma$ is the top type of our model (the set of terminating program). Otherwise $A \vee \gamma$ has the same semantics as A. The type $A \wedge \gamma$ is similar.

  For instance, the implication introduction rule is:

$$\frac{\gamma \vdash \lambda x\ t \in (A \to B) \vee \gamma_0 \subset C \qquad \gamma, \gamma_0 \vdash t[x := \epsilon_{x \in A}(t \notin B)] : B}{\gamma \vdash \lambda x\ t : C} \to_i$$

  The formula $(A \to B) \vee \gamma_0$ allows to assume the positiveness of some ordinals from the subtyping proof, via the rule for $\vee$. This rule is correct because if some ordinal in γ_0 is null, then $\lambda x.t$ is indeed in $(A \to B) \vee \gamma_0$ because it is terminating (recall that we do not reduce under abstraction when we consider recursive programs) and therefore in C by the subtyping premise.

In some sense, we have a more logical account of the relation between ordinals which leads to a system that we find considerably simpler than previous systems. Moreover, no syntactic condition arise anywhere in our system (like the semi-continuous restriction in [2]). In our system, this is completely implicit. However, we checked that some known examples that are rejected by such a test because they are incorrect are still rejected in our system.

We also chose to trade decidability for simplicity and therefore not to prove any decidability result. We are happy to work with a semi-algorithm and the user experience is not different from working with implicit variables in languages like Coq or Agda. Nevertheless, this is not completely satisfactory, and a it would still be better to have a semi-algorithm which is a complete algorithm for the fragment without polymorphism. We think that a complete algorithm solving constraint on sizes is feasible. There is an algorithm for a language with only successor in [16] that we could probably adapt.

Subtyping has been extensively studied in the context of ML-like languages, starting with the work of Roberto Amadio and Luca Cardelli [4]. Recent work includes the MLsub system [9], 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 [26].

However, we are not aware on any work on subtyping that can handle all its aspects when we are in Curry-style system F: permutation of quantifiers and connectives, inductive and coinductive types. For instance, as far as we know, no system available are able to prove the subtyping related to alternation of inductive the simplest non trivial one being (see Section 5):

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

1. Syntactic ordinals

In this section, we introduce a syntax for representing ordinals. It will be used to equip the types of our language with a notion of size, as is usually done for sized types [15]. Here, ordinals will also be used to show that infinite typing derivations are well-founded.

Convention 1.1. We will use the vector notation $\vec{e}$ to denote a finite sequence $(e_1, \ldots, e_n)$ of elements (e.g. syntactic ordinals). The size of such a sequence will be denoted $|\vec{e}| = n$. Note that we will sometimes have implicit size constraints on vectors (e.g. when writing substitutions like $\gamma[\vec{\alpha} := \vec{\kappa}]$).

Definition 1.2. The sets of *syntactic ordinals* $\mathcal{O}$ is defined from a set of ordinal variables $\mathcal{V}_{\mathcal{O}} = \{\alpha, \beta, \ldots\}$ and a predicate language $\mathcal{P} = \{P, J, K \ldots\}$ by the following BNF grammar.

$$\begin{aligned} \kappa, \tau, \upsilon &::= \alpha \mid \infty \mid \tau + 1 \mid \varepsilon^i_{\vec{\alpha} < \vec{\xi}} P(\vec{\alpha}, \vec{\kappa}) \\ \xi &::= \kappa \mid \infty{+}\omega \end{aligned}$$

In syntactic ordinals of the form $\varepsilon^i_{\vec{\alpha} < \vec{\xi}} P(\vec{\alpha}, \vec{\kappa})$, the variables in the vector $\vec{\alpha}$ are bound in $P(\vec{\alpha}, \vec{\kappa})$, but not in $\vec{\xi}$.

Moreover, we syntactically enforce $1 \leq i \leq |\vec{\alpha}| = |\vec{\xi}|$ and $|P| = |\vec{\alpha}| + |\vec{\kappa}|$, where $|P|$ denotes the arity of the predicate P.

The language of syntactic ordinals contains the constant ∞, a successor symbol $\tau + 1$ and *ordinal choice operators* of the form $\varepsilon^i_{\vec{\alpha} < \vec{\xi}} P(\vec{\alpha}, \vec{\kappa})$ (also referred to as *ordinal witnesses*). The vector $\vec{\tau}$ defined as $\tau_i = \varepsilon^i_{\vec{\alpha} < \vec{\xi}} P(\vec{\alpha}, \vec{\kappa})$ denotes ordinals point-wise smaller than $\vec{\xi}$ and such that "$P(\vec{\tau}, \vec{\kappa})$ is true" (this will be made formal in Definition 1.6). In the upper bound $\vec{\xi}$, one can use a special ordinal $\infty{+}\omega$ which is not a syntactic ordinals. In fact, $\alpha < \xi$ means that there are no upper bound. Indeed, in the semantics, all ordinals will be interpreted as element of a large enough ordinal $[\![\mathcal{O}]\!]$ which will be the interpretation of $\infty{+}\omega$.

Convention 1.3. For convenience, we will use the following notations.

– We will use the abbreviation $\kappa + n$ with $n \in \mathbb{N}$ for $\kappa + 1 + \cdots + 1$ (n times).
– We will write $\varepsilon_{\alpha < \xi} P(\alpha, \vec{\kappa})$ for $\varepsilon^1_{\vec{\alpha} < \vec{\xi}} P(\vec{\alpha}, \vec{\kappa})$ when $|\vec{\alpha}| = |\vec{\kappa}| = 1$.
– We will also write $\vec{\varepsilon}_{\vec{\alpha} < \vec{\xi}} P(\vec{\alpha}, \vec{\kappa})$ for the vector which i-th component is $\varepsilon^i_{\vec{\alpha} < \vec{\xi}} P(\vec{\alpha}, \vec{\kappa})$.
– We will denote 0 the zero ordinal and $\vec{0}$ a vector of zero ordinals. These denote (vector of) actual ordinals, not syntactic ordinals.

In our semantics, the symbol ∞ represents the ordinal 2^{2^ω}. This value is required to ensure the convergence of all fixpoints corresponding to inductive and coinductive types. Using the successor in the syntax, we may reach ordinals such as $2^{2^\omega} + k$ for any finite ordinal k, hence the following definition.

Definition 1.4. We denote $[\![\mathcal{O}]\!]$ the ordinal $2^{2^\omega} + \omega$ (where ω is the cardinal of the natural numbers).

To handle free variables in syntactic ordinals, we will extend their syntax using actual ordinals. In other words, we will include elements of the semantics into the syntax. This will allow us to substitute the free ordinal variables using ordinals directly, and thus allow us to work only on closed (parametric) syntactic ordinals.

Definition 1.5. The set of *parametric syntactic ordinals* $\mathcal{O}^*$ is obtained by extending the language of syntactic ordinals with actual ordinals $o \in [\![\mathcal{O}]\!]$.

$$\kappa, \tau, \upsilon \ ::= \alpha \ \mid \ \infty \ \mid \ \tau + 1 \ \mid \ o \ \mid \ \varepsilon^i_{\vec{\alpha}<\vec{\xi}} P(\vec{\alpha}, \vec{\kappa})$$
$$\xi \ ::= \kappa \ \mid \ \infty{+}\omega$$

We will denote $\kappa[\alpha := o]$ the syntactic ordinal κ in which the free occurrences of the ordinal variable α have been replaced by the ordinal $o \in [\![\mathcal{O}]\!]$. We will also use the notation $\kappa[\vec{\alpha} := \vec{o}]$ for multiple simultaneous substitution of ordinal variables with ordinals of $[\![\mathcal{O}]\!]$.

We will now give a semantical interpretation to our parametric syntactic ordinals, using ordinals. As syntactic ordinals contain predicates, we will need to provide an interpretation to our predicate symbols as well.

Definition 1.6. To interpret predicate symbols, we require an interpretation function (or valuation) $[\![-]\!]$ such that for all $P \in \mathcal{P}$ we have $[\![P]\!] \in [\![\mathcal{O}]\!]^{|P|} \to \{0, 1\}$. The semantics of closed (vectors of) parametric syntactic ordinals is defined inductively as follows.

$$[\![\infty]\!] = 2^{2^\omega} \qquad [\![\infty{+}\omega]\!] = 2^{2^\omega}{+}\omega \qquad [\![\kappa{+}1]\!] = [\![\kappa]\!]{+}1 \qquad [\![o]\!] = o \qquad [\![\vec{\kappa}]\!] = ([\![\kappa_1]\!], \dots, [\![\kappa_n]\!])$$

$$[\![\vec{\varepsilon}_{\vec{\alpha}<\vec{\xi}} P(\vec{\alpha}, \vec{\kappa})]\!] = \begin{cases} \text{some } \vec{o} \in [\![\mathcal{O}]\!]^{|\vec{\xi}|} \text{ with } \vec{o} < [\![\vec{\xi}]\!] \text{ and } [\![P]\!](\vec{o}, [\![\vec{\xi}]\!]) = 1 \text{ if it exists} \\ \vec{0} \text{ otherwise} \end{cases}$$

In this definition, $\vec{o} < [\![\vec{\xi}]\!]$ denotes point-wise ordering. There are many possible choices for the vector $\vec{o}$ in the case of an ordinal witness. We will consider different models, for which the choice of $\vec{o}$ will be made differently. If $\mathcal{M}$ denotes such a model, we will denote $[\![\kappa]\!]^{\mathcal{M}}$ the induced interpretation. Here $\mathcal{M}$ simply denotes the choice function for ordinal witnesses.

We now consider an ordering relation $\kappa \leq \tau$ and a strict ordering relation $\kappa < \tau$ on syntactic ordinals. Both relations will be defined in terms of a third (ternary) relation $\kappa \leq_i \tau$ in which $i \in \mathbb{Z}$. This relation will be specified using a deduction rule system including *ordinal contexts*.

Definition 1.7. An *ordinal contexts* is a finite set of ordinals represented using lists generated by the following BNF grammar.

$$\gamma, \delta \ ::= \ \emptyset \ \mid \ \gamma, \kappa$$

Ordinal contexts will contain ordinals assumed to be positive. Therefore, it will not be useful to include ordinals of the form $\tau + 1$ or ∞ in such a context.

$$\frac{i \leq 0}{\gamma \vdash \kappa \leq_i \kappa} = \qquad \frac{\gamma \vdash \kappa \leq_{i+1} \tau}{\gamma \vdash \kappa + 1 \leq_i \tau}\, s_l \qquad \frac{\gamma \vdash \kappa \leq_{i-1} \tau}{\gamma \vdash \kappa \leq_i \tau + 1}\, s_r$$

$$\frac{\gamma, \kappa_j \vdash \kappa_j \leq_{i-1} \tau}{\gamma, \kappa_j \vdash \varepsilon^j_{\vec{\alpha}<\vec{\kappa}}(P(\vec{\alpha}, \vec{v})) \leq_i \tau}\, \varepsilon \qquad \frac{\gamma \vdash \kappa_j \leq_i \tau}{\gamma \vdash \varepsilon^j_{\vec{\alpha}<\vec{\kappa}}(P(\vec{\alpha}, \vec{v})) \leq_i \tau}\, \varepsilon_w$$

Figure 1: Rules for ordinal ordering and strict ordering.

Definition 1.8. The syntactic ordinals are equipped with an ordering relation $\kappa \leq_i \tau$ with $i \in \mathbb{Z}$. Intuitively, $\kappa \leq_i \tau$ can be understood as "$\kappa + i \leq \tau$" when $i \geq 0$ and as "$\kappa \leq \tau + i$" if $i \leq 0$. Given a context of positive ordinals γ, we will have $\kappa \leq_i \tau$ if and only if the judgment $\gamma \vdash \kappa \leq_i \tau$ is derivable using the deduction rules of Figure 1. We then take $\kappa \leq_0 \tau$ as the definition of $\kappa \leq \tau$ and $\kappa \leq_1 \tau$ as the definition of $\kappa < \tau$.

Note that the deduction rule system of Figure 1 can be implemented as a deterministic and terminating procedure. Indeed, it is easy to see that the rule s_r commutes with the rules s_l, ε and ε_w. When both rules ε and ε_w may apply it is always better to use ε as it yields a lower index and will thus prove strictly more judgments (see the following lemma).

Lemma 1.9. *For every ordinal contexts γ and δ, every syntactic ordinals κ_1, κ_2 and κ_3, and for every relative number i and j we have:*

- *if $\gamma \vdash \kappa_1 \leq_i \kappa_2$ then $\gamma, \delta \vdash \kappa_1 \leq_i \kappa_2$,*
- *if $\gamma \vdash \kappa_1 \leq_i \kappa_2$ and $j \leq i$ then $\gamma \vdash \kappa_1 \leq_j \kappa_2$,*
- *if $\gamma \vdash \kappa_1 \leq_i \kappa_2$ and $\gamma \vdash \kappa_2 \leq_j \kappa_3$ then $\gamma \vdash \kappa_1 \leq_{i+j} \kappa_3$.*

Proof. The first two items are immediate by induction on the derivation. We prove the last one by induction on the sum of the size of the derivations of $\gamma \vdash \kappa_1 \leq_i \kappa_2$ and $\gamma \vdash \kappa_2 \leq_j \kappa_3$. If the last applied rule on either side is $=$, then we have $\kappa_1 = \kappa_2$ and $i \leq 0$ or $\kappa_2 = \kappa_3$ and $j \leq 0$. In both case we can use the second item to conclude.

If the last rule used on the left is s_l then $\kappa_1 = \kappa_1' + 1$. By induction hypothesis we have $\gamma \vdash \kappa_1' \leq_{i+j+1} \kappa_3$ and thus $\gamma \vdash \kappa_1 \leq_{i+j} \kappa_3$. The proof is similar if the last rule used on the right is s_r

If the last used rule on the left is ε or ε_w then we have $\kappa_1 = \varepsilon^j_{\vec{\alpha}<\vec{\kappa}'} P(\vec{\alpha}, \vec{v})$. By induction hypothesis, we get $\gamma \vdash \kappa_j' \leq_k \kappa_3$ with $k = i + j - 1$ if we applied the ε rule and $k = i + j$ if we applied the ε_w rule. This implies that $\gamma \vdash \kappa_1 \leq_{i+j} \kappa_3$ in both cases.

If the last rule used on the right is the ε or ε_w then we must be in one of the previous cases. More precisely, the rules that can be applied on the left when κ_2 is an ordinal witness are $=$, s_l, ε and ε_w. They are all treated above. □

Lemma 1.10. *Let γ be a closed context and κ_1 and κ_2 be two syntactic ordinals such that $\gamma \vdash \kappa_1 \leq_i \kappa_2$ is provable. For any model $\mathcal{M}$, if for all $\tau \in \gamma$ we have $[\![\tau]\!]^{\mathcal{M}} > 0$ then:*

- *if $i \geq 0$ then $[\![\kappa_1]\!] + i \leq [\![\kappa_2]\!]$ and*
- *if $i \leq 0$ then $[\![\kappa_1]\!] \leq [\![\kappa_2]\!] + i$.*

Proof. By induction on the derivation size. In the case of the ε rule, $[\![\varepsilon^j_{\vec{\alpha}<\vec{\xi}}(P(\vec{\alpha}, \vec{\kappa}))]\!]^{\mathcal{M}}$ is defined to be either some ordinal strictly smaller than $[\![\xi_j]\!]^{\mathcal{M}}$ or to be 0. Therefore, if

$[\![\xi_j]\!]^{\mathcal{M}} > 0$ then we always have $[\![\varepsilon^{j}_{\vec{\alpha}<\vec{\xi}}(P(\vec{\alpha},\vec{\kappa}))]\!]^{\mathcal{M}} < [\![\xi_j]\!]^{\mathcal{M}}$. If $[\![\xi_j]\!]^{\mathcal{M}} = 0$ we still have $[\![\varepsilon^{j}_{\vec{\alpha}<\vec{\xi}}(P(\vec{\alpha},\vec{\kappa}))]\!]^{\mathcal{M}} \leq [\![\xi_j]\!]^{\mathcal{M}}$. Then, the result follows by the induction hypothesis. □

2. Size change matrices

In this section, we present the formalism that we will use in the following section to relate our syntactic ordinals to the size-change principle [19].

Definition 2.1. We consider the set $\{-1, 0, \infty\}$ ordered as $-1 < 0 < \infty$. It is equipped with a semi-ring structure using the minimum operator *min* as its addition, and the composition operator $\circ$ (defined bellow) as its product. The verification that this is indeed a semi-ring is easy.

$$\begin{array}{lll} x \circ \infty = \infty & -1 \circ x = -1 & \text{if } x \neq \infty \\ \infty \circ x = \infty & x \circ -1 = -1 & \text{if } x \neq \infty \\ 0 \circ 0 = 0 & & \end{array}$$

Definition 2.2. A *size-change matrix* is simply a matrix with coefficient in $\{-1, 0, \infty\}$. Given an $n \times m$ matrix A and an $m \times p$ matrix B, the product AB is an $n \times p$ matrix C defined as follows.

$$C_{i,j} = \min_{1 \leq k \leq m} A_{i,k} \circ B_{k,j}$$

Note that this is exactly the definition of the usual matrix product expressed with the operations of our semi-ring $(\{-1, 0, \infty\}, min, \circ)$.

Lemma 2.3. *The size-change matrix product is associative.*

Proof. We consider an $n \times m$ matrix A, an $m \times p$ matrix B and a $p \times q$ matrix C. The products $L = AB$ and $R = BC$ are well-defined, and we have $L_{i,j} = \min_{1 \leq k \leq m} A_{i,k} \circ B_{k,j}$ and $R_{i,j} = \min_{1 \leq k \leq p} B_{i,k} \circ C_{k,j}$. As L is an $n \times p$ matrix and R is an $m \times q$ matrix, the products LC and AR are well-defined and both produce an $n \times q$ matrix. We thus need to show that $\min_{1 \leq k \leq p} L_{i,k} \circ C_{k,j} = \min_{1 \leq k \leq m} A_{i,k} \circ R_{k,j}$.

$$\begin{aligned} \min_{1 \leq k \leq p} L_{i,k} \circ C_{k,j} &= \min_{1 \leq k \leq p} (\min_{1 \leq l \leq m} A_{i,l} \circ B_{l,k}) \circ C_{k,j} \\ &= \min_{1 \leq k \leq p} \min_{1 \leq l \leq m} (A_{i,l} \circ B_{l,k}) \circ C_{k,j} \\ &= \min_{1 \leq k \leq m} \min_{1 \leq l \leq p} A_{i,k} \circ (B_{k,l} \circ C_{l,j}) \\ &= \min_{1 \leq k \leq m} A_{i,k} \circ (\min_{1 \leq l \leq p} B_{k,l} \circ C_{l,j}) \\ &= \min_{1 \leq k \leq m} A_{i,k} \circ R_{k,j} \end{aligned}$$

□

Definition 2.4. Let A be an $n \times m$ size-change matrix, let $(X, \leq)$ be an ordered set and $\vec{x}$, $\vec{y}$ be two vectors of X with $|\vec{x}| = n$ and $|\vec{y}| = m$. We write $\vec{y} <_A \vec{x}$ if for all $1 \leq i \leq n$ and $1 \leq j \leq m$ we have:

- $A_{i,j} = -1$ implies $y_j < x_i$ and
- $A_{i,j} = 0$ implies $y_j \leq x_i$.

Lemma 2.5. *Let $(X, \leq)$ be an ordered set and $\vec{x}$, $\vec{y}$ and $\vec{z}$ be three vectors of X with $|\vec{x}| = n$, $|\vec{y}| = m$ and $|\vec{z}| = p$. Let A be an $n \times m$ size-change matrix and B be an $m \times p$ size-change matrix. If $\vec{z} <_B \vec{y}$ and $\vec{y} <_A \vec{x}$ then $\vec{z} <_{AB} \vec{x}$*

Proof. Let us take $C = AB$. By definition, if $C_{i,j} = -1$ then there must be k such that $A_{i,k} \circ B_{k,j} = -1$. There are three possibilities:

- $A_{i,k} = -1$ and $B_{k,j} = -1$, which implies $z_j < y_k < x_i$,
- $A_{i,k} = -1$ and $B_{k,j} = 0$, which implies $z_j < y_k \leq x_i$,
- $A_{i,k} = 0$ and $B_{k,j} = -1$, which implies $z_j \leq y_k < x_i$.

In all cases we have $z_j < x_i$. Now, if $C_{i,j} = 0$ then there must be k such that $A_{i,k} \circ B_{k,j} = 0$, which implies $z_j \leq y_k \leq x_i$. □

3. Circular proofs and size change principle

We will now introduce an abstract notion of circular proof and a related *well-foundedness* condition. This framework will be used twice in the following sections. It will first be used to build circular subtyping proofs in Section 4. The type system will then be extended with circular typing proofs, in Section 7, to allow recursive programs. The first ingredient for building a circular proof system is a notion of *abstract judgment*.

Definition 3.1. A language of *abstract judgments* $\mathscr{J}$ is associated with a language of *individuals* Λ. Every symbol $J \in \mathscr{J}$ has an arity $|J|$, and it should carry exactly one element of Λ and $|J|$ syntactic ordinals (possibly 0). For all $J \in \mathscr{J}$ and $\vec{\kappa} \in \mathcal{O}^{|J|}$ we require that there is an individual $\varepsilon_x \neg J(x, \vec{\kappa}) \in \Lambda$, where x is a bound variable[4].

In the following, we will use abstract judgments to build ordinal witnesses of the following forms: $\varepsilon^i_{\vec{\alpha}<\vec{\kappa}} J(t, \vec{\alpha}, \vec{v})$, $\varepsilon^i_{\vec{\alpha}<\vec{\kappa}} \neg J(t, \vec{\alpha}, \vec{v})$, $\varepsilon^i_{\vec{\alpha}<\vec{\kappa}} \forall x J(x, \vec{\alpha}, \vec{v})$ and $\varepsilon^i_{\vec{\alpha}<\vec{\kappa}} \neg\forall x J(x, \vec{\alpha}, \vec{v})$, where J is an abstract judgment, $t \in \Lambda$ and $\vec{v}$ is a vector of syntactical ordinals. According to the terminology of section 1, this means that abstract judgements are used to build four kinds of predicates over ordinals.

Definition 3.2. Given a language of abstract judgments $\mathscr{J}$ and an associated language of individuals Λ, the semantics of ordinals is extended by induction on $\mathscr{J}$ and Λ. Given a model $\mathcal{M}$ (including a set $[\![\Lambda]\!]^{\mathcal{M}} \subseteq \Lambda$), we require that:

- An individual $t \in \Lambda$ is interpreted by $[\![t]\!]^{\mathcal{M}} \in [\![\Lambda]\!]^{\mathcal{M}}$,
- $J \in \mathscr{J}$ is interpreted by a function $[\![J]\!]^{\mathcal{M}} : [\![\Lambda]\!]^{\mathcal{M}} \times [\![\mathcal{O}]\!]^{|J|} \to \{0, 1\}$.
- If $t = \varepsilon_x \neg J(x, \vec{\kappa}) \in \Lambda$ then $[\![t]\!]^{\mathcal{M}}$ must satisfy $[\![J]\!]^{\mathcal{M}}([\![t]\!]^{\mathcal{M}}, [\![\vec{\kappa}]\!]^{\mathcal{M}}) = 0$ if possible.

A predicate $P(\vec{\alpha})$ involving an abstract judgment $J \in \mathscr{J}$ is then interpreted as follows.

- If $P(\vec{\alpha}) = J(t, \vec{\alpha})$ then $[\![P]\!]^{\mathcal{M}}(\vec{o}) = [\![J]\!]^{\mathcal{M}}([\![t]\!]^{\mathcal{M}}, \vec{o})$,
- if $P(\vec{\alpha}) = \neg J(t, \vec{\alpha})$ then $[\![P]\!]^{\mathcal{M}}(\vec{o}) = 1 - [\![J]\!]^{\mathcal{M}}([\![t]\!]^{\mathcal{M}}, \vec{o})$,
- if $P(\vec{\alpha}) = \forall x J(x, \vec{\alpha})$ then $[\![P]\!]^{\mathcal{M}}(\vec{o}) = \min_{t \in [\![\Lambda]\!]^{\mathcal{M}}} [\![J]\!]^{\mathcal{M}}(t, \vec{o})$,

[4]Intuitively $\varepsilon_x \neg J(x, \vec{\kappa}) \in \Lambda$ denotes a counter-example to $J(x, \vec{\kappa})$.

– if $P(\vec{\alpha}) = \forall x \neg J(x, \vec{\alpha})$ then $[\![P]\!]^{\mathcal{M}}(\vec{o}) = 1 - \max_{t \in [\![\Lambda]\!]^{\mathcal{M}}} [\![J]\!]^{\mathcal{M}}(t, \vec{o})$.

Definition 3.3. An *abstract sequent* $\gamma \vdash J(t, \vec{\kappa})$ is built using an ordinal context γ, an abstract judgment J, an individual $t \in \Lambda$ and syntactic ordinals $\vec{\kappa} \in \mathcal{O}^{|J|}$. Given a model $\mathcal{M}$, we say that the abstract sequent $\gamma \vdash J(t, \vec{\kappa})$ is true if $[\![J]\!]^{\mathcal{M}}([\![t]\!]^{\mathcal{M}}, [\![\vec{\kappa}]\!]^{\mathcal{M}}) = 1$ provided that $[\![\gamma]\!]^{\mathcal{M}}$ only contains positive ordinals.

In order to relate the notion of size-change matrices (defined in the previous section) to abstract sequents, we introduce *ordinal constraints*. They will allow us to concisely represent, in the form of a sequence of index, a conjunction of strict relations between the ordinals of a given vector.

Definition 3.4. A list of *ordinal constraints* C of arity n is given by a function C from $\{1, \dots, n\}$ to $\{0, 1, \dots, n\}$. Given a vector of ordinals $\vec{o}$ with $|\vec{o}| = n$, we denote $C(\vec{o})$ the vector of size n defined as $C(\vec{o})_i = \infty$ if $C(i) = 0$ and as $C(\vec{o})_i = o_j$ if $C(i) = j \neq 0$. We say that C is satisfied by $\vec{o}$ if we have $\vec{o} < C(\vec{o})$ (i.e. $\vec{o}$ is pointwise smaller than $C(\vec{o})$, using the rules of figure 1).

For example, if $C(1) = 2$ and $C(2) = 0$, $\vec{o} < C(\vec{o})$ means $o_1 < o_2$ and $o_2 < \infty$.

The building of circular proofs will require the generalization of sequents. In other words, we will sometimes need to prove that an abstract sequent is true for any ordinal parameters (satisfying some constraints) and for any individual. To this aim, we introduce the notion of *general abstract sequent*.

Definition 3.5. A *general abstract sequent* is an abstract sequent that is quantified over. It may be of the form $\forall \vec{\alpha}\, (\gamma \vdash C(\vec{\alpha}) \Rightarrow J(t, \vec{\alpha}, \vec{v}))$ or $\forall \vec{\alpha} \forall x\, (\gamma \vdash C(\vec{\alpha}) \Rightarrow J(x, \vec{\alpha}, \vec{v}))$, where γ is an ordinal context using only variables of $\vec{\alpha}$, C is an ordinal constraint of arity $|\vec{\alpha}|$, J is an abstract judgement and t is an individual. Given a model $\mathcal{M}$ we say that:

- $\forall \vec{\alpha}\, (\gamma \vdash C(\vec{\alpha}) \Rightarrow J(t, \vec{\alpha}, \vec{v}))$ is true if for all $\vec{o} \in [\![\mathcal{O}]\!]^n$ such that $[\![\gamma[\vec{\alpha} := \vec{o}]]\!]$ does not contains 0 and C is satisfied by $\vec{o}$, $[\![J]\!]^{\mathcal{M}}([\![t]\!]^{\mathcal{M}}, \vec{o}, [\![\vec{v}]\!]^{\mathcal{M}}) = 1$,
- $\forall \vec{\alpha} \forall x\, (\gamma \vdash C(\vec{\alpha}) \Rightarrow J(x, \vec{\alpha}, \vec{v}))$ is true if for all $\vec{o} \in [\![\mathcal{O}]\!]^n$ and $t \in [\![\Lambda]\!]^{\mathcal{M}}$ such that $[\![\gamma[\vec{\alpha} := \vec{o}]]\!]$ does not contain 0 and C is satisfied by $\vec{o}$, $[\![J]\!]^{\mathcal{M}}(t, \vec{o}, [\![\vec{v}]\!]^{\mathcal{M}}) = 1$.

The first kind of general abstract sequent (without quantifier on individuals) will be used in typing proof for recursive function from section 7 only. The second form will be used for subtyping proofs.

Note that, in a general abstract sequent, a judgement J may use ordinals $\vec{v}$ that are not quantified over. In the following, we will often omit to mention them explicitly. In particular, our definition implies that the ordinal context γ and the ordinals of $C(\vec{\alpha})$ cannot use ordinals of $\vec{v}$. This restriction is not essential, but a more general (and thus complex) definition of ordinal constraints seems useless in practice.

Definition 3.6. A *circular deduction rule system* is given by a set of deduction rules defined over abstract sequents (i.e. their conclusion and premises are abstract sequents) together with the following four rules for introducing and eliminating general abstract sequents.

$$\frac{\forall \vec{\alpha}(\gamma \vdash C(\vec{\alpha}) \Rightarrow J(t, \vec{\alpha})) \qquad (\gamma[\vec{\alpha} := \vec{\kappa}], \delta \vdash \kappa_i < C(\vec{\kappa})_i)_{1 \le i \le |\vec{\alpha}|}}{\gamma[\vec{\alpha} := \vec{\kappa}], \delta \vdash J(t, \vec{\kappa})}\, G$$

$$\dfrac{\begin{array}{c}[\forall\vec{\alpha}(\gamma \vdash C(\vec{\alpha}) \Rightarrow J(t,\vec{\alpha}))]_k \\ \vdots \\ \gamma[\vec{\alpha} := \vec{\kappa}] \vdash J(t,\vec{\kappa}) \qquad \text{where } \vec{\kappa} = \vec{\varepsilon}_{\vec{\alpha}<C(\vec{\alpha})}\neg J(t,\vec{\alpha})\end{array}}{\forall\vec{\alpha}(\gamma \vdash C(\vec{\alpha}) \Rightarrow J(t,\vec{\alpha}))}\, I_k$$

$$\dfrac{\forall\vec{\alpha}\forall x(\gamma \vdash C(\vec{\alpha}) \Rightarrow J(x,\vec{\alpha})) \qquad (\gamma[\vec{\alpha} := \vec{\kappa}], \delta \vdash \kappa_i < C(\vec{\kappa})_i)_{1\leq i\leq|\vec{\alpha}|}}{\gamma[\vec{\alpha} := \vec{\kappa}], \delta \vdash J(t,\vec{\kappa})}\, G^+$$

$$\dfrac{\begin{array}{c}[\forall\vec{\alpha}\forall x(\gamma \vdash C(\vec{\alpha}) \Rightarrow J(x,\vec{\alpha}))]_k \\ \vdots \\ \gamma[\vec{\alpha} := \vec{\kappa}] \vdash J(\varepsilon_x\neg J(x,\vec{\kappa}),\vec{\kappa}) \qquad \text{where } \vec{\kappa} = \vec{\varepsilon}_{\vec{\alpha}<C(\vec{\alpha})}\neg\forall x J(x,\vec{\alpha})\end{array}}{\forall\vec{\alpha}\forall x(\gamma \vdash C(\vec{\alpha}) \Rightarrow J(x,\vec{\alpha}))}\, I_k^+$$

The generalization rules G and G^+ allow one to prove an abstract sequent using a general abstract sequent. In particular, the list of ordinal constraints that is used in the left premise should be satisfied in the conclusion (see the second premise). The induction rules I_k and I_k^+ allow a general abstract sequent to be proved using itself as an hypothesis (this is the meaning of the square brackets). Note that a natural number k (unique in a proof) is used to keep track of the originating induction rule.

Of course, this allows for clearly invalid circular proof (e.g. by using the hypothesis immediately). Note that individuals of the form $\varepsilon_x\neg J(x,\vec{\kappa})$ appear in the I_k^+ rule. Their existence is required by Definition 3.1.

As a circular deduction system can be used to build incorrect proofs, we will now define a notion of well-founded circular proof. In other words, an abstract judgment will be considered correct in a circular deduction system if it is derivable and its derivation is well-founded according to a criterion based on the size-change principle [19]. To formulate the well-foundedness condition, we will need to decompose our circular proofs into *blocks*.

Definition 3.7. Given a proof Π expressed in a circular proof system, a *block* is a subproof B of Π such that its conclusion is either the conclusion of Π or some general abstract sequent, and its premises (if any) are also general abstract sequents. We also require blocks to be minimal, which means that they do not contain general abstract sequents (except in their conclusion and premises). This condition implies that a proof admits a unique decomposition into blocks. A block B has an arity $|B|$ which is 0 if the conclusion of the block is also the conclusion of Π, and it is the size of the quantified vector $\vec{\alpha}$ in the conclusion of B otherwise.

Definition 3.8. Let Π be a proof expressed in a circular proof system. The *call graph* of Π is the graph induced by the block structure of Π. Its vertices are the blocks of Π, and every block B has one outgoing edge for each of its premises. It is directed toward the block proving the considered premise. Note that the block pointed by the edge may be the block directly above B in Π, B itself, or even a block bellow B. In the latter two cases, the premise must correspond to an hypothesis in square brackets introduced by an instance of the I_k or I_k^+ rules.

Every edge (B_1, B_2) of a call graph is labeled by a size-change matrix M. To give its definition, we need to remark that a premise of a block necessarily uses the G or G^+ rules. Indeed, they are the only available rules having a general abstract sequent as a premise. As a consequence, we can represent the block B_1 as follows[5], if we only include the premise used for the definition of the edge (B_1, B_2).

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots \quad B_2 \\ \vdots \\ \forall\vec{\beta}\forall x(\delta \vdash D(\vec{\beta}) \Rightarrow K(x,\vec{\beta}))\end{array} \qquad (\delta[\vec{\beta} := \vec{\tau}], \delta' \vdash \tau_i < D(\vec{\tau})_i)_{1 \leq i \leq |\vec{\beta}|}}{\delta[\vec{\beta} := \vec{\tau}], \delta' \vdash K(u, \vec{\tau})}\,G^+ \quad B_1}{\begin{array}{c}\vdots \\ \gamma[\vec{\alpha} := \vec{\kappa}] \vdash J(\varepsilon_x \neg J(x,\vec{\kappa}), \vec{\kappa}) \text{ where } \vec{\kappa} = \vec{\varepsilon}_{\vec{\alpha} < C(\vec{\alpha})} \neg\forall x J(x,\vec{\alpha})\end{array}}}{\forall\vec{\alpha}\forall x(\gamma \vdash C(\vec{\alpha}) \Rightarrow J(x,\vec{\alpha}))}\,I_k^+$$

We can then take M to be the $|\vec{\alpha}| \times |\vec{\beta}|$ matrix defined as follows.

$$M_{j,k} = \begin{cases} -1 & \text{if } \delta[\vec{\beta} := \vec{\tau}], \delta' \vdash \tau_k < \kappa_j \\ 0 & \text{otherwise if } \delta[\vec{\beta} := \vec{\tau}], \delta' \vdash \tau_k \leq \kappa_j \\ \infty & \text{otherwise} \end{cases}$$

As the edges of a call graph are labeled by matrices, any path in the transitive closure of the graph can also be labeled using the matrix product of all the labels along the path. To be more precise, in particular for the order of the products, if there is a path from a block B_1 to B_2 with label M and a path from B_2 to B_3 with label N, we add a path from B_1 to B_3 with label MN.

A call graph has finitely many vertices and edges. There are also finitely many possible labels for paths in the graph. If we consider two paths with the same label to be equal, there can only be finitely many distinct paths in the transitive closure of a graph. The transitive closure can hence be computed in finite time by performing all the possible compositions until saturation.

Definition 3.9. We say that a proof is *well-founded* if every idempotent loop in the transitive closure of its call graph (i.e. closed path with label M such that $MM = M$) has at least one -1 on the diagonal[6] of its label.

We will now consider an example of circular proof, which derivation is given in the upper part of Figure 2. For simplicity, the terms and ordinals are not given explicitly. We will however assume that (besides reflexivity) we have $\kappa_2 \vdash \upsilon_2 < \kappa_2$ and $\tau_1 \vdash \upsilon_3 < \tau_1$. The proof can be decomposed into three blocks B_0, B_1 and B_2. The corresponding call graph is given in the lower part of Figure 2. Here are the idempotent loops in the transitive closure of this call-graph:

– there are none on B_0,
– there are two on B_1 with labels $\left(\begin{smallmatrix} 0 & \infty \\ \infty & -1 \end{smallmatrix}\right)$ and $\left(\begin{smallmatrix} -1 & \infty \\ \infty & -1 \end{smallmatrix}\right)$,
– there are three on B_2 with labels $\left(\begin{smallmatrix} -1 & \infty \\ \infty & 0 \end{smallmatrix}\right)$, $\left(\begin{smallmatrix} -1 & \infty \\ \infty & -1 \end{smallmatrix}\right)$ and $\left(\begin{smallmatrix} 0 & \infty \\ \infty & -1 \end{smallmatrix}\right)$.

We can finally conclude that our proof example is indeed well-founded as every idempotent loop is labeled with a matrix having at least one -1 on its diagonal[7].

Lemma 3.10. *The four rules given above are correct. In other words, if the premises of such a rule are semantically valid for a given model, then so is its conclusion.*

[5]The structure is the same for the three other cases: G with I_k, G^+ with I_k or G with I_k^+.

[6]Loops are necessarily labeled with square matrices.

[7]The proof example of Figure 2 has the same structure as the proof of Figure 6 page 22.

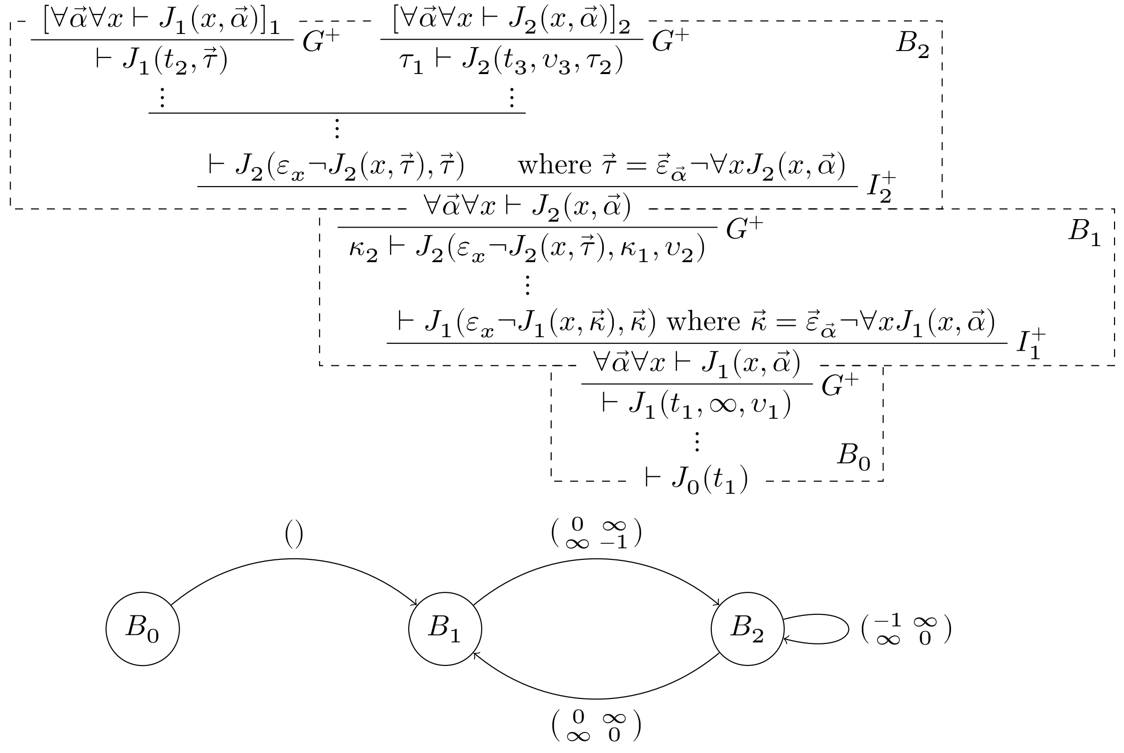

$[\forall\vec{\alpha}\forall x \vdash J_1(x,\vec{\alpha})]_1$
$\vdash J_1(t_2,\vec{\tau})$
G^+
$[\forall\vec{\alpha}\forall x \vdash J_2(x,\vec{\alpha})]_2$
$\tau_1 \vdash J_2(t_3,\upsilon_3,\tau_2)$
G^+
B_2
$\vdash J_2(\varepsilon_x\neg J_2(x,\vec{\tau}),\vec{\tau})$ where $\vec{\tau} = \vec{\varepsilon}_{\vec{\alpha}}\neg\forall x J_2(x,\vec{\alpha})$
I_2^+
$\forall\vec{\alpha}\forall x \vdash J_2(x,\vec{\alpha})$
$\kappa_2 \vdash J_2(\varepsilon_x\neg J_2(x,\vec{\tau}),\kappa_1,\upsilon_2)$
G^+
B_1
$\vdash J_1(\varepsilon_x\neg J_1(x,\vec{\kappa}),\vec{\kappa})$ where $\vec{\kappa} = \vec{\varepsilon}_{\vec{\alpha}}\neg\forall x J_1(x,\vec{\alpha})$
I_1^+
$\forall\vec{\alpha}\forall x \vdash J_1(x,\vec{\alpha})$
$\vdash J_1(t_1,\infty,\upsilon_1)$
G^+
B_0
$\vdash J_0(t_1)$
$()$
$\left(\begin{smallmatrix}0 & \infty\\ \infty & -1\end{smallmatrix}\right)$
$\left(\begin{smallmatrix}-1 & \infty\\ \infty & 0\end{smallmatrix}\right)$
$\left(\begin{smallmatrix}0 & \infty\\ \infty & 0\end{smallmatrix}\right)$
B_0
B_1
B_2

Figure 2: Circular proof example

Proof. The G and G^+ rules can be seen as the composition of standard elimination rules for the universal quantifier, followed by a weakening of the ordinal context. They are therefore correct. For the I_k and I_k^+ rules, we consider the semantics of the choice operators over ordinals (and the choice operator over individuals for the I_k^+ rule). By definition, if the conclusion of the sequent is false, then there is a counterexample that the choice operator can use. However in this case, the premise of the rule is false as well, which imply the correctness by contraposition. □

Lemma 3.11. *Let $\mathcal{M}$ be a model, C be a list of ordinal constraints of arity n and $\vec{o}$ be ordinals with $|\vec{o}| = n$. If $C(\vec{o})$ is satisfied and if $[\![P]\!]^{\mathcal{M}}(\vec{o},[\![\vec{v}]\!]^{\mathcal{M}}) = 1$, then there is a model $\mathcal{M}'$ such that $[\![P]\!]^{\mathcal{M}'} = [\![P]\!]^{\mathcal{M}}$ and $[\![\vec{v}]\!]^{\mathcal{M}'} = [\![\vec{v}]\!]^{\mathcal{M}}$ with $[\![\varepsilon^i_{\vec{\alpha}<C(\vec{\alpha})}P(\vec{\alpha},\vec{v})]\!]^{\mathcal{M}'} = \vec{o}$.*

Proof. We first take $[\![Q]\!]^{\mathcal{M}'} = [\![Q]\!]^{\mathcal{M}}$ for all predicates. We define the height of a syntactic ordinal by $h(\infty) = 0$, $h(\kappa+1) = h(k)+1$ and $h(\vec{\varepsilon}_{\vec{\beta}<D(\vec{\alpha})}Q(\vec{\beta},\vec{\tau})) = 1+\max(h(\tau_1),\dots,h(\tau))$ for ordinal witnesses. Then, we define $[\![\kappa]\!]^{\mathcal{M}'}$ by induction on $h(\kappa)$. First, we take $[\![\kappa]\!]^{\mathcal{M}'} = [\![\kappa]\!]^{\mathcal{M}}$ if $h(\kappa) < h(\varepsilon^i_{\vec{\alpha}<C(\vec{\alpha})}P(\vec{\alpha},\vec{v}))$. Second, we chose $[\![\vec{\varepsilon}_{\vec{\alpha}<C(\vec{\alpha})}P(\vec{\alpha},\vec{v})]\!]^{\mathcal{M}'} = \vec{o}$. And finally we can complete the definition of $\mathcal{M}'$ with arbitrary choices for the other ordinal witnesses. □

Proposition 3.12. *Let us assume that all the deduction rules for abstract sequents are correct with respect to the semantics. If an abstract sequent admits a well-founded circular proof then it is true in any model.*

Proof. Let us consider an abstract judgment that is derivable using a well-founded circular proof. We will assume, by contradiction, that there is a model $\mathcal{M}$ such that the considered abstract judgment is false. As all the deduction rules are supposed correct (by hypothesis and by Lemma 3.10), the call-graph of our proof necessarily contains cycles. We will thus unroll the proof to exhibit an infinite branch that will imply the existence of an infinite, decreasing sequence of ordinals (which is a contradiction).

We will now build an infinite sequence $(B_i, \vec{o}_i, \mathcal{M}_i)_{i\in\mathbb{N}}$ of triples of a block, a vector of ordinals and a model. We will take B_0 to be the block at the root of our proof, $\vec{o}_0$ to be the empty vector and $\mathcal{M}_0$ to be $\mathcal{M}$. By construction, we will enforce that for all i:

– the conclusion of B_i is false in $\mathcal{M}_i$ and $\vec{o}_i$ is a counterexample (thus $|\vec{o}_i| = |B_i|$),
– the call-graph contains an edge linking B_i to B_{i+1} and it is labeled with a matrix M_i such that $\vec{o}_{i+1} <_{M_i} \vec{o}_i$ (the conclusion of B_{i+1} is thus a general abstract).

Note that the first element of the sequence $(B_0, \vec{o}_0, \mathcal{M}_0)$ satisfies the above conditions. In particular, the conclusion of B_0 has been assumed to be false (independently of any ordinal). Moreover, the matrix labeling the edge between B_0 and B_1 will necessarily be empty.

Let us now suppose that the sequence has been constructed for all $0 \le j \le i$, and define $(B_{i+1}, \vec{o}_{i+1}, \mathcal{M}_{i+1})$. If $i \neq 0$ then the conclusion of B_i must be a general sequent, which means that the last rule in B_i is either
G or
GP. Without loss of generality we can assume that it is I_k^+, and thus B_i ends with the following.

$$\frac{\gamma[\vec{\alpha} := \vec{\kappa}] \vdash J(\varepsilon_x \neg J(x, \vec{\kappa}), \vec{\kappa}) \qquad \text{where } \vec{\kappa} = \vec{\varepsilon}_{\vec{\alpha} < C(\vec{\alpha})} \neg \forall x J(x, \vec{\alpha})}{\forall \vec{\alpha} \forall x (\gamma \vdash C(\vec{\alpha}) \Rightarrow J(x, \vec{\alpha}))}\, I_k^+$$

By construction, we know that $\forall \vec{\alpha} \forall x(\gamma \vdash C(\vec{\alpha}) \Rightarrow J(x, \vec{\alpha}))$ is false in the model $\mathcal{M}_i$ and that $\vec{o}_i$ is a counterexample. This means that $[\![\gamma[\vec{\alpha} := \vec{o}_i]]\!]^{\mathcal{M}_i}$ contains only positive ordinals, C is satisfied by $\vec{o}$ and $[\![J]\!]^{\mathcal{M}_i}(t, \vec{o}_i) = 0$ for all $t \in \Lambda$. Thus, using Lemma 3.11 we can define $\mathcal{M}_{i+1}$ to be a model such that $[\![\vec{\kappa}]\!]^{\mathcal{M}_{i+1}} = \vec{o}_i$. By definition 3.2 the indivudual $t = [\![\varepsilon_x \neg J(x, \vec{\kappa})]\!]^{\mathcal{M}_{i+1}}$ satisfies $[\![J(t, \vec{o}_i)]\!] = 0$. This establishes that the premise of our I_k^+ rule is a false abstract sequent in the model $\mathcal{M}_{i+1}$.

As all the deduction rules for abstract sequents are supposed correct, at least one premise of the block B_i must be false in the model $\mathcal{M}_{i+1}$. The first rule of such a leaf must be either G or G^+ as they are the only deduction rules having a general abstract sequent as premise. Without loss of generality we can assume a G^+ rule.

$$\frac{\forall \vec{\beta} \forall y (\delta \vdash D(\vec{\beta}) \Rightarrow K(y, \vec{\beta})) \qquad (\delta[\vec{\beta} := \vec{\tau}], \delta' \vdash \tau_i < D(\vec{\tau})_i)_{1 \le i \le |\vec{\beta}|}}{\delta[\vec{\beta} := \vec{\tau}], \delta' \vdash K(u, \vec{\tau})}\, G^+$$

As the conclusion of this rule is false is the model $\mathcal{M}_{i+1}$, we know that $[\![\delta[\vec{\beta} := \vec{\tau}], \delta']\!]^{\mathcal{M}_{i+1}}$ only contains positive ordinals and that $[\![K]\!]^{\mathcal{M}_{i+1}}([\![u]\!]^{\mathcal{M}_{i+1}}, [\![\vec{\tau}]\!]^{\mathcal{M}_{i+1}}) = 0$. By Proposition 1.10, the right premises of our G^+ rule cannot be false. As a consequence, the general abstract sequent $\forall \vec{\beta} \forall y (\delta \vdash D(\vec{\beta}) \Rightarrow K(y, \vec{\beta}))$ must be false in the model $\mathcal{M}_{i+1}$. Therefore, we can define B_{i+1} to be the block proving this sequent and $\vec{o}_{i+1}$ to be $[\![\vec{\tau}]\!]^{\mathcal{M}_{i+1}}$, which is indeed a counterexample for this sequent.

By definition, there is an edge linking the block B_i to the block B_{i+1} in the call-graph. It is labeled with a matrix M_i and we will show $\vec{o}_{i+1} <_{M_i} \vec{o}_i$ to conclude the construction of our sequence. Let us take $1 \leq m \leq |\vec{o}_i|$ and $1 \leq n \leq |\vec{o}_{i+1}|$ and consider $(M_i)_{m,n}$. If it is equal to -1 then there is a proof of $\delta[\vec{\beta} := \vec{\tau}], \delta' \vdash \tau_n < \kappa_m$ and hence proposition 1.10 gives us $[\![\tau_n]\!]^{\mathcal{M}_{i+1}} < [\![\kappa_m]\!]^{\mathcal{M}_{i+1}}$. We can hence conclude that $o_{i+1,n} < o_{i,m}$ since we have $[\![\kappa_m]\!]^{\mathcal{M}_{i+1}} = o_{i,m}$ and $o_{i+1,n} = [\![\tau_n]\!]^{\mathcal{M}_{i+1}}$ by definition of $\mathcal{M}_{i+1}$ and $\vec{o}_{i+1}$ respectively. If it is 0 then a similar reasoning can be applied to get $o_{i+1,n} \leq o_{i,m}$ and if it is ∞ then there is nothing to prove.

To conclude, we will now use that same argument as in the proof of [19, Theorem 4]. For all $0 \leq i < j$, we define $M_{i,j}$ to be the matrix $M_i M_{i+1} \dots M_{j-1}$. The number of possible different tuples of the form $(B_i, B_j, M_{i,j})$ being finite, we can apply Ramsey's theorem for pairs to find an infinite, increasing sequence of natural numbers $(u_n)_{n\in\mathbb{N}}$ such that the tuples of the form $(B_{u_i}, B_{u_j}, M_{u_i,u_j})$ with $0 \leq i < j$ are all equal. We will call M the matrix contained in all of these tuples. Thanks to the associativity of the matrix product and to the definition of $M_{i,j}$, this implies that $MM = M_{u_0,u_1} M_{u_1,u_2} = M_{u_0,u_2} = M$.

Finally, we can use Lemma 2.5 to obtain $\vec{o}_j <_{M_{i,j}} \vec{o}_i$ for all $0 \leq i < j$. Our circular proof being well-founded, the matrix M must have a -1 on the diagonal at some index k. Therefore, $\vec{o}_{u_{i+1}} <_M \vec{o}_{u_i}$ implies that $o_{u_{i+1},k} < o_{u_i,k}$ for all $i \in \mathbb{N}$, which gives an infinite, decreasing sequence of ordinals $(o_{u_i,k})_{i\in\mathbb{N}}$ and thus a contradiction. □

4. Language and type system

In this section, we consider a first (restricted) version of our language and type system. It does not provide general recursion and is shown strongly normalizing in Section 6. Surprisingly, recursion is still possible (for specific algebraic datatypes) using λ-calculus recursors that are only typable thanks to subtyping (see Section 5). The language is formed using three syntactic entities: terms, types and ordinals. Ordinals are used to annotate types with a size information that is used to show the well-foundedness of subtyping proofs. They are only introduced internally and they are not accessible to the user. However, we will see in Section 7 that the type system can be naturally extended to allow the user to express size invariants using ordinals. Although the system is Curry-style (or implicitly typed), terms, types and ordinals need to be defined simultaneously since they all include ε-choice operators. Indeed, these symbolic witnesses contain the syntactic representation of predicates involving elements of each three syntactic entities.

Definition 4.1. Let $\mathcal{V}_\Lambda = \{x, y, z \dots\}$, $\mathcal{V}_\mathcal{F} = \{X, Y, Z \dots\}$ be two disjoint and countable sets of λ-variables and propositional variables respectively. The set of terms Λ, the set of types (or formulas) $\mathcal{F}$ and the set of syntactic ordinals $\mathcal{O}$ are defined by simultaneous induction. The terms and types use the following three BNF grammars.

$$t, u \ ::= x \ | \ \lambda x.t \ | \ t\ u \ | \ \{(l_i = t_i)_{i\in I}\} \ | \ t.l_k \ | \ C_k\, t \ | \ [t \mid (C_i \to t_i)_{i\in I}] \ | \ \varepsilon_{x\in A}(t \notin B)$$

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

The predicate used to build ordinals according to section 1 are of the following forms (the two firsts are not related to abstract judgements):

- $P(t, \vec{\kappa}) = t \in A$ if the ordinals appearing in t and A are in $\vec{\kappa}$,
- $P(t, \vec{\kappa}) = t \notin A$ if the ordinals appearing in t and A are in $\vec{\kappa}$,

- $P(t, \vec{\kappa}) = t \in A \subset B$ if the ordinals appearing in t, A and B are in $\vec{\kappa}$. The choice operator $\varepsilon^i_{\vec{\alpha}<\vec{\kappa}} \neg\forall x(x \in A \subset B)$ is abbreviated as $\varepsilon^i_{\vec{\alpha}<\vec{\kappa}} A \not\subset B$.

The only form of abstract judgements will be $t \in A \subset B$ and we will only use the G^+ and I^+_k rules in this section. The G and I_k rules are in fact useless for subtyping.

According to the definition 3.1, Λ must contain a term $\varepsilon_x \neg(x \in A \subset B)$. In fact, this term is written $\varepsilon_{x\in A} x \notin B$ which will have the intended semantics.

As usual, types of the form $\mu_\kappa X.A$ and $\nu_\kappa X.A$ are restricted so that X occurs only positively in A. For example $\mu_\kappa X.X \Rightarrow X$ is not a valid type. In term witnesses such as $\varepsilon_{x\in A}(t \notin B)$, x is bound in t and B. Moreover, we require that terms of the form $\varepsilon_{x\in A}(t \notin B)$ do not contain any free λ-variable. For example $\lambda y.\varepsilon_{x\in A}(y x \notin B)$ is not a valid term.

In type witnesses such as $\varepsilon_X(t \in A)$ and $\varepsilon_X(t \notin A)$, X is bound in A only. Note that in terms, type variables are neither considered positive nor negative.

The term language contains the usual syntax of the λ-calculus extended with records, projections, constructors and pattern matching[8] (see the reduction rules of Figure 3). These constructs are included as their presence makes subtyping more useful. A term of the form $\varepsilon_{x\in A}(t \notin B)$ corresponds to an ε-choice operator denoting a closed term u of type A such that $t[x := u]$ does not have type B. The restriction to closed ε-choice operators for terms 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. Product types may be extensible. In this case we add dots at the end. An extensible record may have more fields than those explicitly mentioned. The non extensible record are used to get a more interesting type safety result from a semantic proof (theorem 6.25). Our least and greatest fixpoints (corresponding to inductive and coinductive types) carry size information in the form of a syntactic ordinals τ. The ordinal ∞ denotes the largest ordinal of our semantics and it is assumed to be large enough so that $\mu_\infty F$ and $\nu_\infty F$ denotes fixpoint of F. Two ε-choice operators $\varepsilon_X(t \in A)$ and $\varepsilon_X(t \notin A)$ are also provided. As for ε-choice operators in terms, they correspond to witnesses of the property they denote and they will be interpreted as such in the semantics. Contrary to ε-choice operators in terms, they do not need to be closed to be given an interpretation in the semantics.

Definition 4.2. The one step reduction relation $(\succ) \subseteq \Lambda \times \Lambda$ is defined as the contextual closure of the rules given in Figure 3. Note that bad terms corresponding to runtime errors are reduced to a diverging term Ω for termination to subsume type safety. We write as usual $\succ^*$ the transitive and reflexive closure of $\succ$.

Convention 4.3. To lighten the syntax and reduce the need for parentheses we will use the following syntactic sugars and notation conventions.

- Binders can be grouped (e.g. $\lambda x\,y.t = \lambda x.\lambda y.t$ and $\forall X\,Y.A = \forall X.\forall Y.A$) and
- have the lowest priority (e.g. $\lambda x.x\,x = \lambda x.(x\,x)$ and $\forall X.A \Rightarrow B = \forall X.(A \Rightarrow B)$).
- We may write $\mu X.A$ for $\mu_\infty X.A$ and $\nu X.A$ for $\nu_\infty X.A$.
- We may use the letter F to denote a type with one parameter $X \mapsto A$ so that we can write $F(\mu_\kappa F)$ for $A[X := \mu_\kappa X.A]$.

[8] In our meta-language we use the notation $\{(l_i = t_i)_{i\in I}\}$, where $I = \{i_1, \dots, i_n\}$ is a finite subset of $\mathbb{N}$, to denote the record $\{l_{i_1} = t_{i_1}; \dots; l_{i_n} = t_{i_n}\}$. Similar notations are used for pattern matching, product types and sum types. In particular, if $i \in \mathbb{N}$ then l_i is a record field label and C_i is a constructor.

$$(\lambda x.t)u \succ t[x := u] \qquad [(\lambda x.t) \mid (C_i \to t_i)_{i\in I}] \succ \Omega$$

$$\{(l_i : t_i)_{i\in I}\}.l_j \succ \begin{cases} t_j & \text{if } j \in I \\ \Omega & \text{otherwise} \end{cases} \qquad (\lambda x.t).l_k \succ \Omega$$

$$\{(l_i = t_i)_{i\in I}\}\, u \succ \Omega$$

$$[C_j\, u \mid (C_i \to t_i)_{i\in I}] \succ \begin{cases} t_j\, u & \text{if } j \in I \\ \Omega & \text{otherwise} \end{cases} \qquad (C_k\, t)\, u \succ \Omega$$

$$[\{(l_i = t_i)_{i\in I}\} \mid (C_i \to t_i)_{i\in I}] \succ \Omega$$

$$(C_k\, t).l \succ \Omega$$

Figure 3: Reduction rules of the language

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

$$\frac{\vdash t : A \to B \qquad \vdash u : A}{\vdash t\, u : B} \to_e \qquad \frac{\vdash \epsilon_{x\in A}(t \notin B) \in A \subset C}{\vdash \epsilon_{x\in A}(t \notin B) : C} \epsilon$$

$$\frac{\vdash \{(l_i = t_i)_{i\in I}\} \in \{(l_i : A_i)_{i\in I}\} \subset B \qquad (\vdash t_i : A_i)_{i\in I}}{\vdash \{(l_i = t_i)_{i\in I}\} : B} \times_i \qquad \frac{\vdash t : \{l_k : A; \ldots\}}{\vdash t.l_k : A} \times_e$$

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

Figure 4: Typing rules of the strongly normalizing system.

– In a pattern matchings we use the notation $C_k x \to t$ to denote $C_k \to \lambda x.t$.
– If C_k is a constructor, the notation $t.C_k$ corresponds to the term $[t \mid C_k \to \lambda x.x]$.

As our system relies on ε-choice operators, usual contexts giving the type of free variables are not required. In particular, open terms will never appear in typing and subtyping rules.

Definition 4.4. In addition to rather usual typing judgments of the form $\vdash t : A$, we introduce local subtyping judgements of the form $\gamma \vdash t \in A \subset B$ meaning "if t has type A, then it also has type B" (in the positivity context γ). Usual subtyping judgments of the form $\gamma \vdash A \subset B$ are then encoded as $\gamma \vdash \varepsilon_{x\in A}(x \notin B) \in A \subset B$. The typing and subtyping rules of the system are given in Figure 4 and Figure 5 respectively. The $\times^*$ subtyping rule for product type, is for extensible product.

As mentionned above, local subtyping judgements can be used as abstract judgements to build well founded circular proofs as explained in section 3. The typing proof are not allowed to be circular.

Thanks to local subtyping judgements, quantifiers can be handled in the subtyping part of the system. Using ε-choice operators allows for valid commutations of quantifiers with other connectors, without changing the syntax-directed nature of the system.

$$\frac{\gamma \vdash \epsilon_{x \in A_2}(t\,x \notin B_2) \in A_2 \subset A_1 \qquad \gamma \vdash t\;\epsilon_{x \in A_2}(t\,x \notin B_2) \in B_1 \subset B_2}{\gamma \vdash t \in A_1 \to B_1 \subset A_2 \to B_2}\;\to$$

$$\frac{}{\gamma \vdash t \in A \subset A}\;= \qquad \frac{\gamma \vdash t \in A[X := U] \subset B}{\gamma \vdash t \in \forall X.A \subset B}\;\forall_l \qquad \frac{\gamma \vdash t \in A \subset B[X := \epsilon_X(t \notin B)]}{\gamma \vdash t \in A \subset \forall X.B}\;\forall_r$$

$$\frac{\gamma \vdash t \in B \subset A[X := U]}{\gamma \vdash t \in B \subset \exists X.A}\;\exists_r \qquad \frac{\gamma \vdash t \in B[X := \epsilon_X(t \in B)] \subset A}{\gamma \vdash t \in \exists X.B \subset A}\;\exists_l$$

$$\frac{(\gamma \vdash t.l_i \in A_i \subset B_i)_{i \in I}}{\gamma \vdash t \in \{(l_i : A_i)_{i \in I}\} \subset \{(l_i : B_i)_{i \in I}\}}\;\times \qquad \frac{I_1 \subseteq I_2 \qquad (\gamma \vdash t.C_i \in A_i \subset B_i)_{i \in I_1}}{\gamma \vdash t \in [(C_i : A_i)_{i \in I_1}] \subset [(C_i : B_i)_{i \in I_2}]}\;+$$

$$\frac{I_2 \subseteq I_1 \qquad (\gamma \vdash t.l_i \in A_i \subset B_i)_{i \in I_2}}{\gamma \vdash t \in \{(l_i : A_i)_{i \in I_1}[;\ldots]\} \subset \{(l_i : B_i)_{i \in I_2};\ldots\}}\;\times^*$$

In the conclusion, $[;\ldots]$ means that the product type may be extensible or not.

$$\frac{\gamma \vdash t \in A \subset F(\mu_\tau F) \qquad \gamma \vdash \tau < \kappa}{\gamma \vdash t \in A \subset \mu_\kappa F}\;\mu_r \qquad \frac{\gamma \vdash t \in F(\nu_\tau F) \subset B \qquad \gamma \vdash \tau < \kappa}{\gamma \vdash t \in \nu_\kappa F \subset B}\;\nu_l$$

$$\frac{\gamma \vdash t \in A \subset F(\mu F)}{\gamma \vdash t \in A \subset \mu F}\;\mu_r^\infty \qquad \frac{\gamma, \kappa \vdash t \in F(\mu_\tau F) \subset B \text{ with } \tau = \varepsilon_{\alpha<\kappa} t \in F(\mu_\tau F)}{\gamma \vdash t \in \mu_\kappa F \subset B}\;\mu_l$$

$$\frac{\gamma, \kappa \vdash t \in A \subset F(\nu_\tau F) \text{ with } \tau = \varepsilon_{\alpha<\kappa} t \notin F(\nu_\tau F)}{\gamma \vdash t \in A \subset \nu_\kappa F}\;\nu_r \qquad \frac{\gamma \vdash t \in F(\nu F) \subset B}{\gamma \vdash t \in \nu F \subset B}\;\nu_l^\infty$$

Figure 5: Subtyping rules (excluding the induction rules)

5. Strongly normalisable examples

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. This first example is a good illustration of circular subtyping proof.

Our system is suitable for handling types containing both least and greatest fixpoints. Let us consider the following two types S and L where $F(X, Y)$ is a predicate covariant in X and Y.

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

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 S $\subset$ L. It is displayed in Figure 6.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[\forall\alpha_0,\alpha_1(\vdash S_{\alpha_1} \subset \mathrm{G}(L_{\alpha_0}))]_1}{\vdash x_2.A \in S_{\kappa_5} \subset \mathrm{G}(L_{\kappa_4})}\,G^+ \qquad \dfrac{\dfrac{[\forall\alpha_0,\alpha_1(\vdash \mathrm{F}(S_{\alpha_1}) \subset \mathrm{G}(L_{\alpha_0}))]_2}{\kappa_4 \vdash x_2.B \in \mathrm{F}(S_{\kappa_5}) \subset \mathrm{G}(L_{\kappa_6})}\,G^+}{\vdash x_2.B \in \mathrm{F}(S_{\kappa_5}) \subset L_{\kappa_4}}\,\nu_r}{\vdash x_2 \in [\mathrm{A\ of\ } S_{\kappa_5}|\mathrm{B\ of\ F}(S_{\kappa_5})] \subset [\mathrm{A\ of\ G}(L_{\kappa_4})|\mathrm{B\ of\ } L_{\kappa_4}]}\,+}{\vdash x_2 \in [\mathrm{A\ of\ } S_{\kappa_5}|\mathrm{B\ of\ F}(S_{\kappa_5})] \subset \mathrm{G}(L_{\kappa_4})}\,\mu_r}{\vdash x_2 \in \mathrm{F}(S_{\kappa_5}) \subset \mathrm{G}(L_{\kappa_4})}\,\nu_l}{\forall\alpha_0,\alpha_1(\vdash \mathrm{F}(S_{\alpha_1}) \subset \mathrm{G}(L_{\alpha_0}))}\,I_2^+}{\kappa_2 \vdash x_1 \in \mathrm{F}(S_{\kappa_3}) \subset \mathrm{G}(L_{\kappa_1})}\,G^+}{\vdash x_1 \in S_{\kappa_2} \subset \mathrm{G}(L_{\kappa_1})}\,\mu_l}{\forall\alpha_0,\alpha_1(\vdash S_{\alpha_1} \subset \mathrm{G}(L_{\alpha_0}))}\,I_1^+}{\infty \vdash x_0 \in \mathrm{S} \subset \mathrm{G}(L_{\kappa_0})}\,G^+}{\vdash x_0 \in \mathrm{S} \subset \mathrm{L}}\,\nu_r$$

where

$$\begin{aligned}
\mathrm{F}(X) &= \nu Y.[\mathrm{A\ of\ } X|\mathrm{B\ of\ } Y] & \kappa_4 &= \varepsilon^1_{\alpha_1,\alpha_2<\infty+\omega,\infty+\omega}(\mathrm{F}(S_{\alpha_2}) \not\subset \mathrm{G}(L_{\alpha_1}))\\
S_\alpha &= \mu_\alpha X \nu Y[A \text{ of } X|B \text{ of } Y] = \mu_\alpha X F(X) & \kappa_5 &= \varepsilon^2_{\alpha_1,\alpha_2<\infty+\omega,\infty+\omega}(\mathrm{F}(S_{\alpha_2}) \not\subset \mathrm{G}(L_{\alpha_1}))\\
\mathrm{G}(Y) &= \mu X.[\mathrm{A\ of\ } X|\mathrm{B\ of\ } Y] & \kappa_6 &= \varepsilon_{\alpha<\kappa_4}(x_2.B \notin \mathrm{G}(L_\alpha))\\
L_\alpha &= \nu_\alpha Y \mu X[A \text{ of } X|B \text{ of } Y] = \nu_\alpha Y G(Y) & x_0 &= \varepsilon_{x\in\mathrm{S}}(x \notin \mathrm{L})\\
\kappa_0 &= \varepsilon_{\alpha<\infty}(x_0 \notin \mathrm{G}(L_\alpha)) & x_1 &= \varepsilon_{x\in S_{\kappa_2}}(x \notin \mathrm{G}(L_{\kappa_1}))\\
\kappa_1 &= \varepsilon^1_{\alpha_1,\alpha_2<\infty+\omega,\infty+\omega}(S_{\alpha_2} \not\subset \mathrm{G}(L_{\alpha_1})) & x_2 &= \varepsilon_{x\in\mathrm{F}(S_{\kappa_5})}(x \notin \mathrm{G}(L_{\kappa_4}))\\
\kappa_2 &= \varepsilon^2_{\alpha_1,\alpha_2<\infty+\omega,\infty+\omega}(S_{\alpha_2} \not\subset \mathrm{G}(L_{\alpha_1})) & &\\
\kappa_3 &= \varepsilon_{\alpha<\kappa_2}(x_1 \in \mathrm{F}(S_\alpha)) & &
\end{aligned}$$

Figure 6: Proof of $\mathrm{S} = \mu X.\mathrm{F}(X) \subset \nu Y.\mathrm{G}(Y) = \mathrm{L}$.

The fact that this proof is well-founded was already studied in section 3 in figure 2.

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 [23] 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 7. Note that the choice operators for terms and types are all well defined (i.e. their definitions are not cyclic). 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.

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

where

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

Figure 7: Derivation of Mitchell's containment axiom.

$$\dfrac{\vdash t : A \qquad \vdash t \in A \subset B}{\vdash t : A : B}$$

However, such type annotations are often required to reference bound type variables. A type abstraction constructor $\Lambda X.t$ is not natural in a Church style calculus. Moreover, we only need a way for the user to denote the ε-choice operator that are constructed to replace bound variables.

A simple idea to solve this problem is to write annotation like

$$\text{let } X, \ldots \text{ such that } x : A(X, \ldots) \text{ in } \ldots$$

This allows to name ε-choice operators by pattern matching on the current type of a variable. During typing bound variables are replaced by ε-choice operators which carry their type, so this is simple to implement which is crucial for syntactic sugar.

With this syntactic sugar, a fully annotated identity can be written

$$\lambda x.\text{let } X \text{ such that } x : X \text{ in } x : X.$$

This may seem strange at first, but is very handy in practice.

We also allow a variant which match the given type with the type of the whole term:

$$\text{let } X, \ldots \text{ such that } \_ : A(X, \ldots) \text{ in } \ldots$$

The same kind of definitions can be used to define dot notation on existential types which can replace the usual dot notation for abstract types. Indeed, if a λ-variable x has type $\exists X \exists Y A(X, Y)$, we can access X and Y using the same syntactic sugar:

$$\text{let } X, Y \text{ such that } x : A(X, Y) \text{ in } \ldots$$

Because we use local subtyping when matching type, the implementation can easily search X_0 and Y_0 such that $\vdash t \in A(X_0, Y_0) \subset \exists X \exists Y A(X, Y)$. This will leads to $X_0 = \varepsilon_X t : \exists Y A(X, Y)$ and $Y_0 = \varepsilon_Y t : A(X_0, Y)$.

Yet this notation is too heavy and in this particular case, we prefer the notation $x.X$ and $x.Y$ which rely on the name of bound variables to build the same witnesses as above from the type of x, or more precisely from the type of the term witness that will be substituted to x. It is important to remark that the implementation never needs to rename a bound variables because we substitute closed terms/types/ordinals to variables and renaming is never necessary in this case.

This is a limited form of dot notation, but the same limitation is already present in most language using abstract type (like OCaml). A more general notation as $(f t).X$ could be more difficult, in particular in presence of effects, because it denotes a type that contains a possible computation.

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 both ways to annotate the definition of a function "dual" computing the opposite of a category.

$$\begin{aligned}
\mathrm{C}(O,M) &= \{\mathrm{dom} : M \to O; \mathrm{cod} : M \to O; \mathrm{cmp} : M \to M \to M\}\\
\mathrm{Cat} &= \exists O.\exists M.\mathrm{C}(O,M)\\
\mathrm{dual} &: \mathrm{Cat} \to \mathrm{Cat}\\
&= \lambda c.\left\{\begin{array}{ll}\mathrm{dom} : c.M \to c.O & = c.cod;\\ \mathrm{cod} : c.M \to c.O & = c.dom;\\ \mathrm{cmp} : c.M \to c.M \to c.M & = \lambda x\,y.c.cmp\;y\;x\end{array}\right\}\\
\mathrm{dual2} &: \mathrm{Cat} \to \mathrm{Cat}\\
&= \lambda c.\ \text{let } O, M \text{ such that } c : \mathrm{C}(O,M) \text{ in}\\
&\qquad \left\{\begin{array}{ll}\mathrm{dom} : M \to O & = c.cod;\\ \mathrm{cod} : M \to O & = c.dom;\\ \mathrm{cmp} : M \to M \to M & = \lambda x\,y.c.cmp\;y\;x\end{array}\right\}
\end{aligned}$$

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.

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

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 [18]. It can be typechecked giving only the type to use for the natural number: $N_t = (\mathrm{T} \to \mathrm{T}) \to \mathrm{T} \to \mathrm{T}$ where $\mathrm{T} = \mu X.((X \to \mathbb{B}) \to \mathbb{B})$. They are defined as follows, where $T, F : \mathbb{B}$ denote booleans.

$$\begin{aligned}
P_C &: \mathbb{N}_C \to \mathbb{N}_C = \lambda n.n\,(\lambda p\,x{:}\mathbb{N}_C\,y{:}\mathbb{N}_C.p\,(S_C\,x)\,x)\,(\lambda x\,y.y)\,0_C\,0_C\\
R_C &: \forall P.(P \to \mathbb{N}_C \to P) \to P \to \mathbb{N}_C \to P\\
&\quad = \lambda f\,a\,n.n\,(\lambda x\,p{:}\mathbb{N}_C.f\,(x\,(S_C\,p))\,p)\,(\lambda p.a)\,0_C
\end{aligned}$$

$$\begin{aligned}(\leq) &= \mathbb{N}_C \to \mathbb{N}_C \to \mathbb{B}\\ &= \lambda n\, m.(n : N_t)\,(\lambda f\, g.g\, f)\,(\lambda i.T)\,((m : N_t)\,(\lambda f\, g.g\, f)\,(\lambda i.F))\end{aligned}$$

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[24]. It is displayed below together with several other terms and types that are used for Scott encoding.

$$\begin{aligned}\text{pred} : \mathbb{N}_S \to \mathbb{N}_S &= \lambda n.n\,(\lambda p.p)\,0_S\\ \text{U}(P) &= \forall Y.Y \to \mathbb{N}_S \to P\\ \text{T}(P) &= \forall Y.(Y \to \text{U}(P) \to Y \to \mathbb{N}_S \to P) \to Y \to \mathbb{N}_S \to P\\ \mathbb{N}' &= \forall P.\text{T}(P) \to \text{U}(P) \to \text{T}(P) \to \mathbb{N}_S \to P\\ \zeta\ &: \forall P.P \to \text{U}(P) = \lambda a\, r\, q.a\\ \delta\ &: \forall P.P \to (\mathbb{N}_S \to P \to P) \to \text{T}(P)\\ &= \lambda a\, f\, p\, r\, q.f\ (\text{pred}\ q)\ (p\ r\ (\zeta\ a)\ r\ q)\\ R_S\ &: \forall P.P \to (\mathbb{N}_S \to P \to P) \to \mathbb{N}_S \to P\\ &= \lambda a\, f\, n.(n : \mathbb{N}')\,(\delta\, a\, f)\,(\zeta\, a)\,(\delta\, a\, f)\, n\end{aligned}$$

It is easy to check that the term R_S is indeed a recursor for Scott numerals. The term is similar to a fixpoint combinator, with a limited number of steps of recursion. As this term is typable, Theorem 6.23 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.

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

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 stream. 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.

$$\begin{aligned}
\mathrm{T}(A,P) &= \forall Y.(P \times Y) \to \{\mathrm{hd} : (P \times Y) \to A; \mathrm{tl} : Y; \mathrm{st} : P \times Y\} \\
\mathbb{S}'(A,P) &= \{\mathrm{hd} : (P \times \mathrm{T}(A,P)) \to A; \mathrm{tl} : \mathrm{T}(A,P); \mathrm{st} : P \times \mathrm{T}(A,P)\} \\
\zeta &: \forall A.\forall P.(P \to A) \to \forall X.(P \times X) \to A \\
&= \lambda f\, s.f\ s.1 \\
\delta &: \forall A.\forall P.(P \to A) \to (P \to P) \to \mathrm{T}(A,P) \\
&= \lambda f\, n\, s.\{\mathrm{hd} = \zeta\ f; \mathrm{tl} = s.2; \mathrm{st} = (n\ s.1, s.2)\} \\
\mathrm{coiter} &: \forall A.\forall P.P \to (P \to A) \to (P \to P) \to \mathbb{S}(A) \\
&= \lambda s\, f\, n.\ \text{let } A, P \text{ such that } f : P \to A \text{ in} \\
&\qquad \left\{\begin{array}{ll} \mathrm{hd} & = \zeta\ f; \\ \mathrm{tl} & = \delta\ f\ n; \\ \mathrm{st} & = (s, \delta\ f\ n) \end{array}\right\} : \mathbb{S}'(A,P)
\end{aligned}$$

In these definitions, we use the same names as in the previous one to show the similarities. The minimum type annotation for this program is to use the following subtyping relation: $\mathbb{S}'(\mathrm{A},\mathrm{P}) \subset \mathbb{S}(\mathrm{A})$. Here we gave also the type of ζ and δ, but it is not necessary. As for Scott's numeral, a question arise about the meaning of the type $\exists P.\mathbb{S}'(\mathrm{A}, P)$.

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 type $\mu N.[\mathrm{Z}|\mathrm{S} \text{ of } N]$, we were not able program such a recursor and we conjecture that it is impossible. However, if we encode the sum type using a record type, we can provide a recursor. The type would then be $N_t = \mu X.\forall Y.\{\mathrm{z} : Y; \mathrm{s} : X \to Y\} \to Y$ which is very similar to the type of Scott's numeral.

6. Realizability semantics

In this section, we build a realizability model that is shown adequate with our type system. In particular, a formula A is interpreted as a set of strongly normalizing *pure terms* $[\![A]\!]$ so that a term provably of type A is in $[\![A]\!]$.

Definition 6.1. A term is said *pure* when it does not contain choice operators $\varepsilon_{x \in A}(t \notin B)$. We denote $[\![\Lambda]\!] \subset \Lambda$ the set of pure terms.

Definition 6.2. 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 $\mathcal{N} \subset [\![\Lambda]\!]$ the set of strongly normalizing pure terms.

Definition 6.3. The set $\mathcal{H}$ of head contexts (i.e. terms with a hole in head position) is generated by the following grammar.

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

Given a term t and a head context H we denote $H[t]$ the term formed by plugging t into the hole of H. We extend naturally the notion of reduction to context writing $H \succ H'$ when $H[t] \succ H'[t]$ for any term $t \in \Lambda$ (including λ-variables).

Definition 6.4. We write $t \succ_H u$ for the *head reduction*, that is when a reduction rule is applied only in the hole of a head context.

Definition 6.5. We say that a set of pure terms $\Phi \subset [\![\Lambda]\!]$ is *saturated* if the following conditions hold.

- Φ is closed by head reduction.
- If $H[t[x := u]] \in \Phi$ and $u \in \mathcal{N}$ then $H[(\lambda x.t)\ u] \in \Phi$.
- If $H[t\ u] \in \Phi$, then $H[[D\ u \mid D \to t]] \in \Phi$.
- If $H[t \mid (C_i \to t_i)_{i\in I}] \in \Phi$ and $I \subset J$, and for all $j \in J \setminus I$, $t_j \in \mathcal{N}$, then
$$H[t \mid (C_i \to t_i)_{i\in J}] \in \Phi.$$
- If $H[t] \in \Phi$ and for all $i \in I$ we have $t_i \in \mathcal{N}$, then
$$H[\{l = t; (l_i = t_i)_{i\in I}\}.l] \in \Phi.$$

Note: the closure by head reduction is not usual but is needed for subtyping of sum type. More details are included at the end of this section.

Lemma 6.6. *If Φ is saturated and $t \succ_H u \in \Phi$ and $t \in \mathcal{N}$, then $t \in \Phi$.*

Proof. It is enough to prove the result for one step of head reduction, and this is immediate by definition of saturated sets and head reduction. □

Definition 6.7. The set of neutral terms $\mathcal{N}_0 \subset [\![\Lambda]\!]$ is defined as the smallest set 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 $[t \mid (C_i \to t_i)_{i\in I}] \in \mathcal{N}_0$.

The set $\mathcal{N}_0$ is not saturated. We denote by $\overline{\mathcal{N}_0}$ the smallest saturated set containing $\mathcal{N}_0$ (i.e. the intersection of all saturated sets).

Lemma 6.8. *We have $\mathcal{N}_0 \subset \overline{\mathcal{N}_0} \subset \mathcal{N}$.*

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. More precisely, a term t in $\mathcal{N}_0$ can be written $H[x]$, H using only term in $\mathcal{N}$. The definition of reduction implies that if $t \succ^* t'$ then $t' = H'[x]$ with $H \succ^* H'$. Then, we conclude because an infinite reduction of H means an infinite reduction of one of the term appearing in H. □

Lemma 6.9. *The head normal form of a pure term (when it exists) is either an abstraction $\lambda x.t$, a record $\{(l_1 = t_i)_{i\in I}\}$, a constructor Cu or a term in $\mathcal{N}_0$.*

Proof. A pure term can be written $H[u]$ where u is either an abstraction $\lambda x.t$, a record $\{(l_1 = t_i)_{i\in I}\}$, a constructor Cu or a variable. If H is non empty, u must be a variable otherwise the term can be reduced, therefore, the term is in $\mathcal{N}_0$. If H is empty, we conclude immediatly. □

Lemma 6.10. *The set $\mathcal{N}$ is saturated.*

Proof. We have four cases to prove:

- Let H be a head context, $t \in \Lambda$ and $u \in \mathcal{N}$, such that $H[t[x := u]] \in \mathcal{N}$. If $H[(\lambda x.t)\ u]$ has an infinite reduction, then we can assume that there is no infinite reduction in H or t (because otherwise $H[t[x := u]] \notin \mathcal{N}$). Therefore, the infinite reduction must start by $H[(\lambda x.t)\ u] \succ^* H'[(\lambda x.t')\ u'] \succ H'[t' := u'] \succ^* ...$ with $H \succ^* H'$, $t \succ^* t'$ and $u \succ^* u'$. This reduction may be transformed into $H[(\lambda x.t)\ u] \succ H[t[x := u]] \succ^* H'[t' := u'] \succ^* ...$.

- Let H be a head context, $t, u \in \Lambda$ and $u \in \mathcal{N}$, such that $H[t\,u] \in \mathcal{N}$. If $H[[D\ u \mid D \to t]]$ has an infinite reduction, then we can assume that there is no infinite reduction in H, u or t (because otherwise $H[t\,u] \notin \mathcal{N}$). Therefore, the infinite reduction must start by $H[[D\ u \mid D \to t]] \succ^* H'[[D\,u' \mid D \to t']] \succ H'[t'\,u'] \succ^* ...$ with $H \succ^* H'$, $t \succ^* t'$ and $u \succ^* u'$. This reduction can transformed into $H[[D\ u \mid D \to t]] \succ H[t\,u] \succ^* H'[t'\,u'] \succ^* ...$.
- Let H be a head context, $t \in \Lambda$, $I \subset J$ with $t_i \in \Lambda$ for $i \in I$ and $t_j \in \mathcal{N}$ for $j \in J \quad I$ such that $H[[t \mid (C_i \to t_i)_{i\in I}]] \in \mathcal{N}$. If $H[[t \mid (C_i \to t_i)_{i\in J}]]$ has an infinite reduction, then we can assume that there is no infinite reduction in H, t or any of the t_i for $i \in I$ (because otherwise $H[[t \mid (C_i \to t_i)_{i\in I}]] \notin \mathcal{N}$). Therefore, the infinite reduction must start by $H[[t \mid (C_i \to t_i)_{i\in J}]] \succ^* H'[[C_k u \mid (C_i \to t'_i)_{i\in J}]] \succ H'[t'_k\,u] \succ^* ...$ with $H \succ^* H'$, $t \succ^* C_k u$ and $t_i \succ^* t'_i$ for all $i \in J$. Then we have the following infinite reduction if $k \in I$: $H[[t \mid (C_i \to t_i)_{i\in I}]] \succ^* H'[[C_k u \mid (C_i \to t'_i)_{i\in I}]] \succ H'[t'_k\,u] \succ^* ...$ and the following infinite reduction if $k \in J \setminus I$: $H[[t \mid (C_i \to t_i)_{i\in I}]] \succ^* H'[[C_k u \mid (C_i \to t'_i)_{i\in I}]] \succ H'[\Omega] \succ H'[\Omega] \succ^* ...$.
- Let H be a head context, $t \in \Lambda$, $I \subset N$ with $t_i \in \mathcal{N}$ for $i \in I$ such that $H[t] \in \mathcal{N}$. If $H[\{l = t; (l_i = t_i)_{i\in I}\}.l]$ has an infinite reduction, then we can assume that there is no infinite reduction in H or t. Therefore, the infinite reduction must start by $H[\{l = t; (l_i = t_i)_{i\in I}\}.l] \succ^* H[\{l = t'; (l_i = t'_i)_{i\in I}\}.l] \succ t' \succ^* ...$ with $H \succ^* H'$, $t \succ^* t'$ and $t_i \succ^* t'_i$ for all $i \in I$. This reduction may be transformed to $H[\{l = t; (l_i = t_i)_{i\in I}\}.l] \succ^* H[t] \succ H'[t'] \succ^* ...$

□

Definition 6.11. Given two sets $\Phi_1 \subset \Lambda$ and $\Phi_2 \subset \Lambda$, we define $(\Phi_1 \Rightarrow \Phi_2) \subset \Lambda$ as follows.

$$\Phi_1 \Rightarrow \Phi_2 = \{t \in \Lambda \mid \forall u \in \Phi_1,\ t\,u \in \Phi_2\}$$

Lemma 6.12. *Let Φ_1, Φ_2, Ψ_1, $\Psi_2 \subset \Lambda$ be sets of terms such that $\Phi_2 \subset \Phi_1$ and $\Psi_1 \subset \Psi_2$. We have $(\Phi_1 \Rightarrow \Psi_1) \subset (\Phi_2 \Rightarrow \Psi_2)$.*

Proof. Immediate by definition. □

Lemma 6.13. *We have $\mathcal{N}_0 \subset (\mathcal{N} \Rightarrow \mathcal{N}_0) \subset (\mathcal{N}_0 \Rightarrow \mathcal{N}) \subset \mathcal{N}$.*

Proof. By Lemma 6.8 we know that $\mathcal{N}_0 \subset \mathcal{N}$ and hence we obtain $(\mathcal{N} \Rightarrow \mathcal{N}_0) \subset (\mathcal{N}_0 \Rightarrow \mathcal{N})$ using Lemma 6.12. 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 $\mathcal{N}_0 \subset (\mathcal{N} \Rightarrow \mathcal{N}_0)$. Finally, if we take $t \in (\mathcal{N}_0 \Rightarrow \mathcal{N})$ then by definition $t\,x \in \mathcal{N}$ since $x \in \mathcal{N}_0$. Hence $t \in \mathcal{N}$, which gives us $(\mathcal{N}_0 \Rightarrow \mathcal{N}) \subset \mathcal{N}$. □

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 $\mathcal{N}_0 \subset [\![A]\!] \subset \mathcal{N}$. We denote $[\![\mathcal{F}]\!]$ the set of every such type interpretation.

$$[\![\mathcal{F}]\!] = \{\Phi \subset [\![\Lambda]\!] \mid \Phi \text{ saturated},\ \mathcal{N}_0 \subset \Phi \subset \mathcal{N}\}$$

- A syntactic ordinal $\kappa \in \mathcal{O}$ will be interpreted by an ordinal $[\![\kappa]\!] \in [\![\mathcal{O}]\!]$ defined in section 1.

To simplify the definition of the semantics, the syntax of terms, formulas and ordinals is extended with elements of their respective models. This avoids to always carry the interpretation function giving the value of the free variables.

Definition 6.14. 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. This was already defined for ordinals in section 1.

Definition 6.15. 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. λ-variables must be allowed because of the definition of $\mathcal{N}_0$.

Definition 6.16. 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 8 and definition 1.6 for ordinals. The semantics of terms and types need to be defined mutually inductively because of the choice operators.

To interpret $\mu_\kappa F$ and $\nu_\kappa F$, we need respectively the union with $\overline{\mathcal{N}_0}$ and the intersection with $\mathcal{N}$ ortherwise we would get $[\![\mu_0 F]\!] = \emptyset$ and $[\![\nu_0 F]\!] = \Lambda$.

In the interpretation of an ε-choice operator $\varepsilon_{x\in A}(t \notin B)$, it is important that no λ-variable other that x is bound in t. This is enforced by the syntactic restriction that was given in definition 4.1. 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 the ε-choice operators.

Lemma 6.17. *LetB be a closed parametric type, we have*

- $[\![t[X := B]]\!] = [\![t[X := [\![B]\!]]]\!]$ *for any term t with $t[X := B]$ closed parametric.*
- $[\![\kappa[X := B]]\!] = [\![\kappa[X := [\![B]\!]]]\!]$ *for any ordinal κ with $\kappa[X := B]$ closed parametric.*
- $[\![A[X := B]]\!] = [\![A[X := [\![B]\!]]]\!]$ *for any type A with $xA[X := B]$ closed parametric.*

Proof. Immediate by mutual induction on the definition of the semantics. □

Lemma 6.18. *For any term $t \in \mathcal{N}$, if t has a weak head normal form, then a weak head normal form can be reached by weak head reduction only.*

Proof. The lemma is easy thanks to the hypothesis that $t \in \mathcal{N}$ because this implies that the weak-head reduction terminates. It is true without the hypothesis $t \in \mathcal{N}$ but we don't need it. □

Lemma 6.19. $t \in [\![[(C_i \text{ of } A_i)_{i\in I}]]\!]$ *if and only if $t \in \mathcal{N}$ and t reduces by head reduction to a term in $\mathcal{N}_0$ or to $C_i u$ with $i \in I$ and $u \in [\![A_i]\!]$.*

Proof. We first prove the left to right direction. From $t \in [\![[(C_i \text{ of } A_i)_{i\in I}]]\!]$, we deduce that $[t \mid (C_i \to \lambda x.x)_{i\in I}] \in \mathcal{N}$, because $\lambda x.x \in [\![A_i]\!] \Rightarrow \mathcal{N}$ for all $i \in I$. This means that t can not reduce to an abstraction, a record or Du with D distinct from all C_i. If t reduces to a term in $\mathcal{N}_0$, we have nothing left to prove by lemma 6.18. The only remaining case is when there exists $i \in I$ such that $t \succ_H^* C_i u$. Then, we consider the

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

$$\begin{aligned}
[\![\Phi]\!] &= \Phi \ \text{ if } \Phi \in [\![\mathcal{F}]\!] \\
[\![A \Rightarrow B]\!] &= [\![A]\!] \to [\![B]\!] \\
[\![\{(l_i : A_i)_{i\in I}; \dots\}]\!] &= \begin{cases} \{t \in \Lambda \mid \forall i \in I,\ t.l_i \in [\![A_i]\!]\} \text{ if } I \neq \emptyset \\ \mathcal{N} \text{ otherwise} \end{cases} \\
[\![\{(l_i : A_i)_{i\in I}\}]\!] &= \{(l_i : A_i)_{i\in I}; \dots\} \cap \{t \in \Lambda \mid t.l \succ \Omega \text{ if } \forall i \in I l \neq l_i\} \\
[\![[(C_i : A_i)_{i\in I}]]\!] &= \cap_{\Phi\in[\![\mathcal{F}]\!]}\{t \in \Lambda \mid \forall (t_i \in ([\![A_i]\!] \to \Phi))_{i\in I}, \\
&\qquad\qquad [t \mid (C_i \to t_i)_{i\in I}] \in \Phi\} \\
[\![\forall X A]\!] &= \cap_{\Phi\in[\![\mathcal{F}]\!]}[\![A[X := \Phi]]\!] \\
[\![\exists X A]\!] &= \cup_{\Phi\in[\![\mathcal{F}]\!]}[\![A[X := \Phi]]\!] \\
[\![\mu_\kappa X A]\!] &= \overline{\mathcal{N}_0} \cup \left(\cup_{o<[\![\kappa]\!]}[\![A[X := \mu_o X A]]\!]\right) \\
[\![\nu_\kappa X A]\!] &= \mathcal{N} \cap \left(\cap_{o<[\![\kappa]\!]}[\![A[X := \nu_o X A]]\!]\right) \\
[\![\varepsilon_X(t \in A)]\!] &= \begin{cases} \Phi \in [\![\mathcal{F}]\!] \text{ s.t. } [\![t]\!] \in [\![A[X := \Phi]]\!] \\ \mathcal{N} \text{ if there is no such } \Phi \end{cases} \\
[\![\varepsilon_X(t \notin A)]\!] &= \begin{cases} \Phi \in [\![\mathcal{F}]\!] \text{ s.t. } [\![t]\!] \notin [\![A[X := \Phi]]\!] \\ \mathcal{N} \text{ if there is no such } \Phi \end{cases}
\end{aligned}$$

Figure 8: Semantical interpretation of closed parametric terms, semantic ordinals and types.

term $u_0 = [t \mid C_i \to \lambda x.x \mid (C \to \lambda x.y)_{j\in I\setminus\{i\}}]$ where y is a free variable. We have $\lambda x.x \in [\![A_i]\!] \Rightarrow [\![A_i]\!]$ and $\lambda x.y \in [\![A]\!] \Rightarrow \mathcal{N}_0 \subset [\![A]\!] \Rightarrow [\![A_i]\!]$ for $j \in I \setminus \{i\}$ and therefore $u_0 \in [\![A_i]\!]$. But from $t \succ_H^* C_i u$ we deduce $u_0 \succ_H^* u$ and therefore $u \in [\![A_i]\!]$ because $[\![A_i]\!]$ is closed by head reduction.

For the converse direction, we have $t \in \mathcal{N}$, and the result is a consequence of the saturation of $[\![[(C_i \text{ of } A_i)_{i\in I}]]\!]$. □

Lemma 6.20. *$t \in [\![\{(l_i : A_i)_{i\in I}; \dots\}]\!]$ (resp. $[\![\{(l_i : A_i)_{i\in I}\}]\!]$) if and only if $t \in \mathcal{N}$ and t reduces by head reduction to a term in $\mathcal{N}_0$ or to $\{(l_i = u_i)_{i\in J}\}$ with $I \subset J$ (resp. $I = J$) and $u_i \in [\![A_i]\!]$ for all $i \in I$.*

Proof. We first prove the left to right direction. By definition of $[\![\{(l_i : A_i)_{i\in I}; ...\}]\!]$ (resp. $[\![\{(l_i : A_i)_{i\in I}; ...\}]\!]$), we know that $t.l_i \in [\![A_i]\!]$ for all $i \in I$. This implies that t must reduce to a term in $\mathcal{N}_0$ or a record where the fields $(l_i)_{i\in I}$ are present. This reduction can be done using weak head reduction by lemma 6.18. In the second case, the term attached to l_i in this record must be in $[\![A_i]\!]$ which ends the proof by saturation of $[\![A_i]\!]$. If the product type is not extentible, and if t reduce to a record with an extra field $l \neq l_i$ for all $i \in I$, then we have $t.l \succ u$ and $u \in \mathcal{N}$ because otherwise $t \notin \mathcal{N}$, hence $t.l_i \notin \mathcal{N}$.

For the converse, From $t \in \mathcal{N}$, we do have $t \succ_H^* \{(l_i = u_i)_{i\in J}$ with $I \subset J$ (resp. $I = J$) and $u_i \in [\![A_i]\!]$ implies $t \in [\![\{(l_i : A_i)_{i\in I}; ...\}]\!]$ (resp. $[\![\{(l_i : A_i)_{i\in I}; ...\}]\!]$) by saturation. □

Theorem 6.21. *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 $u \in \mathcal{N} \subset [\![\Lambda]\!]$ since $\mathcal{N}_0 \subset [\![A]\!] \subset \mathcal{N}$ by induction hypothesis. For ordinals, the proof is immediate by definition and induction hypothesis. The saturation is easy for types because the closure of the definition of the semantics by weak head expansion is exactly what is needed. For the inclusions, using lemma 6.10 we obtain $\mathcal{N} \in [\![\mathcal{F}]\!]$, hence the proof is immediate for types of the forms Φ, $\{\}$, $\varepsilon_X(t \in A)$ or $\varepsilon_X(t \notin A)$. The remaining cases are given below.

– If it is of the form $A \Rightarrow B$, then we have $\mathcal{N}_0 \subset [\![A]\!] \subset \mathcal{N}$ and $\mathcal{N}_0 \subset [\![B]\!] \subset \mathcal{N}$ by induction hypothesis. We need to show that $\mathcal{N}_0 \subset [\![A \Rightarrow B]\!] \subset \mathcal{N}$. By Lemma 6.13, it is enough to show $(\mathcal{N} \Rightarrow \mathcal{N}_0) \subset [\![A \Rightarrow B]\!] \subset (\mathcal{N}_0 \Rightarrow \mathcal{N})$. We can hence conclude using Lemma 6.12.
– If it is of the form $[(C_i \text{ of } A_i)_{i\in I}]$, we use lemma 6.19.
– If it is of the form $\{(l_i : A_i)_{i\in I\neq\emptyset}; ...\}$ (resp. $\{(l_i : A_i)_{i\in I\neq\emptyset}\}$) we get the wanted result by lemma 6.20.
– If it is of the form $\forall X.A$ or $\exists X.A$ then for every $\Phi \in [\![\mathcal{F}]\!]$ we have $\mathcal{N}_0 \subset [\![A[X := \Phi]]\!] \subset \mathcal{N}$ by induction hypothesis. As a consequence we obtain that $\mathcal{N}_0 \subset [\![\forall X.A]\!] \subset \mathcal{N}$ and that $\mathcal{N}_0 \subset [\![\exists X.A]\!] \subset \mathcal{N}$.
– If it is of the form $\mu_\kappa X.A$ or $\nu_\kappa X.A$, we prove that $\mathcal{N}_0 \subset [\![\mu_o X.A]\!] \subset \mathcal{N}$ for $o \leq [\![\kappa]\!]$ by induction on the ordinal o. If $o = 0$, this is immediate as $[\![\mu_0 X.A]\!] = \mathcal{N}_0$. Otherwise, $[\![\mu_o X.A]\!] = \cup_{o'<o}[\![A[X := \mu_{o'} X.A]]\!]$. By induction hypothesis on o', we have $\mathcal{N}_0 \subset [\![\mu_{o'} X.A]\!] \subset \mathcal{N}$. We have $[\![A[X := \mu_{o'} X.A]]\!] = [\![A[X := [\![\mu_{o'} X.A]\!]]]\!]$ by lemma 6.17, hence $\mathcal{N}_0 \subset [\![A[X := [\![\mu_{o'} X.A]\!]]]\!] \subset \mathcal{N}$ by induction hypothesis on A.
– The case of $\nu_\kappa X.A$ is identical. □

Theorem 6.22. *Let A and B be types, t be a term, κ_1, κ_2 two syntactic ordinals and γ a finite set of ordinals. such that $[\![\tau]\!] > 0$ for all $\tau \in \gamma$.*

– *If $\gamma \vdash t \in A \subset B$ is derivable by a well-founded proof and $[\![t]\!] \in [\![A]\!]$ then $[\![t]\!] \in [\![B]\!]$.*
– *If $\vdash t : A$ is derivable by a well-founded proof then $[\![t]\!] \in [\![A]\!]$.*

Proof. By lemma 1.10 we already have the correctness of the rule for ordinal comparison and by the proposition 3.12 we only have to prove that the subtyping rules are correct to deduce that circular proofs are correct too.

We prove each rule, using the variables names introduced in figure 5 and 10. We start by the subtyping rules.

$\Rightarrow$ We assume that $[\![t]\!] \in [\![A_1 \Rightarrow B_1]\!]$ (1) and $[\![t]\!] \notin [\![A_2 \Rightarrow B_2]\!]$. The latter means that we can find $u \in [\![A_2]\!]$ such that $[\![t]\!]u \notin [\![B_2]\!]$. This implies that $v = [\![\varepsilon_{x\in A_2}(t\, x \notin B_2)]\!]$

satisfies $v \in [\![A_2]\!]$ and $[\![t]\!]\ v \notin [\![B_2]\!]$ (2) by definition of the choice operator. By induction hypothesis, we deduce $v \in [\![A_1]\!]$. From (1), we get $[\![t]\!]\ v \in [\![B_1]\!]$ which gives $[\![t]\!]\ v \in [\![B_2]\!]$ with the second induction hypothesis and contradicts (2).

$=$ Immediate

$\forall_l$ We assume $[\![t]\!] \in [\![\forall X.A]\!]$. By definition, this means that $[\![t]\!] \in [\![A[X := \Phi]]\!]$ for any $\Phi \in [\![\mathcal{F}]\!]$. Applying this with $[\![U]\!]$ gives $[\![t]\!] \in [\![A[X := U]]\!]$ using lemmas 6.17 and 6.21 and the induction hypothesis then gives $[\![t]\!] \in [\![B]\!]$.

$\forall_r$ We assume $[\![t]\!] \in [\![A]\!]$ and show $[\![t]\!] \in [\![\forall X.B]\!]$. By induction hypothesis $[\![t]\!] \in [\![B[X := \varepsilon_X(t \notin B)]]\!]$. Consequently we have $[\![t]\!] \in [\![B[X := \Phi]]\!]$ for all $\Phi \in [\![\mathcal{F}]\!]$ by definition of the choice operator. We hence obtain $[\![t]\!] \in [\![\forall X.B]\!]$.

$\exists_l$, $\exists_r$ dual to the two previous rules.

$\times$ Let us assume that $[\![t]\!] \in [\![\{(l_i : A_i)_{i\in I_1}\}]\!]$ (1). We must prove $[\![t]\!] \in [\![\{(l_i : B_i)_{i\in I_2}\}]\!]$. By definition of the semantics of product, we must prove that for any $i \in I_2 \subset I_1$, $[\![t]\!].l_i \in [\![B_i]\!]$. From (1) and $I_2 \subset I_1$ we get $[\![t]\!].l_i \in [\![A_i]\!]$ and the induction hypothesis gives the wanted result.

$+$ Let us assume $[\![t]\!] \in [\![[(C_i \text{ of } A_i)_{i\in I_1}]]\!]$. By lemma 6.19, $t \succ_H^* v \in \mathcal{N}_0$ or there exists $i \in I_1$ such that $t \succ_H^* C_i u$ and $u \in [\![A_i]\!]$. The first case is immediate by saturation. In the second case, we have $t.C_i \succ_H^* u$, and by saturation, we get $t.C_i \in [\![A_i]\!]$. By induction hypothesis, we obtain $t.C_i \in [\![B_i]\!]$ and by closure by reduction $u \in [\![B_i]\!]$. We conclude using lemma 6.19 in the opposite direction.

μ_l Let us assume $[\![t]\!] \in [\![\mu_\kappa F]\!]$. By definition, this means that we can find $o < [\![\kappa]\!]$ such that $[\![t]\!] \in [\![F(\mu_o X.A)]\!]$. By definition of the interpretation of the ε-choice operator, this means that $o = [\![\varepsilon_{\alpha<\kappa} t \in F(\mu_\alpha F)]\!]$ does verify $o < [\![\kappa]\!]$ and $[\![t]\!] \in [\![F(\mu_o X.A)]\!]$. From the induction hypothesis, we can conclude $[\![t]\!] \in [\![B]\!]$.

μ_r We have $[\![\mu_\kappa F]\!] = \cup_{o<[\![\kappa]\!]}[\![F(\mu_o F)]\!]$. From $\gamma \vdash \tau < \kappa$, we deduce $[\![\tau]\!] < [\![\kappa]\!]$ and therefore, we have $[\![F(\mu_{[\![\tau]\!]} F)]\!] \subset [\![\mu_\kappa F]\!]$, which is enough to get the wanted result by induction.

μ_r^∞ The cardinal of the ordinal $[\![\infty]\!]$ is $2^{2^\omega} + 1$ and it is larger than the cardinal of $[\![\mathcal{F}]\!]$ which is 2^{2^ω}. Hence the inductive definition of $[\![\mu F]\!]$ must reach its fixpoint strictly before $[\![\infty]\!]$. Thus we have $[\![\mu F]\!] = [\![F(\mu F)]\!]$ (this shows that this rule is invertible and could be used on both side).

ν_l, ν_l^∞, ν_r dual to the three previous rules.

We now deal with the typing rules.

ε By definition of the semantics of $\varepsilon_{x\in A}(t \notin B)$, we have $u = [\![\varepsilon_{x\in A}(t \notin B)]\!] \in [\![A]\!]$, and hence $u \in [\![C]\!]$ by induction hypothesis.

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

$\Rightarrow_e$ Immediate by induction and definition of $[\![A \Rightarrow B]\!]$.

$+_i$ By induction, we have $[\![t]\!] \in [\![A]\!]$. We need to prove $[\![Ct]\!] \in [\![B]\!]$. Using the induction hypothesis on the second premise, it is enough to prove $[\![Ct]\!] \in [\![[C \text{ of } A]]\!]$, which is a consequence of lemme 6.19.

$+_e$ Immediate by induction and definition of $[\![[(C_i : A_i)_{i\in I}]]\!]$.

$\times_i$ By induction, we have $[\![t_i]\!] \in [\![A_i]\!]$ for all $i \in I$. From this we get $[\![\{(l_i = t_i)_{i\in I}]\!] \in [\![\{(l_i : A_i)_{i\in I}\}]\!]$ by lemma 6.20. The correctness of the subtyping gives $[\![\{(l_i : A_i)_{i\in I}\}]\!] \in [\![B]\!]$ and we can conclude.

$\times_e$ Immediate by induction and definition of $[\![\{(l_i : A_i)_{i\in I}\}]\!]$.

□

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 6.23. *Let t be a pure term. If there is a type A such that $\vdash t : A$ then t is strongly normalizable.*

Proof. First, we can assume that A has no free type nor ordinal variable, because we have no rule specific to such variables, replacing all type variables by the same type (for instance $\exists X.X$) and all ordinal variables by the same ordinal (for instance ∞), to build a type A', we will still have $\vdash t : A'$. Then, using the adequacy lemma (Theorem 6.22), we obtain that $[\![t]\!] \in [\![A]\!]$. We hence get $t \in [\![A]\!] \subset \mathcal{N}$ using Theorem 6.21. □

A direct consequence of strong normalization is that well-typed terms cannot produce runtime errors thanks to our definition of reduction.

Definition 6.24. We say that a data type is simple if it is closed and only contains sums, non extensible products and fixpoints limited to μ_∞. Moreover, to simplify the proof of the theorem below we consider that A has no consecutive μ_∞, nor μ_∞ applied directly to a type variable (like $\mu_\infty X.Y$ or $\mu_\infty X.X$).

This definition covers most common inductive data types like natural numbers, lists or binary trees.

Theorem 6.25. *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 such that $\vdash u : A$ is derivable.*

Proof. Using strong normalization (Theorem 6.23) we know that t reduces to a normal form u that remains closed (no free variables are introduced during reduction). We proceed by induction on the size of u. If $A = \mu_\infty X.B$ we know that $[\![A]\!] = [\![B[X := A]]\!]$. Let us define $A' = B[X := A]$ if $A = \mu_\infty X.B$ and $A' = A$ otherwise. The hypothesis on μ_∞ are still true for $B[X := A]$ because we did not allow μ_∞ applied to type variables. By hypothesis, we know that A' is a sum type or a non extensible product type.

If A' is a sum type $[(C_i : A_i)_{i\in I}]$, by lemma 6.19 and the fact that u is normal and $u \in [\![A']\!]$ we conclude that $u = C_i v$ and $v \in [\![A_i]\!]$. By induction hypothesis, we have $\vdash v : A_i$. If $A' = A$, this allows to conclude $\vdash v : A'$: using the typing rule for sums. If $A = \mu XB$ and $A' = [\![B[X := A]]\!]$, we build the following proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash (C_i\, v).C_i \in [C \text{ of } A_i] \subset A_i}\,=}{\vdash C_i\, v \in [C \text{ of } A_i] \subset B[X := A]}\,+}{\vdash C_i\, v \in [C \text{ of } A_i] \subset A}\,\mu_r^\infty \qquad \vdash v : A_i}{\vdash C_i\, v : A}\,+_i$$

If A' is a non extensible product type, we conclude similarly use lemma 6.20. □

Theorem 6.26 (logical consistency)**.** *There is no closed term t such that we can derive $\vdash t : \forall X.X$ or $\vdash t : [\,]$.*

Proof. This result follows from Theorem 6.22 using the fact that $[\![\forall X.X]\!] = [\![[\,]]\!] = \mathcal{N}_0$ only contains open terms. $\square$

We now discuss the necessity of the closure by head reduction in the definition of saturation. It is necessary to have the theorem 6.25, but this is not the main reason. This was really needed for the lemma 6.19 which is used for the correctness of the subtyping rule for sum types when proving $\vdash t \in [(C_i \text{ of } A_i)_{i\in I_1}] \subset [(C_i \text{ of } B_i)_{i\in I_2}]$.

This is because we use local subtyping with a simple witness $t.C_i$. We could use a more complex witness that in fact would be similar to the witness introduced by the encoding of sums as product: $[(C_i \text{ of } A_i)_{i\in I}] = \forall X\{(C_i : A_i \Rightarrow X)_{i\in I}\} \Rightarrow X$. Such an encoding is not a good idea in practice, it would make it difficult to implement deep pattern matching or generalised algebraic data types. But there is a more fundamental problem: to prove $\vdash t \in [(C_i \text{ of } A_i)_{i\in I_1}] \subset [(C_i \text{ of } B_i)_{i\in I_2}]$ this encoding introduce a term witness that mentions both members of the subtyping relation. This would prevent to prove some subtyping like $\forall X[C \text{ of } A] \subset [C \text{ of } \forall X A]$ or $[C \text{ of } \exists X A] \subset \exists X[C \text{ of } A]$.

7. Fixpoint and termination

We will now extend the system to allow typing terminating program using recursion. This requires extending terms with a fixpoint combinator $Yx.t$ which is a binder. In this section, termination does not mean strong normalisation, as this is not possible with fixpoints. Instead, we will consider weak reduction, that is reduction that never reduce under binders (i.e. λ-abstraction) and right members of case analysis.

Moreover, to prove termination of certain program, like quick sort, one needs to prove that some function preserves sizes. To do so, we will provide quantification over ordinals in types. This will allow to write $\forall A.\forall B.\forall\alpha.(A \Rightarrow B) \Rightarrow (L_\alpha(A) \Rightarrow L_\alpha(B))$ for the type of the map function with $L_\alpha(A) = \mu_\alpha X.[\text{Nil of } \{\}, \text{Cons of } \{car : A, cdr : X\}]$. Thanks to subtyping, the above type is a subtype of the more usual type $\forall A.\forall B.(A \Rightarrow B) \Rightarrow (L_\infty(A) \Rightarrow L_\infty(B))$ which means that we do not need two versions of the map function.

In the subtyping rules we will reuse the choice operators $\epsilon_{\alpha<\infty}(t \in A)$ and $\epsilon_{\alpha<\infty}(t \notin A)$ for quantification.

Finally, to work with quantification over ordinals in types, and more generally to prove termination of recursive programs, typing judgement will also require a positivity context: $\gamma \vdash t : A$. We also need to extract positivity hypothesis for some ordinals. Indeed, if we have $l : L_\alpha(A)$, when typing u and v inside a case analysis $[l|\text{Nil} \to u|\text{Cons} \to v]$, we can assume that $\alpha > 0$ and failing to do so would often prevent to have $u, v : L_\alpha(A)$ and therefore establish the above typing for the map function.

We can obtain these hypotheses in a nice way thanks to the new connectives $A \wedge \alpha$ and $A \vee \alpha$. The former means A if α denotes a positive ordinal and the type $\forall X.X$ if it denotes zero. The latter means A if α denotes a positive ordinals and the type $\exists X.X$ otherwise.

Definition 7.1. With these extensions, the grammar for terms, and types is now the following:

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

$$\frac{\gamma \vdash t \in A[\alpha := \kappa] \subset B}{\gamma \vdash t \in \forall \alpha.A \subset B}\ \forall^o_l \qquad \frac{\gamma \vdash t \in A \subset B[\alpha := \epsilon_{\alpha<\infty+\omega}(t \notin B)]}{\gamma \vdash t \in A \subset \forall \alpha.B}\ \forall^o_r$$

$$\frac{\gamma \vdash t \in B \subset A[\alpha := \kappa]}{\gamma \vdash t \in B \subset \exists \alpha.A}\ \exists^o_r \qquad \frac{\gamma \vdash t \in B[X := \epsilon_{\alpha<\infty+\omega}(t \in B)] \subset A}{\gamma \vdash t \in \exists \alpha.B \subset A}\ \exists^o_l$$

$$\frac{\gamma \cup \{\kappa\} \vdash t \in A \subset B}{\gamma \vdash t \in A \wedge \kappa \subset B}\ \wedge_l \qquad \frac{\gamma \vdash t \in A \subset B \qquad \kappa \in \gamma}{\gamma \vdash t \in A \subset B \wedge \kappa}\ \wedge_r$$

$$\frac{\gamma \cup \{\kappa\} \vdash t \in A \subset B}{\gamma \vdash t \in A \subset B \vee \kappa}\ \vee_r \qquad \frac{\gamma \vdash t \in A \subset B \qquad \kappa \in \gamma}{\gamma \vdash t \in A \vee \kappa \subset B}\ \vee_l$$

Figure 9: Extra subtyping rules.

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

The subtyping rules are just extended to handle the new connective using a few rules given in figure 9.

The typing rules given completely in figure 10 changes more. They use a positivity context build thanks to the new connectives. In two rules, we need a syntactic check on the term that we define now:

Definition 7.2. We say that a term is *strongly normal*, denoted $t \downarrow$ if the following holds:
- t is of the form $\varepsilon_{x \in A} u \notin B$ or $\lambda x.u$.
- $t = Cu$ for some $u \downarrow$.
- $t = \{(l_i = u_i)_{i \in I}\}$ for $u_i \downarrow$ (for all $i \in I$).

In the typing rules, we use an abbreviation $A \wedge \gamma_0$ and $A \vee \gamma_0$ where $\gamma_0 = \kappa_1, \dots, \kappa$ is a context. This means $A \wedge \kappa_1 \cdots \wedge \kappa$ and $A \vee \kappa_1 \cdots \vee \kappa$ respectively.

Furthermore typing proof are allowed to be circular to handle recursive programs. To allow circular typing proof, we consider that $\gamma \vdash Yx.t : A$ is a new kind of abstract judgement (we still use the same abstract judgment as before for subtyping, with the extended terms and types). For this kind of judgement, we will only use ordinals $\varepsilon^i_{\vec{\alpha}<C(\vec{\alpha})}\neg(Yx.t : A)$, abbreviated as $\varepsilon^i_{\vec{\alpha}<C(\vec{\alpha})} Yx.t \notin A$. This means that we will only use the I_k and G rules for induction. To make circular proof, we also need to unfold the fixpoint which is the role of the Y typing rule, which is in general applied below application of I_k and G rules.

The figure 11 presents the simplest possible example of a recursive program: the identity function on natural numbers:

$$\begin{aligned} \mathbb{N} &= \mu X.[\text{Z}|\text{S of } X] \\ 0 \ : \mathbb{N} &= Z \\ \text{s} \ : \mathbb{N} \to \mathbb{N} &= \lambda n.Sn \\ \text{id} \ : \mathbb{N} \to \mathbb{N} &= Yid.\lambda n.[n \mid Z \to 0 \mid Sp \to \text{s}\ (id\ p)] \end{aligned}$$

$$\frac{\gamma \vdash \epsilon_{x\in A}(t \notin B) \in A \subset C}{\gamma \vdash \epsilon_{x\in A}(t \notin B) : C}\,\epsilon$$

$$\frac{\gamma \vdash \lambda x\, t \in (A \to B) \vee \gamma_0 \subset C \qquad \gamma, \gamma_0 \vdash t[x := \epsilon_{x\in A}(t \notin B)] : B}{\gamma \vdash \lambda x\, t : C}\,\to_i$$

$$\frac{\gamma \vdash t : (A \to B) \wedge \gamma_0 \qquad \gamma, \gamma_0 \vdash u : A}{\gamma \vdash t\, u : B}\,\to_e \qquad\qquad \frac{\gamma \vdash t : \{l : A; \dots\}}{\gamma \vdash t.l : A}\,\times_e$$

$$\frac{\gamma \vdash \{(l_i = t_i)_{i\in I}\} \in \{(l_i : A_i)_{i\in I}\} \vee \gamma_0 \subset B \quad (\gamma, \gamma_0 \vdash t_i : A_i)_{i\in I} \quad \gamma_0 = \emptyset \text{ or } \forall i, t_i \downarrow}{\gamma \vdash \{(l_i = t_i)_{i\in I}\} : B}\,\times_i$$

$$\frac{\gamma \vdash C\, t \in [C \text{ of } A] \vee \gamma_0 \subset B \qquad \gamma, \gamma_0 \vdash t : A \qquad \gamma_0 = \emptyset \text{ or } t \downarrow}{\gamma \vdash C\, t : B}\,+_i$$

$$\frac{\gamma \vdash t : [(C_i \text{ of } A_i)_{i\in I}] \wedge \gamma_0 \qquad (\gamma, \gamma_0 \vdash t_i : A_i \to B)_{i\in I}}{\gamma \vdash [t \mid (C_i \to t_i)_{i\in I}] : B}\,+_e$$

$$\frac{\gamma \vdash t[x := Yx.t] : A}{\gamma \vdash Yx.t : A}\,Y$$

Figure 10: Typing rules for termination.

The proof has only two blocks: B_0 at the bottom (it contains only one rule) and B_1 for the rest of the proof, one edge from B_0 to B_1 labelled with the empty matrix and one call from B_1 to B_1 labelled with the 1×1 matrice (-1) because one can prove $\kappa_0 \vdash \kappa_1 < \kappa_0$.

One must also notive the use of the $\wedge$ connective in the left subproof to collect the information that we can assume $\kappa_0 > 0$ in the right subproof. Without this, we would not have $\kappa_0 \vdash \kappa_1 < \kappa_0$ and the proof would be rejected.

The definition of the semantics is changed because we do not want to consider reduction under binders:

Definition 7.3. We denote $t \succ_W u$ the relation of one step weak reduction, it obey the same rule as $\succ$, but reduction is not closed under any context: We forbid reduction under abstraction and in the right member of a case analysis. We also add the reduction rule $Yx.t \succ_W t[x := Yx.t]$ for fix point. We write as usual $\succ_W^*$ the transitive closure of $\succ_W$.

Definition 7.4. Therefore, we denote by $\mathcal{N}'$ the set of pure terms which are strongly weak normalizing. This means having no infinite sequence of reduction for the relation $\succ_W$.

Definition 7.5. We say that a set of pure terms $\Phi \subseteq [\![\Lambda]\!]$ is *weak saturated* if the following conditions hold.

– Φ is closed by head reduction.
– If $H[t[x := u]] \in \Phi$ and $u \in \mathcal{N}'$ then $H[(\lambda x.t)\; u] \in \Phi$.
– If $H[t\; u] \in \Phi$, then $H[[D\; u \mid D \to t]] \in \Phi$.

$$\begin{prooftree}
\AxiomC{}
\RightLabel{$=$}
\UnaryInfC{$\kappa_0 \vdash n_0 \in \mathrm{F}(\mathbb{N}_{\kappa_1}) \subset \mathrm{F}(\mathbb{N}_{\kappa_1})$}
\RightLabel{$\vee_r$}
\UnaryInfC{$\kappa_0 \vdash n_0 \in \mathrm{F}(\mathbb{N}_{\kappa_1}) \subset \mathrm{F}(\mathbb{N}_{\kappa_1}) \wedge \kappa_0$}
\RightLabel{μ_l}
\UnaryInfC{$\vdash n_0 \in \mathbb{N}_{\kappa_0} \subset \mathrm{F}(\mathbb{N}_{\kappa_1}) \wedge \kappa_0$}
\RightLabel{Ax}
\UnaryInfC{$\vdash n_0 : \mathrm{F}(\mathbb{N}_{\kappa_1}) \wedge \kappa_0$}
\AxiomC{}
\UnaryInfC{$\kappa_0 \vdash 0 : \mathbb{N}$}
\AxiomC{}
\RightLabel{Ax}
\UnaryInfC{$\kappa_0 \vdash p_0 : \mathbb{N}_{\kappa_1}$}
\AxiomC{$[\forall \alpha_0(\vdash \mathrm{id} : \mathbb{N}_{\alpha_0} \to \mathbb{N})]_0$}
\RightLabel{G}
\UnaryInfC{$\kappa_0 \vdash \mathrm{id} : \mathbb{N}_{\kappa_1} \to \mathbb{N}$}
\RightLabel{$\to_e$}
\BinaryInfC{$\kappa_0 \vdash \mathrm{id}\ p_0 : \mathbb{N}$}
\AxiomC{}
\UnaryInfC{$\kappa_0 \vdash \mathrm{s} : \mathbb{N} \to \mathbb{N}$}
\RightLabel{$\to_e$}
\BinaryInfC{$\kappa_0 \vdash \mathrm{s}\,(\mathrm{id}\ p_0) : \mathbb{N}$}
\RightLabel{$\to_i$}
\UnaryInfC{$\kappa_0 \vdash \lambda p.\mathrm{s}\,(\mathrm{id}\ p) : \mathbb{N}_{\kappa_1} \to \mathbb{N}$}
\RightLabel{$+_e$}
\TrinaryInfC{$\vdash [n_0 \mid Z \to 0 \mid Sp \to \mathrm{s}\,(\mathrm{id}\ p)] : \mathbb{N}$}
\RightLabel{$\to_i$}
\UnaryInfC{$\vdash \lambda n.[n \mid Z \to 0 \mid Sp \to \mathrm{s}\,(\mathrm{id}\ p)] : \mathbb{N}_{\kappa_0} \to \mathbb{N}$}
\RightLabel{Y}
\UnaryInfC{$\vdash \mathrm{id} : \mathbb{N}_{\kappa_0} \to \mathbb{N}$}
\RightLabel{I_0}
\UnaryInfC{$\forall \alpha_0(\vdash \mathrm{id} : \mathbb{N}_{\alpha_0} \to \mathbb{N})$}
\RightLabel{G}
\UnaryInfC{$\vdash \mathrm{id} : \mathbb{N} \to \mathbb{N}$}
\end{prooftree}$$

where

$$\begin{aligned}
\mathrm{F}(X) &= [\mathrm{Z}|\mathrm{S}\text{ of }X] & n_0 &= \varepsilon_{n \in \mathbb{N}_{\kappa_0}}([n \mid Z \to 0 \mid Sp \to \mathrm{s}\,(\mathrm{id}\ p)] \notin \mathbb{N}) \\
\mathbb{N}_\alpha &= \mu_\alpha X.\mathrm{F}(X) & p_0 &= \varepsilon_{p \in \mathbb{N}_{\kappa_1}}(\mathrm{s}\,(\mathrm{id}\ p) \notin \mathbb{N}) \\
\kappa_0 &= \varepsilon^1_{\alpha_1 < \infty + \omega}(\mathrm{id} \notin \mathbb{N}_{\alpha_1} \to \mathbb{N}) \\
\kappa_1 &= \varepsilon_{\alpha < \kappa_0}(n_0 \in \mathrm{F}(\mathbb{N}_\alpha))
\end{aligned}$$

Figure 11: Typing proof of recursive identity on $\mathbb{N}$.

– If $H[t \mid (C_i \to t_i)_{i \in I}] \in \Phi$ and $I \subset J$, and for all $j \in J \quad I$, $t_j \in \mathcal{N}'$, then
$$H[t \mid (C_i \to t_i)_{i \in J}] \in \Phi.$$
– If $H[t] \in \Phi$ and for all $i \in I$ we have $t_i \in \mathcal{N}'$, then
$$H[\{l = t; (l_i = t_i)_{i \in I}\}.l] \in \Phi.$$

This is the same definition as 6.5 with $\mathcal{N}$ replaced by $\mathcal{N}'$.

Definition 7.6. We define $\mathcal{N}'_0$ as before, using $\mathcal{N}'$ insead of $\mathcal{N}_0$, it is the smallest set 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 $[t \mid (C_i \to t_i)_{i \in I}] \in \mathcal{N}'_0$.

We denote by $\overline{\mathcal{N}}'_0$ the least weak saturated set containing $\mathcal{N}'_0$.

For instance, the term $x(Y\lambda r.\lambda x.r)$ is in $\mathcal{N}'_0$, but not in $\mathcal{N}_0$.

With these definitions, the lemma 6.6 to 6.9 and lemma 6.13 are preserved, mainly because these lemmas never consider reductions which are allowed for $\succ$ and forbidden for $\succ_W$. We now give the adaptation of lemma 6.10.

Lemma 7.7. *The set $\mathcal{N}'$ is weak saturated.*

Proof. The proof is unchanged except the first case which now needs the following lemma which was immediate in section 6. Indeed, in this case, we now have to prove that if $H[t[x := u]] \in \mathcal{N}'$ and $u \in \mathcal{N}'$, then $H[(\lambda x.t)u] \in \mathcal{N}'$. We consider an infinite weak reduction of $H[(\lambda x.t)u]$ which must start as $H[(\lambda x.t)u] \succ^*_W H'[(\lambda x.t)u'] \succ^*_W H'[t[x := u']] \succ^*_W \ldots$. This reduction can be transformed into $H[t[x := u]] \succ^*_W H'[t[x := u]]$ and we can apply the lemma 7.8 below to get an infinite reduction of $H'[t[x := u]]$ from the infinite reduction of $H'[t[x := u']]$. $\square$

Lemma 7.8. *For any terms $t \in \Lambda$ and $u \in \mathcal{N}'$ such that $u \succ_W^* u'$, if $t[x := u'] \in \mathcal{N}'$ has an infinite weak reduction then $t[x := u]$ has an infinite weak reduction too.*

Proof. We prove this by co-induction. We first distinguish the occurrences of x in t that are under an abstraction or on the right of a case analysis (blocked positions, using x_0) and the others (using x_1). So we can write $t[x := u] = t_0[x_0 := u, x_1 := u] = (t_0[x_1 := u])[x_0 := u]$ and $t[x := u'] = (t_0[x_1 := u'])[x_0 := u']$.

We consider the first step of an infinite reduction of $t[x := u'] \succ_W t_1[x_0 := u']$ with $t_0[x_1 := u'] \succ_W t_1$ (there can not be any reduction for the occurences of u' that replace x_0 because x_0 is always in blocked position). We have $t[x := u] = (t_0[x_1 := u])[x_0 := u] \succ_W^* = (t_0[x_1 := u'])[x_0 := u] \succ_W t_1[x_0 := u]$.

This step being productive, we can apply the co-induction hypothesis with t_1 to get an infinite weak reduction of $t_1[x_0 := u]$ from the infinite weak reduction of $t_1[x_0 := u']$. □

Definition 7.9. We can thus modify and extend the definition of the interpretation of formulas, replacing $\mathcal{N}$ and $\mathcal{N}_0$ by $\mathcal{N}'$ and $\mathcal{N}_0'$ respectively, everywhere and adding these four cases. This means that we now have:

$$[\![\mathcal{F}]\!] = \{\Phi \subseteq [\![\Lambda]\!] \mid \Phi \text{ saturated, } \mathcal{N}_0' \subseteq \Phi \subseteq \mathcal{N}'\}$$

- $[\![\forall \alpha.A]\!] = \cap_{o \in [\![\mathcal{O}]\!]} [\![A[\alpha := o]]\!]$
- $[\![\exists \alpha.A]\!] = \cup_{o \in [\![\mathcal{O}]\!]} [\![A[\alpha := o]]\!]$
- $[\![A \wedge \kappa]\!] = [\![A]\!]$ if $[\![\kappa]\!] > 0$ and $[\![A \wedge \kappa]\!] = \overline{\mathcal{N}_0}'$ if $[\![\kappa]\!] = 0$
- $[\![A \vee \kappa]\!] = [\![A]\!]$ if $[\![\kappa]\!] > 0$ and $[\![A \vee \kappa]\!] = \mathcal{N}'$ if $[\![\kappa]\!] = 0$

Theorem 7.10. *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. Same proof as 6.21, the four new cases being immediate by induction. □

Lemma 7.11. *If $t \in \Lambda$ is strongly normal ($t \downarrow$) then $[\![t]\!] \in \mathcal{N}'$.*

Proof. Immediate by induction. We use theorem 7.10 when $t = \varepsilon_{x \in A} u \notin B$. □

Lemma 7.12. *$t \in [\![[(C_i \text{ of } A_i)_{i \in I}]]\!]$ if and only if $t \in \mathcal{N}'$ and t reduces by head reduction to a term in $\mathcal{N}_0'$ or to $C_i u$ with $i \in I$ and $u \in [\![A_i]\!]$.*

Lemma 7.13. *$t \in [\![\{(l_i : A_i)_{i \in I}; ...\}]\!]$ (resp. $[\![\{(l_i : A_i)_{i \in I}; ...\}]\!]$) if and only if $t \in \mathcal{N}'$ and t reduces by head reduction to a term in $\mathcal{N}_0'$ or to $\{(l_i = u_i)_{i \in J}$ with $u_i \in [\![A_i]\!]$ for all $i \in I$ and $I \subset J$ (resp. $I = J$).*

Proof. The same proof as 6.19 and 6.20 works, replacing $\mathcal{N}$ and $\mathcal{N}_0$ with $\mathcal{N}'$ and $\overline{\mathcal{N}_0}'$. When using the $+_i$ and $\times_i$ rules we take $\gamma_0 = \emptyset$. Indeed, we do not need positivity context as the proof we build is not circular. □

Lemma 7.14. *is still valid and moreover can be extended to subsitution of ordinal variables by syntactic ordinals.*

Proof. By induction on the size of the term, ordinal or type in which we perform the substitution. □

Theorem 7.15. *The adequacy lemma still holds*

Proof. Again the same proof as in section 6 works, replacing $\mathcal{N}$ and $\mathcal{N}_0$ with $\mathcal{N}'$ and $\overline{\mathcal{N}_0}'$, except for the typing rules that we changed (see figure 10) and the new subtyping rules. Here are the proofs only for these cases:

- $\wedge_l$ We assume that $[\![t]\!] \in [\![A \wedge \kappa]\!]$. If $[\![\kappa]\!] = 0$, then $[\![A \wedge \kappa]\!] = \overline{\mathcal{N}_0}'$ and there $[\![t]\!] \in B$. If $[\![\kappa]\!] > 0$, then $[\![A \wedge \kappa]\!] = [\![A]\!]$ and the induction hypothesis allows to conclude.
- $\wedge_r$ Because $\kappa \in \gamma$, we can assume $[\![\kappa]\!] > 0$, hence $[\![A \wedge \kappa]\!] = [\![A]\!]$, and the result follows from the induction hypothesis.
- $\vee_r$, $\vee_l$: dual to the previous cases.
- $\forall_l^o$ If $[\![t]\!] \in [\![\forall\alpha.A]\!]$, then $[\![t]\!] \in [\![A[\alpha := [\![\kappa]\!]]]\!] = [\![A[\alpha := \kappa]]\!]$ by lemma 7.14. Then the induction hypothesis gives $[\![t]\!] \in [\![B]\!]$.
- $\exists_r^o$ similar to the previous case.
- $\forall_r^o$ By contraposition, it $[\![t]\!] \notin [\![\forall\alpha.B]\!]$, then there must an ordinal $o \in [\![\mathcal{O}]\!]$ such that $[\![t]\!] \notin [\![B[x := o]]\!]$. This means that for $o = [\![\varepsilon_{\alpha<\infty+\omega}(t \notin B)]\!]$, we also have $[\![t]\!] \notin [\![B[x := o]]\!]$. Then, by induction hypothesis, we deduce $[\![t]\!] \notin [\![A]\!]$.
- $\exists_l^o$ similar to the previous case, without using contraposition.
- $\Rightarrow_i$ We need to show that $[\![\lambda xt]\!] \in [\![C]\!]$. By induction hypothesis, we have $[\![\lambda xt]\!] \in [\![(A \Rightarrow B) \vee \gamma_0]\!]$ implies $[\![\lambda xt]\!] \in [\![C]\!]$. If for some syntactic ordinal $\kappa \in \gamma_0$ we have $[\![\kappa]\!] = 0$, we have $[\![\lambda xt]\!] \in [\![(A \Rightarrow B) \vee \gamma_0]\!] = \mathcal{N}'$ which is always true. Therefore, we have $[\![\lambda xt]\!] \in [\![C]\!]$.

  We can now assume that $\kappa \in \gamma_0$ implies $[\![\kappa]\!] > 0$, which means that the positivity context of the right premise is valid. It remains to prove that in this case too, we have $[\![\lambda xt]\!] \in [\![A \Rightarrow B]\!]$. By induction hypothesis on the right premise, we have $[\![t[x := \varepsilon_{x\in A}(t \notin B)]]\!] \in [\![B]\!]$ which by definition of the choice operator means that for all $u \in [\![A]\!]$ we have $[\![t[x := u]]\!] \in [\![B]\!]$. Hence $[\![\lambda xt]\!] \in [\![A \Rightarrow B]\!]$ by saturation.
- $\Rightarrow_e$ By induction hypothesis, we have $[\![t]\!] \in [\![(A \to B) \wedge \gamma_0]\!]$. If for some syntactic ordinal $\kappa \in \gamma_0$ we have $[\![\kappa]\!] = 0$, we have $[\![t]\!] \in \overline{\mathcal{N}_0}'$ which implies $[\![t\,u]\!] \in \overline{\mathcal{N}_0}' \subset [\![B]\!]$.

  If for all syntactic ordinal $\kappa \in \gamma_0$ we have $[\![\kappa]\!] > 0$, we have $[\![t]\!] \in [\![A \to B]\!]$ and we can use the induction hypothesis on the right premise to get $[\![u]\!] \in [\![A]\!]$. The definition of $[\![A \Rightarrow B]\!]$ allows us to conclude.
- $+_i$ It t is strongly normal and for some syntactic ordinal $\kappa \in \gamma_0$ we have $[\![\kappa]\!] = 0$, Then $[\![Ct]\!] \in \mathcal{N}'$. In this case, the left premise means that $[\![Ct]\!] \in [\![[C \text{ of } A] \vee \gamma_0]\!] = \mathcal{N}'$, implying $[\![Ct]\!] \in [\![B]\!]$.

  Otherwise, we must have $\gamma_0 = \emptyset$ or $[\![\kappa]\!] > 0$ for all syntactic ordinal $\kappa \in \gamma_0$. Therefore, by induction, we have $[\![t]\!] \in [\![A]\!]$. From this we get $[\![Ct]\!] \in [\![[C \text{ of } A]]\!]$ by saturation. The correctness of the subtyping gives $[\![Ct]\!] \in [\![B]\!]$.
- $+_e$ If for some syntactic ordinal $\kappa \in \gamma_0$ we have $[\![\kappa]\!] = 0$, the left premise give $[\![t]\!] \in \overline{\mathcal{N}_0}'$ thus we have $[\![[t|(C_i \to t_i)_{i\in I}]]\!] \in \overline{\mathcal{N}_0}' \subset [\![B]\!]$. Otherwise, the right premises can be used and the result is immediate by induction and definition of $[\![[(C_i : A_i)_{i\in I}]]\!]$.
- $\times_i$ If for some syntactic ordinal $\kappa \in \gamma_0$ we have $[\![\kappa]\!] = 0$ and $t_i\downarrow$, then the left premise is enough to conclude because $[\![\{(l_i = t_i)_{i\in I}\}]\!] \in [\![\{(l_i : A_i)_{i\in I}\} \vee \gamma_0]\!] = \mathcal{N}'$ implies $[\![\{(l_i = t_i)_{i\in I}\}]\!] \in [\![B]\!]$ using only the left premise.

  Otherwise, we can use the right premise and by induction, we have $[\![t_i]\!] \in [\![A_i]\!]$ for all $i \in I$. From this we get $[\![\{(l_i = t_i)_{i\in I}\}]\!] \in [\![\{(l_i : A_i)_{i\in I}\}]\!]$ by saturation. Then the correctness of the subtyping gives $[\![\{(l_i : A_i)_{i\in I}\}]\!] \in [\![B]\!]$ and we can conclude.

$\times_e$ Immediate by induction hypothesis and by definition of $[\![\{l : A; \dots\}]\!]$

Y by definition, we have $(Yx.t) \succ_W t[x := Yx.t]$, therefore the rule is valid by saturation of $[\![A]\!]$.

We also have to use the result of section 3, now both for the subtyping rules and the typing rules because we can use the I_k and G rules in typing. $\square$

Theorem 7.16. *As for the initial system, we get termination (typed terms are strongly weakly normalizable), type safety for simple data types and consistency.*

Proof. These result are the analogous of theorem 6.23, 6.25 and 6.26 and the same proofs work, using the previous results in this section. $\square$

8. Terminating examples

The classical inductive functions on unary natural works more or less like the example of identity we gave in the previous section. More interesting examples arise when using lists.

$$
\begin{aligned}
\mathrm{F}(A, X) &= [\mathrm{Nil} | \mathrm{Cons\ of}\ \{\mathrm{hd} : A; \mathrm{tl} : X\}] \\
\mathbb{L}_\alpha(A) &= \mu_\alpha X.\mathrm{F}(A, X) \\
\mathbb{L}(A) &= \mu X.\mathrm{F}(A, X) \\
\mathrm{map} &: \forall A.\forall B.\forall \alpha.(A \to B) \to \mathbb{L}_\alpha(A) \to \mathbb{L}_\alpha(B) \\
&= Y map.\lambda f\, l.\left[l \left| \begin{array}{l} [] \to [] \\ x :: l \to f\ x{::}map\ f\ l \end{array} \right. \right] \\
\mathrm{map}_2 &: \forall A.\forall B.\forall C.\forall \alpha.(A \to B \to C) \to \mathbb{L}_\alpha(A) \to \mathbb{L}_\alpha(B) \to \mathbb{L}_\alpha(C) \\
&= Y map_2.\lambda f\, l_1\, l_2.\left[l_1 \left| \begin{array}{l} [] \to [] \\ x :: l_1 \to \left[l_2 \left| \begin{array}{l} [] \to [] \\ y :: l_2 \to f\ x\ y{::}map_2\ f\ l_1\ l_2 \end{array} \right. \right] \end{array} \right. \right] \\
\mathrm{flatten} &: \forall A.\mathbb{L}(\mathbb{L}(A)) \to \mathbb{L}(A) \\
&= Y flatten.\lambda l_s.\left[l_s \left| \begin{array}{l} [] \to [] \\ l :: l_s \to \left[l \left| \begin{array}{l} [] \to flatten\ l_s \\ x :: l \to x{::}flatten\ (l{::}l_s) \end{array} \right. \right] \end{array} \right. \right]
\end{aligned}
$$

The type of lists of *size* α is straight forward. With this type we can define the traditionnal map function on list with the information that it preserves the size of lists.

More surprisingly, functions such as map_2 are also typable with some size information. However, the type given here is not enough because it forbids to use map_2 on two lists of unraleted sizes and still keep some size information about the result. A better type for map_2 would require adding a min function symbol on syntactic ordinals to give an accurate information about the size of the result.

It is important to notice that the type for map and map_2 are subtypes of the type carrying no size information (i.e. $\forall A.\forall B.\forall \alpha.(A \to B) \to \mathbb{L}_\alpha(A) \to \mathbb{L}_\alpha(B) \subset \forall A.\forall B.(A \to B) \to \mathbb{L}(A) \to \mathbb{L}(B)$). It is therefore not necessary to provide two versions of each function.

The flatten example illustrates the possibility of unrolling the fixpoint more that once. This program is not the natural way to write this function, but it is an interesting example because it requires, in the induction, the size of the first element of the list.

When we unroll the fixpoint only once, our algorithm infer the type $\forall A.\forall\alpha.\forall\beta.\mathbb{L}_\alpha(\mathbb{L}_\beta(A)) \rightarrow \mathbb{L}(A)$ which is not sufficient for inferring termination.

However, if we unroll the second recursive call of flatten, keeping two induction hypotheses, our algorithm succeed in proving the termination. This is equivalent to typing the following program:

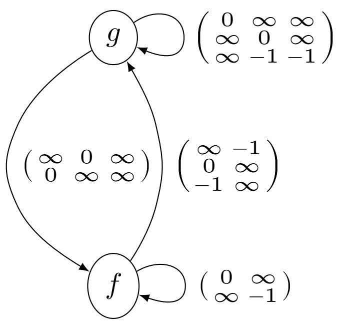
$$Yf.\lambda l_s.\left[l_s \,\middle|\, \begin{array}{l} [] \rightarrow [] \\ l :: l_s \rightarrow \left[l \,\middle|\, \begin{array}{l} [] \rightarrow f\ l_s \\ x :: l \rightarrow x{::}Yg.\lambda l_s.\left[l_s \,\middle|\, \begin{array}{l} [] \rightarrow [] \\ l :: l_s \rightarrow \left[l \,\middle|\, \begin{array}{l} [] \rightarrow f\ l_s \\ x :: l \rightarrow x{::}g\ (l{::}l_s) \end{array}\right] \end{array}\right]\ (l{::}l_s) \end{array}\right] \end{array}\right]$$

where

– f has schema $\forall\alpha_0, \alpha_1(\vdash \mathrm{f} : \forall A.\mathbb{L}_{\alpha_1}(\mathbb{L}_{\alpha_0}(A)) \rightarrow \mathbb{L}(A))$ and

– g has $\forall\beta_0, \beta_1, \beta_2, \beta_2 < \beta_1 \Rightarrow (\beta_1 \vdash \mathrm{g} : F(\mathbb{L}_{\beta_2}(A_0), \mathbb{L}_{\beta_0}(\mathbb{L}_{\beta_1}(A_0))) \rightarrow \mathbb{L}(A_0))$

This leads to the following call graph where the order of the ordinal quantifiers corresponds to the indexes of the lines and columns of the matrices:



$$\begin{tikzcd}[row sep=huge]
g \arrow[loop right, "\begin{pmatrix} 0 & \infty & \infty \\ \infty & 0 & \infty \\ \infty & -1 & -1 \end{pmatrix}"] \arrow[d, bend right, "\begin{pmatrix} \infty & 0 & \infty \\ 0 & \infty & \infty \end{pmatrix}"'] \\
f \arrow[u, bend right, "\begin{pmatrix} \infty & -1 \\ 0 & \infty \\ -1 & \infty \end{pmatrix}"'] \arrow[loop right, "\begin{pmatrix} 0 & \infty \\ \infty & -1 \end{pmatrix}"]
\end{tikzcd}$$

We explain now each matrix. You should be aware that the above schemas and call graph are directly produced by the application and as a consequence, the name for the ordinal variables are not optimal:

– The loop on f corresponds to the first recursive call and the 2×2 matrice is justified because in this call α_0, the size of the inner list type, is constant and α_1 decreases.

– The edge from f to g represents the definition of g inside f. In this call, the first line of the matrix is justified by $\beta_0 < \alpha_1$ because β_0 does not count the first element. The second line is justified because $\beta_1 = \alpha_0$ the size of the inner list. The last line is justified by $\beta_2 < \alpha_0$ as one element is removed from the first.

– The loop on g corresponds to the last récursive call, where β_0 and β_1 are constant, justifying the first two lines. The first element of the list is decreasing, so β_2 decreases. Moreover, as we keep in the schema the information that $\beta_2 < \beta_1$, we also have a -1 in the middle of the last line.

– Finally, the edge for g to f corresponds to the third recursive call. And we do have $\alpha_0 = \beta_1$, $\alpha_1 = \beta_0$ and α_2 become useless.

We can also prove termination and preservation of size for the insertion sort.

$\mathrm{F}(A, X) = [\mathrm{Nil}|\mathrm{Cons\ of\ } \{\mathrm{hd} : A; \mathrm{tl} : X\}]$

$$\begin{aligned}
\text{insert} &: \forall\alpha.\forall A.(A \to A \to \mathbb{B}) \to A \to \mathbb{L}_{\alpha}(A) \to \mathbb{L}_{\alpha+1}(A) \\
&= Y insert.\lambda f\, a\, l. \left[l \left| \begin{array}{l} [] \to a{::}[] \\ x :: l \to [f\ a\ x \mid T \to a{::}l \mid F \to x{::}insert\ f\ a\ l] \end{array} \right. \right] \\
\text{sort} &: \forall\alpha.\forall A.(A \to A \to \mathbb{B}) \to \mathbb{L}_{\alpha}(A) \to \mathbb{L}_{\alpha}(A) \\
&= Y sort.\lambda f\, l. \left[l \left| \begin{array}{l} [] \to [] \\ x :: l \to \text{insert } f\ x\ (sort\ f\ l) \end{array} \right. \right]
\end{aligned}$$

We also tested the quick sort and merge sort and in both case managed to prove termination, but not size preservation. Proving size preservation of such program would require the addition function on ordinals.

We now illustrate subtyping with the simple example of append lists.

$$\begin{aligned}
\text{AList}(A) &= \mu X.[\text{Nil}|\text{Cons of } \{\text{hd} : A; \text{tl} : X\}|\text{App of } \{\text{left} : X; \text{right} : X\}] \\
\text{fromList} &: \forall X.\text{List}(X) \to \text{AList}(X) \\
&= \lambda l.l \\
\text{toList} &: \forall X.\text{AList}(X) \to \text{List}(X) \\
&= Y toList.\lambda l. \left[l \left| \begin{array}{l} [] \to [] \\ e :: l \to e{::}toList\ l \\ App\{left = l; right = r\} \to \text{append } (toList\ l)\ (toList\ r) \end{array} \right. \right]
\end{aligned}$$

The conversion from append lists to lists in simply the identity function thanks to subtyping.

We now give an example mixing inductive and co-inductive types. We define the type of streams and the type of filter on streams.

$$\begin{aligned}
\mathbb{S}(A) &= \nu X.(\{\} \to A \times X) \\
\mathbb{F} &= \nu X.\mu Y.(\{\} \to [\text{R of } Y|\text{K of } X])
\end{aligned}$$

In the second type, the type of filters, the variant R means *remove* one element of the stream and the variant K means *keep* one. We give the function applying a filter to a stream and the function composing filters:

$$\begin{aligned}
\text{filter} &: \forall A.\mathbb{F} \to \mathbb{S}(A) \to \mathbb{S}(A) \\
&= Y filter.\lambda f\, s.(\lambda(h,t). \left[f\ () \left| \begin{array}{l} Rf' \to filter\, f'\ t \\ Kf' \to \lambda u.(h, filter\, f'\ t) \end{array} \right. \right])\ (s\ ()) \\
\text{cmp} &: \mathbb{F} \to \mathbb{F} \to \mathbb{F} \\
&= Y cmp.\lambda f_1\, f_2\, u. \left[f_2\ () \left| \begin{array}{l} Kf_2' \to \left[f_1\ () \left| \begin{array}{l} Kf_1' \to K(cmp\, f_1'\ f_2') \\ Rf_1' \to R(cmp\, f_1'\ f_2') \end{array} \right. \right] \\ Rf_2' \to R(cmp\, f_1\ f_2') \end{array} \right. \right]
\end{aligned}$$

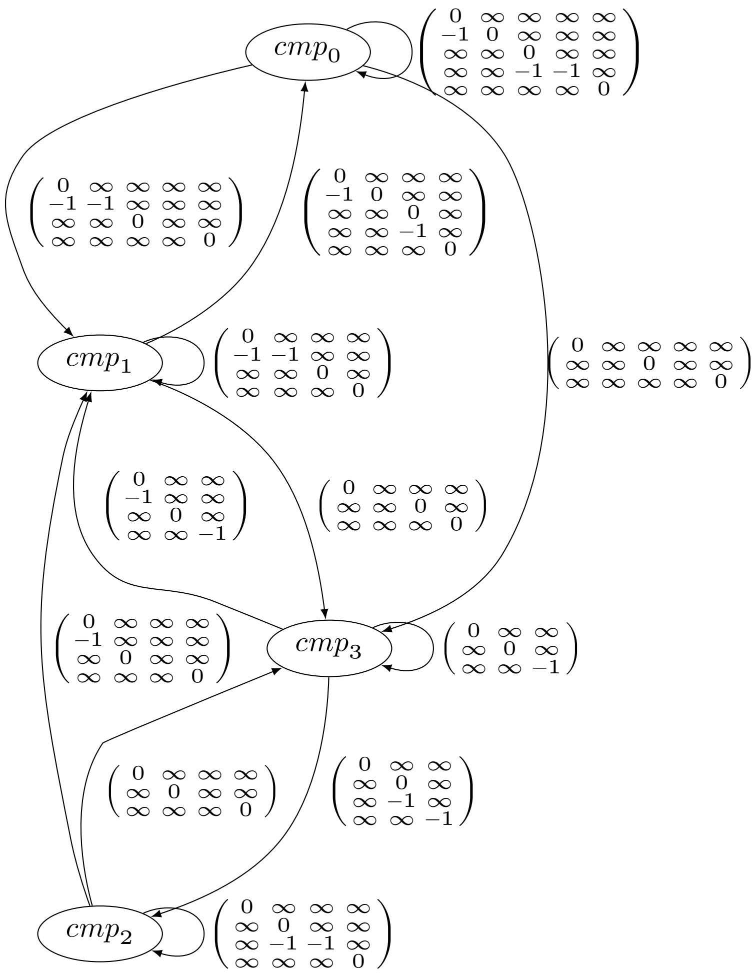

cmp_0
$\begin{pmatrix} 0 & \infty & \infty & \infty & \infty \\ -1 & 0 & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & -1 & -1 & \infty \\ \infty & \infty & \infty & \infty & 0 \end{pmatrix}$
$\begin{pmatrix} 0 & \infty & \infty & \infty & \infty \\ -1 & -1 & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \end{pmatrix}$
$\begin{pmatrix} 0 & \infty & \infty & \infty \\ -1 & 0 & \infty & \infty \\ \infty & \infty & 0 & \infty \\ \infty & \infty & -1 & \infty \\ \infty & \infty & \infty & 0 \end{pmatrix}$
cmp_1
$\begin{pmatrix} 0 & \infty & \infty & \infty \\ -1 & -1 & \infty & \infty \\ \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 0 \end{pmatrix}$
$\begin{pmatrix} 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \end{pmatrix}$
$\begin{pmatrix} 0 & \infty & \infty \\ -1 & \infty & \infty \\ \infty & 0 & \infty \\ \infty & \infty & -1 \end{pmatrix}$
$\begin{pmatrix} 0 & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 0 \end{pmatrix}$
$\begin{pmatrix} 0 & \infty & \infty & \infty \\ -1 & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & 0 \end{pmatrix}$
cmp_3
$\begin{pmatrix} 0 & \infty & \infty \\ \infty & 0 & \infty \\ \infty & \infty & -1 \end{pmatrix}$
$\begin{pmatrix} 0 & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & 0 \end{pmatrix}$
$\begin{pmatrix} 0 & \infty & \infty \\ \infty & 0 & \infty \\ \infty & -1 & \infty \\ \infty & \infty & -1 \end{pmatrix}$
cmp_2
$\begin{pmatrix} 0 & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty \\ \infty & -1 & -1 & \infty \\ \infty & \infty & \infty & 0 \end{pmatrix}$

Figure 12: Call-graph of the cmp function

These functions, as in the previous example, require to unroll the fixpoint, twice for filter and three times for cmp. To avoid the need to unroll the fixpoint we may replace the type $\mathbb{F}$ with the type $\mathbb{F}' = \mu Y.(\{\} \to [\mathrm{R\ of\ } Y | \mathrm{K\ of\ } \mathbb{F}])$, which is equal to $\mathbb{F}$ (i.e. subtyping is provable in both directions) but allows for carrying an ordinal representing the initial number of R constructors in the type. The call-graph for cmp is given in figure 12.

Remark: the type $\mathbb{F}$ is isomorphic to the type of stream of natural numbers and we can prove termination of this isomorphism, even keeping the information about the "size" of streams:

$$\begin{aligned} \mathrm{f2s} &: \forall\alpha.\mathbb{F}_\alpha \to \mathbb{S}_\alpha(\mathbb{N}) \\ &= Y f2s.\lambda s\, u.\left[s\ () \,\middle|\, \begin{array}{l} Rs \to (\lambda(n,r).(Sn,r))\ (f2s\ s\ ()) \\ Ks \to (Z, f2s\ s) \end{array} \right] \\ \mathrm{f} &: \forall\alpha.\mathbb{F}_\alpha \to \mathbb{N} \to \mathbb{F}_{\alpha+1} \end{aligned}$$

$$
\begin{aligned}
&= Yf.\lambda s\, n.\left[n\,\middle|\begin{array}{l} Z \to \lambda u.Ks \\ Sp \to \lambda u.R(f\ s\ p)\end{array}\right] \\
\text{s2f}\ &: \forall\alpha.\mathbb{S}_\alpha(\mathbb{N}) \to \mathbb{F}_\alpha \\
&= Ys2f.\lambda s\, u.(\lambda(n,s).\text{f}\ (s2f\ s)\ n\ ())\ (s\ ())
\end{aligned}
$$

In the current version of the prototype, such functions are not easy to write, mainly because the error messages of the system are not yet clear enough. Is is not easy to see if we have a real error or if (and how) we should help the algorithm with type annotations. This occurs mainly on examples mixing inductive and co-inductive types which are (not yet?) used in prectice, often because simpler representation of the type can be found (like in the above example).

9. 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 each form of terms and one subtyping rule for each for of types (up to commuting rules).

Nonetheless, several subtle details need further discussion. We try to give brief guidelines explaining our implementation. We encourage the reader to look at the code of our prototype [21] that we tried to make accessible.

Unification variables. During type-checking, a term with an unknown type may arise (e.g. the type of the argument of an application or in the subtyping rules for quantifiers). To handle such situations, types are extended with a set of unification variables $\{U, V \dots\}$. Such variables can be used in place of unknown types until their value can be inferred. In our implementation [21], unification variables are handled as follows.

- If we encounter $\vdash t \in U \subset U$ then we use reflexivity.
- If we encounter $\vdash t \in U \subset V$, then we may decide that U and V are equal.
- If we encounter $\vdash t \in U \subset A$ or $\vdash t \in A \subset U$, 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 allowing the system to infer recursive types.

This is a bit too simple, if we have a projection $t.l$ and the type of t is a unification variable, we would set it automatically with a record type with only one field. The same problem arise when we typing the construction of a variant. We solved these issues by carrying a state in unification variables that keep tracks of the type of projected field or constructed variants. The state of a unification variable is created or updated when we encounter $\vdash t \in U \subset \{l_1 : A_1, \dots, l_n : A_n\}$ or $\vdash t \in [C_1 \text{ of } A_1, \dots, C_n \text{ of } A_n] \subset U$. This state can be seens as a subtyping constraint (upper bound for record types, lower bound for variant types) which is delayed until we have a subtyping constraint on the other side.

We also need unification variables for ordinals (in the μ_r, ν_l, $\forall^o_l$ and $\exists^o_r$ rules) and as in the case of types, we keep constraints. An ordinal unification variable O may carry constraints like $\tau \leq O < \kappa$.

In the current implementation, we do some backtracking in only one case. When we want to prove $\gamma \vdash t \in A \subset \mu_O F$ and O is a unification variable, then we try to instanciate it with each of the variable in γ, satisfying the constraint on O. We do this because

– We must fail if there is no positive solution for O and
– we need to create a new unification variable $O' < O$, and if O is not instanciated, we can create long chains of unification variables and make subtyping loop.

Induction in subtyping. The induction rule for subtyping 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 one of the type starts with μ or ν. As a consequence, we do two things:

– We generalise the current subtyping goal into a schema, by quantifying all ordinals appearing in the type, except those occurring only under ε-choice operators. This ensures that only a finite number of schema can be produced when applying the typing rules, ensuring the termination of our algorithm when there are no type quantifiers.
– We store all these produced schemas and before applying a subtyping rule, we check if the current sequent is not a consequence of an assumed schema. The size change principle will tell us if the proof is well-founded.

Induction in typing. It follows the same principle, we create a schema when typing a fixpoint Yt. The generalisation we apply here is a bit more subtle: is the type of Yt does not contain any explicit quantifier on ordinals, we generalise infinite ordinals decorating negative occurrences of μ or positive occurences of ν.

For instance, the sequent $\vdash Yt : \mu XA \to \nu Y \mu ZB$ is generalized to $\forall\alpha\forall\beta \vdash Yt : \mu_\alpha XA \to \nu_\beta Y \mu ZB$.

However, if the type uses ordinal quantifiers, we do not generalize infinite ordinals and only do the same generalisation as for subtyping (actually the code is shared between both form of generalisations).

Breath-first for typing fixpoint. As explained in the previous section, unrolling the fixpoint more than once is often usefull. This requires a breath-first proof search. This means that when typing Yt, we first finish all other branches of the proof, collecting as much as possible information about the type of Yt by having more chances to instantiate unification variable.

This means that each step of the breath-first typing is

– Try to apply the typing rules on the term except for fixpoint. In this case, we only store the current sequent.
– For all these fixpoints, we try first to apply a possible induction hypothesis (note: they are no induction hypothesis at the first stage of the breath-first search).
– For the remaining fixpoints, we perform a generalisation as explained before and store this generalisation as an induction hypothesis.
– The next stage of breath-first can be launched, it consists in proving all these generalised sequent, starting by applying a I rule.

Generalisation and unification variables. To keep unification variables in schema often leads to failure (or even non termination of the type-checking in the case of subtyping). Therefore, unification variables with constraints are instantiated whenever this is possible, using their own constraints.

This means that we fix the set of variant constructor or record field to the current state for a type unification variable or decide that the lower bound for an ordinal unification variable is its value.

Nontheless, unification variables without constraint are still kept in schema. In this case, we need to introduce second order unification variables that may depend upon the generalised ordinals, otherwise the unification variables would not be able to use the quantified ordinals introduced by the generalisation.

For instance, if in the sequent $\vdash Yt : \mu X.A \to \nu Y.\mu Z.B$ a unification variable U occurs in A or B, we introduce a second order unification variable V with two ordinal parameters. The generalized schema is $\forall\alpha\forall\beta \vdash Yt : (\mu_\alpha X.A \to \nu_\beta Y.\mu Z.B)[U := V(\alpha,\beta)]$ and U it self is set to $V(\infty,\infty)$ is the remaining sequents to solve.

We deal with second order unification in a very simple way, using projection when possible and immitation (i.e. constant value) only when projection is not possible. This means that if we need to solve a constraint $\vdash V(\tau,\kappa) \leq \tau$, we will only try to set V to the first projection.

This strategy seems complete enough to handle a lot of cases. However, it still leads of to non termination of type-checking, especially on programs that are complex to type-check or those that contain errors.

10. Perspectives and Future Work

Our experiments show that our framework based on system F, subtyping, circular proofs and choice operators is practical and can be implemented easily. However, a lot of work remains to explore combinations of our system with several common programming features and to transform it into a real programming language.

Higher-order types. In our system, only types and ordinals can be quantified over. We had to introduce second order unification variables and the implementation might be more natural with higher-order types. 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 by Rodolphe Lepigre [20]. To achieve this goal, the system needs to be extended with a first-order layer having terms as individuals. Two new type constructors $t \in A$ (singleton types) and $A \restriction t \equiv u$ (meaning A when t and u are observationnally equal and $\forall X.X$ otherwise) are then required to encode dependent products and program 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 without quantifiers. Our algorithm seems terminating for the fragment without $\forall$ and $\exists$ quantifiers. We are actually able to prove it if we also remove the function type, but a few problems remain when dealing with arrow types, mainly the mere sense of completeness. Various possibilities exist, for instance depending if we want to have $\vdash A \subset ([] \to B)$ for any types A and B (which is not the case currently, but is easy to obtain, with an extra check for arrow type).

A larger complete Subsystem. If we succed in proving the completeness of the fragment without $\forall$ and $\exists$ quantifiers, the next step would be to see if we can gain completeness with some restriction of quantification (like ML style polymorphism).

More generally, the case leading to non-termination of type-checking should be better understood to avoid it as much as possible and try to produce good error messages when the system is interrupted.

References

[1] Martín Abadi, Luca Cardelli, and Gordon Plotkin. Types for the Scott numerals. 1993.

[2] Andreas Abel. *Semi-continuous Sized Types and Termination*, pages 72–88. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[3] Andreas Abel and Brigitte Pientka. Wellfounded recursion with copatterns: a unified approach to termination and productivity. In Greg Morrisett and Tarmo Uustalu, editors, *ICFP Proceedings*, pages 185–196. ACM, 2013.

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

[5] Frédéric Blanqui. Decidability of type-checking in the calculus of algebraic constructions with size annotations. *CoRR*, abs/cs/0608125, 2006.

[6] Frédéric Blanqui and Cody Roux. On the relation between sized-types based termination and semantic labelling. In *Computer Science Logic, 23rd international Workshop, CSL 2009, 18th Annual Conference of the EACSL, Coimbra, Portugal, September 7-11, 2009. Proceedings*, pages 147–162, 2009.

[7] 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.

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

[9] Stephen Dolan and Alan Mycroft. Polymorphism, subtyping and type inference in MLsub. 2015.

[10] Jacques Garrigue. Programming with polymorphic variants. In *ML Workshop*, 1998.

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

[12] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types.* Cambridge University Press, 1989.

[13] Benjamin Grégoire and Jorge Luis Sacchini. On strong normalization of the calculus of constructions with type-based termination. In *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*, pages 333–347, 2010.

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

[15] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In Hans-Juergen Boehm and Guy L. Steele, Jr., editors, *POPL Proceedings.* ACM, 1996.

[16] Frédéric Blanqui (INRIA). Size-bases termination of higher-order rewrite systems. 2017.

[17] Frédéric Blanqui (INRIA) and Colin Riba (INPL). *Combining Typing and Size Constraints for Checking the Termination of Higher-Order Conditional Rewrite Systems*, pages 105–119. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[18] Jean-Louis Krivine. Un algorithme non typable dans le système F. *CRAS*, 304, 1987.

[19] 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.

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

[21] R. Lepigre and C. Raffalli. SubML implementation, 2015.

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

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

[24] Miche Parigot. Un recurseur fortement normalisable et typable pour les entiers de scott. Private communication, 1992.

[25] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

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

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

[28] Christophe Raffalli. The PhoX proof assistant, 2008.

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

[30] Jorge Luis Sacchini. Type-based productivity of stream definitions in the calculus of constructions. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 233–242, 2013.

[31] Jorge Luis Sacchini. Well-founded sized types in the calculus of (co)inductive constructions. 2015.

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

[33] J. B. Wells. Typability and type-checking in the second-order lambda-calculus are equivalent and undecidable. In *LICS Proceedings*, pages 176–185. IEEE Computer Society, 1994.

[34] J.B. Wells. Typability and type checking in system f are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1):111 – 156, 1999.