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 | 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 may carry annotations allowing the encoding of size invariants that are used to ensure the termination of recursive programs. For example, the termination of quicksort can be derived 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 polymorphism and (co-)induction (as for Scott-encoded data types). One of the key ideas is to completely separate the notion of size from recursion. We do not check the termination of programs directly, but rather show that their (circular) typing proofs are well-founded. We then obtain termination using a standard semantic proof of normalisation. To demonstrate the practicality of our system, we provide an implementation which accepts all the examples discussed in the paper.

1. Introduction

Polymorphism and subtyping allow for a more generic programming style. 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. They usually focus on product types like records or modules [28], or on sum types like polymorphic variants [12]. The main reason why subtyping failed to be fully integrated in practical languages like Haskell or OCaml is that it does not mix well with their complex type systems. Moreover, they were not conceived with the aim of supporting a general form of subtyping.

In this paper, we propose a new framework for the construction of type systems with subtyping. Our goal being the design of a practical programming language, we consider a very expressive calculus based on System F. It provides records, polymorphic variants, existential types, inductive types and coinductive types. The latter two carry ordinal numbers which can be used to encode size invariants into the type system [17]. For example we can show that the usual *map* function on lists is size-preserving. The system can be implemented using standard unification techniques thanks to the syntax-directed nature of its typing and subtyping rules (Figures 7 and 8). In particular, only one typing rule applies

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

LOGICAL METHODS
IN COMPUTER SCIENCE

DOI:10.2168/LMCS-???

© R. Lepigre and C. Raffalli
Creative Commons

for each term constructor, and at most one subtyping rule applies for every two type constructors. As a consequence, all of the difficulties are focused in the handling of unification variables and in the construction of circular proofs (see Sections 4 and 10).

Local subtyping and choice operators for terms. To obtain syntax-directed rules, several technical innovations are required. Most notably, a finer notion of subtyping has to be considered: we generalise the usual relation $A \subset B$ using a new *local subtyping* relation $t \in A \subset B$. It is interpreted as “if t has type A then it also has type B”. Usual subtyping is then recovered 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 $A \subset B$ can then be defined as $\varepsilon_{x\in A}(x \notin B) \in A \subset B$. Indeed, $\varepsilon_{x\in A}(x \notin B)$ denotes a counterexample to $A \subset B$, if it exists. Therefore, if we can derive $A \subset B$ then such a counterexample cannot exist, which exactly means that A is a subtype of B in the usual sense.

More generally, choice operators can be used to replace free variables, thus suppressing the need for typing contexts.[2] Intuitively, the term $\varepsilon_{x\in A}(t \notin B)$ denotes a counterexample to the fact that $\lambda x.t$ has type $A \to B$, if it exists. We can thus use this choice operator to build the following unusual typing rule for λ-abstractions.

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

It can be read as a proof by contradiction as its premise is only valid when there is no term u of type A such that $t[x := u]$ does not have type B. Note that this exactly corresponds to the usual realisability interpretation of the arrow type. Thanks to this new typing rule, terms remain closed throughout typing derivations. In particular, the choice operator $\varepsilon_{x\in A}(t \notin B)$ binds the variable x in the term t. As a consequence, the axiom rule is replaced by the following typing rule for choice operators.

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

The other typing rules, including the rule for application given below, are not affected by the introduction of choice operators and they remain usual.

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

Note however that the typing rules of the system (Figure 7) are presented in a slightly more general way. In particular, most of them include a local subtyping judgment.

Choice operators for types. Thanks to local subtyping, the typing rules of the system can be formulated in such a way that connectives without algorithmic contents are only handled in local subtyping judgments (see Figure 7). To manage quantifiers, we introduce two new type constructors $\varepsilon_X(t \in A)$ and $\varepsilon_X(t \notin A)$ corresponding to choice operators satisfying the denoted properties. For example, $\varepsilon_X(t \notin B)$ is interpreted as a type such that t does not have type $B[X := \varepsilon_X(t \notin B)]$. Intuitively, $\varepsilon_X(t \notin B)$ is a counterexample to the fact that t has type $\forall X.B$. Thus, to show that t has type $\forall X.B$, it will be enough

[1] Our model being based on reducibility candidates [13, 14], the interpretation of a type is never empty.

[2] We will still use a form of context to store ordinals assumed to be nonzero (see Section 5).

to show that it has type $B[X := \varepsilon_X(t \notin B)]$. As a consequence, the usual introduction rule for the universal quantifier 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, as there are no free variable thanks to the use of choice operators.

In conjunction with local subtyping, our choice operators for types allow the derivation of valid permutations of quantifiers and connectors. For instance, Mitchell's containment axiom [9] can be easily derived in the system.

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

Another important consequence of these innovations is that our system does not rely on a transitivity rule for local subtyping. In practice, type annotations like $((t : A) : B) : C$ can be used to force 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 find the right instantiation for unification variables. As such annotations are seldom required, we conjecture that a transitivity rule for local subtyping is admissible in the system.

Implicit covariance condition for (co-)inductive types. Inductive and coinductive types are generally handled using types $\mu X.F(X)$ and $\nu X.F(X)$ denoting the least and greatest fixpoint of a covariant parametric type F. In our system, the subtyping rules are so fine-grained that no syntactic covariance condition is required on such types. In fact, the covariance condition is obtained automatically when traversing the types. For instance, if F is not covariant then it will not be possible to derive $\mu X.F(X) \subset \nu X.F(X)$ or $\mu X.F(X) \subset F(\mu X.F(X))$. As far as the authors know, this is the first work in which covariance is not explicitly required for inductive and coinductive types.

Well-founded ordinal induction and size change principle. In this paper, inductive and coinductive types carry an ordinal number κ to form sized types $\mu_\kappa X.F(X)$ and $\nu_\kappa X.F(X)$ [3, 17, 38]. Intuitively, they correspond to κ iterations of F on the types $\bot$ and $\top$ respectively. In particular, if t has type $\mu_\kappa X.F(X)$ then there must be $\tau < \kappa$ such that t has type $F(\mu_\tau X.F(X))$. Dually, if t has type $\nu_\kappa X.F(X)$ then t has type $F(\nu_\tau X.F(X))$ for all $\tau < \kappa$. More precisely, $\mu_\kappa X.F(X)$ is interpreted as the union of all the $F(\mu_\tau X.F(X))$ for $\tau < \kappa$, and $\nu_\kappa X.F(X)$ is interpreted as the intersection of all the $F(\nu_\tau X.F(X))$ for $\tau < \kappa$. These definitions are monotonous in κ, even if F is not covariant. This implies that there exists an ordinal ∞ from which the constructions are stationary. As a consequence, we have $F(\mu_\infty X.F(X)) \subset \mu_\infty X.F(X)$ and $\nu_\infty X.F(X) \subset F(\nu_\infty X.F(X))$, which are sufficient for the correctness of our subtyping rules. In particular, $\mu_\infty X.F(X)$ and $\nu_\infty X.F(X)$ only correspond to the least and greatest fixpoint of F when it is covariant. If F is not covariant, then these stationary points are not fixpoints.

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 [23], which allows us to check for well-foundedness a posteriori. Our system is able to deal with subtyping relations between mixed inductive and coinductive. For example, it is able to derive subtyping relations like $\mu X.\nu Y.F(X, Y) \subset \nu Y.\mu X.F(X, Y)$ 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 completeness in this setting.

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 extending the language with a fixpoint combinator $Yx.t$, reduced using the rule $Yx.t \succ t[x := Yx.t]$. It is handled using the following, very simple typing rule.

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

It is clear that it induces circularity as a proof of $\vdash Yx.t : A$ will require a proof of $\vdash Yx.t : A$. 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, our system is surprisingly powerful. In particular, a fixpoint may be unfolded several times to obtain a well-founded circular proof (see Section 9).

One of the major advantages of our presentation is that it allows for a good integration of the termination check to the type system, both in the theory and in the implementation. Indeed, we do not prove the termination of a program directly, but rather show that its circular typing proof is well-founded. Normalisation is then established indirectly, using a standard semantic proof based on a well-founded induction on the typing derivation. To show that a circular typing proof is well-founded we rely on the size change principle [23]. It is run on size informations that are extracted from the circular structure of our proofs in a precisely defined way (see Section 4).

Quantification over ordinals. As types can carry ordinal sizes, it is natural to allow quantification over the ordinals themselves. We can thus use the following type for the usual *map* function, where $\mathrm{List}(A, \alpha)$ denotes the type of lists of size α with elements of type A (it is defined as $\mu_\alpha L.[\mathrm{Nil} \mid \mathrm{Cons} \text{ of } A \times L]$).

$$\forall A.\forall B.\forall \alpha.(A \to B) \to \mathrm{List}(A, \alpha) \to \mathrm{List}(B, \alpha)$$

Thanks to the quantification on the ordinal α, which links the size of the input list to the size of the output list, we can express the fact that the output is not greater than the input. This means that the system will allow us to make recursive calls through the *map* function, without loosing size information (and thus termination information). This technique also applies to other relevant functions such as insertion sort.

Using size preserving functions and ordinal quantification is important for showing the termination of more complex algorithms. For instance, proving the termination of *quicksort* requires showing that partitioning a list of size α produces two lists of size at most α. To do so, the partitioning function must be defined with the following type.

$$\forall A.\forall \alpha.\mathrm{List}(A, \alpha) \to \mathrm{List}(A, \alpha) \times \mathrm{List}(A, \alpha)$$

It is then possible to define *quicksort* in the usual way, without any other modification. Note that the termination of simple functions is derived automatically by the implementation (i.e., without specific size annotations).

In this paper, the language of the ordinals that can be represented in the syntax is very limited. As in [37], it only contains a constant ∞, a successor symbol and variables for quantification. Working with such a small language allows us to keep things simple while still allowing the encoding of many size invariants. Nonetheless, it is clear that the system could be improved by extending the language of ordinals with function symbols such as, for example, maximum or addition.

Properties of the system. A first version of the language without general recursion (i.e., without the fixpoint combinator) is defined in Section 5. It has three main properties: strong normalisation, type safety and logical consistency (Theorems 7.25, 7.27 and 7.24). These results follow from the construction of a realisability model presented in Section 7. They are consequences of the adequacy lemma (Theorem 7.23), which establishes the compatibility of the model with the language and type system.

After the introduction of the fixpoint combinator in Section 8, the properties of the system are mostly preserved (Theorems 8.17 and 8.18). However, the definition of the model needs to be changed slightly as strong normalisation (in the usual sense) is compromised by the fixpoint combinator. Indeed, the reduction rule $Yx.t \succ t[x := Yx.t]$ is obviously non-terminating. Nonetheless, we can still prove normalisation for all the weak reduction strategies (i.e., those that do not reduce under λ-abstractions).

Implementation. Typing and subtyping are likely to be undecidable in our system. Indeed, it contains Mitchell's variant of System F [9], for which both typing and subtyping are undecidable [42, 43, 44]. Moreover, we believe that there are no practical, complete semi-algorithms for extensions of System F like ours.[3] 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. Indeed, type-checking can only diverge when checking a local subtyping judgment. In this case, a reasonable 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 [24]. Aside from a few subtleties described in Section 10, the implementation is straightforward and remains very close to the typing rules of Figure 12[4] and to the subtyping rules of Figures 8 and 13. Although the system has a great expressive power, its simplicity allows for a very concise implementation. The main functions (type-checking and subtyping) require less than 600 lines of OCaml code. The current implementation, including parsing, evaluation and LaTeX pretty printing contains less than 6500 lines of code.

We conjecture that our implementation is complete (i.e., it may succeed on all typable programs), provided that enough type annotations are given. On practical instances, the required amount of annotations seems to be reasonably small (see Section 9). Overall, the system provides a similar user experience to statically typed functional languages like OCaml or Haskell. In fact, such languages also require type annotations for advanced features like polymorphic recursion.

[3]It is an open problem whether every normalising extensions of system F is undecidable.

[4]The rules of Figure 7 need to be modified slightly to handle fixpoints.

SubML provides literate programming features inspired by the PhoX language [34]. They can notably be used to generate LaTeX documents. In particular, the examples presented in Sections 5, 6 and 9 (including proof trees) have been generated using SubML, and are therefore machine checked. Many other program examples (more than 4000 lines of code) are provided with the implementation of SubML. They can be used to check that the system is indeed usable in practice. SubML can either be installed from its source code or tried online at `https://lama.univ-smb.fr/subml`.[5]

Applications. In addition to classical examples, our system allows for applications that we find very interesting (see Sections 6 and 9). As a first example, we can program with the Church encoding of algebraic data types. Although this has little practical interest (if any), it requires the full power of System F and is a good test suite for polymorphism. As Church encoding is known for having a bad time complexity, Dana Scott proposed a better alternative using a combination of polymorphism and inductive types [1]. For instance, the type of natural numbers can be defined as follows.

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

Unlike Church numerals, Scott numerals admit a constant time predecessor function with the expected type $\mathbb{N}_S \to \mathbb{N}_S$.

In standard systems, recursion on inductive data types requires specific typing rules for recursors, like in Gödel's System T. In contrast, our system is able to type a recursor encoded as a λ-term, without having to extend the language. This recursor was shown to the second author by Michel Parigot [30]. 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.

We also discovered a surprising λ-calculus coiterator for streams encoded as follows, using an existentially quantified type S as an internal state.

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

An element of type S must be provided to progress in the computation of the stream. Note that here, the product type does not have to be encoded using polymorphism as for Church or Scott encoded data types. As a consequence, the above definition of streams may have a practical interest.

Curry style and type annotations. For our incomplete type checking algorithm to be usable in practice, the user has to guide the system using type annotations. However, the language is Curry style, which means that polymorphic types are interpreted as intersections (and existential types as unions) in the semantics. As a consequence, the terms do not include type abstractions and type applications as in Church style, where polymorphic types are interpreted as functions (and existential types as pairs). This means that it is not possible to introduce a name for a type variable in a term, which is necessary for annotating subterms of polymorphic functions with their types.

As our system relies on choice operators for types, it never manipulates type variables. However, we found a way to name choice operators corresponding to local types using a pattern matching syntax. It can be used to extract the definition of choice operators from

[5] The online version is compiled to Javascript using js_of_ocaml (`https://ocsigen.org/js_of_ocaml/`).

types and make it available to the user for giving type annotations. As an example, we can fully annotate the polymorphic identity function as follows.

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

Note that such annotations are not part of the theoretical type system. They are only provided in the implementation to allow the user to guide the system toward guessing the correct instantiation of unification variables.

Another interesting application of choice operators for types is the dot notation for existential types, which allows the encoding of a module system based on records. As an example, we can encode a signature for isomorphisms with the following type.

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

Given a term h of type Iso, we can then define the following syntactic sugars to access the abstract types corresponding to T and U.

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

The first choice operator denotes a type T such that h has type $\exists U.\{\mathrm{f} : T \to U; \mathrm{g} : U \to T\}$. As our system never infers polymorphic or existential types, we can rely on the name that was chosen by the user for the bound variable. This new approach to abstract types seems simpler than previous work like [10].

Related work. The language presented in this paper is an extension of John Mitchell's System F_η [9], which itself extends Jean-Yves Girard and John Reynolds's System F [13, 36] with subtyping. Unlike previous work [5, 31], our system supports mixed induction and coinduction with polymorphic and existential types. In particular, we improve on an unpublished work of the second author [33]. Our type system also strongly relates to sized types [17] as our inductive and coinductive types carry ordinal numbers. Such a technique is widespread for handling induction [7, 8, 15, 20, 38] and even coinduction [3, 4, 37], in settings where termination is required.

The most important difference difference between this paper and previous work precisely lies in the handling of inductive and coinductive types. In all the systems that the authors are aware of, inductive and coinductive types are strongly linked to recursion, and thus to termination. In particular, they rely on specific rules for checking size relations between ordinal parameters when using recursion. In this paper, inductive and coinductive types are handled in a way that is completely orthogonal to recursion. Ordinal sizes are only manipulated in the subtyping rules related to inductive and coinductive types, while recursion is handled separately using a simple typing rule for the fixpoint combinator. This leads to a system that has a rather simple presentation compared to previous work, even if it relies on unusual concepts such as choice operators and circular proofs.

We believe that our system is simpler than previous work for two main reasons. First, the complete distinction between (general) recursion and inductive and coinductive types allows for simpler, more natural typing and subtyping rules. In particular, we do not need to rely on syntactic conditions such as the semi-continuity used by Andreas Abel [3], or even the standard covariance condition. Instead, we consider a formalism of potentially not well-founded circular proofs. We then make a singular use of the size change principle of Lee, Jones and Ben-Amram [23], which is usually used to prove the termination of programs.

For example, it is used in this way in the work of Pierre Hyvernat [18] and Andreas Abel [2], but also in the implementations of Agda [29] and PML [35]. Here however, the size change principle is not used to prove the termination of programs directly, but to show that typing proofs are well-founded. Termination is then obtained using a semantic proof by well-founded induction on the structure of our typing and subtyping derivations.

To our knowledge, techniques from the termination checking community have never been used to check the correctness of circular proofs before. The literature on circular proofs in general seems to be limited to the work of Luigi Santocanale [39, 40], where circular proofs are related to parity games [41] and given a category-theoretic semantics. However, the considered language is based on the modal μ-calculus [6]. Its expressiveness is thus limited and it does not include subtyping.

Subtyping has been extensively studied in the context of ML-like languages, starting with the work of Roberto Amadio and Luca Cardelli [5]. Recent work includes the MLsub system [11], which extends unification to handle subtyping constraints. Unlike our system, it relies on a flow analysis between the input and output types, borrowed from the work of François Pottier [32]. However, we are not aware of any work on subtyping that leads to a system as expressive as ours for a Curry-style extension of System F. In particular, no other system seems to be able to handle the permutation of quantifiers with other connectives as well as mixed inductive and coinductive types (see Sections 5 and 6).

From a more practical perspective, we chose to trade the decidability of type-checking for simplicity. Indeed, we chose not to look for (and prove) a decidability result, unlike most work on programming languages. We are happy to work with a semi-algorithm as our experiments showed that this is perfectly acceptable in practice. In particular, the user experience is not different from working with meta-variables or implicit arguments as in Coq or Agda [26, 29]. Nevertheless, this is not completely satisfactory, and we would like to prove that our semi-algorithm is complete for the quantifier-free fragment of our calculus. We believe that we could achieve completeness using a specific algorithm to solve size constraints. Such an algorithm has already been used by Frédéric Blanqui for a language with only a successor symbol [19]. This could hopefully be adapted to our setting.

2. 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 [17]. Here, ordinals will also be used to show that infinite typing derivations are well-founded.

Convention 2.1. We will use the vector notation $\overline{e}$ for a tuple $(e_1, \dots, e_n)$ which length will be denoted $|\overline{e}| = n$. The concatenation of two vectors $\overline{x}$ and $\overline{y}$ will be denoted $\overline{x}.\overline{y}$. Note that there will sometimes be implicit constraints on the length of vectors (e.g., when working with substitutions such as $E[\overline{x} := \overline{e}]$).

Definition 2.2. Let $\mathcal{P} = \{P, Q, \dots\}$ be a set of predicate symbols (of mixed arities) ranging over ordinals. The sets of *syntactic ordinals* $\mathcal{O}$ is defined by the first category of the following BNF grammar using a set of ordinal variables $\mathcal{V}_{\mathcal{O}} = \{\alpha, \beta, \dots\}$.

$$\kappa, \tau, \upsilon ::= \alpha \mid \infty \mid \tau + 1 \mid \varepsilon_{\overline{\alpha} < \overline{w}} P(\overline{\alpha}.\overline{\kappa})_i$$
$$w ::= \kappa \mid \mathcal{O}$$

In syntactic ordinals of the form $\varepsilon_{\overline{\alpha}<\overline{w}}P(\overline{\alpha}.\overline{\kappa})_i$, the variables of $\overline{\alpha} = (\alpha_1, \ldots, \alpha_n)$ are bound in $P(\overline{\alpha}.\overline{\kappa})$ but not in $\overline{w}$. Moreover, we enforce $1 \leq i \leq |\overline{\alpha}| = |\overline{w}|$ and $|P| = |\overline{\alpha}| + |\overline{\kappa}|$, where $|P|$ denotes the arity of the predicate P. Note that $\mathcal{O}$ may itself appear in the syntax as an upper bound for ordinal variables.

Syntactic ordinals are built using the constant ∞, a successor symbol and *ordinal choice operators* (or *ordinal witnesses*) of the form $\varepsilon_{\overline{\alpha}<\overline{w}}P(\overline{\alpha}.\overline{\kappa})_i$. Intuitively, the vector $\overline{\tau}$ defined as $\tau_i = \varepsilon_{\overline{\alpha}<\overline{w}}P(\overline{\alpha}.\overline{\kappa})_i$ denotes syntactic ordinals that are point-wise smaller than $\overline{w}$, and such that "$P(\overline{\tau}.\overline{\kappa})$ is true" (this will be made formal in Definition 2.6). In the upper bound $\overline{w}$, one can use the notation $\alpha < \mathcal{O}$ in the case where there is no constraint on the variable α. In other words, $\mathcal{O}$ denotes an ordinal that is bigger than all the syntactic ordinals, and as a consequence it is not a syntactic ordinal itself.

In the semantics, the symbol ∞ will be interpreted using the ordinal 2^{2^ω}, where ω denotes the cardinal of the natural numbers. This ordinal will be large enough to ensure the convergence of all the fixpoints corresponding to inductive and coinductive types. However, 2^{2^ω} cannot be the biggest ordinal of our semantics since larger ones may be represented in the syntax using the successor symbol.[6]

Definition 2.3. We denote $[\![\mathcal{O}]\!]$ the ordinal $2^{2^\omega} + \omega$, which is also the set of all the ordinals of our semantics. Note that it can be thought of as the interpretation of $\mathcal{O}$.

We will now extend the syntax of syntactic ordinals with (actual) ordinals, thus embedding the elements of the semantics into the syntax. This common technique will allow us to substitute variables using ordinals directly, without having to rely on a semantical map for interpreting variables. This will allow us to only manipulate closed (parametric) syntactic ordinals.

Definition 2.4. The set of *parametric syntactic ordinals* $\mathcal{O}^*$ is obtained by extending the language of syntactic ordinals with (actual) ordinals $o \in [\![\mathcal{O}]\!]$.

$$\begin{aligned} \kappa, \tau, \upsilon &::= \alpha \mid \infty \mid \tau + 1 \mid o \mid \varepsilon_{\overline{\alpha}<\overline{w}}P(\overline{\alpha}.\overline{\kappa})_i \\ w &::= \kappa \mid \mathcal{O} \end{aligned}$$

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

Convention 2.5. We will use the notation $\overline{\varepsilon}_{\overline{\alpha}<\overline{w}}P(\overline{\alpha}.\overline{\kappa})$ for the vector $(\varepsilon_{\overline{\alpha}<\overline{w}}P(\overline{\alpha}.\overline{\kappa})_i)_{1\leq i\leq|\alpha|}$. When $|\overline{\alpha}| = |\overline{\kappa}| = 1$, we will write $\varepsilon_{\alpha<w}P(\alpha.\overline{\kappa})$ for both $\overline{\varepsilon}_{\overline{\alpha}<\overline{w}}P(\overline{\alpha}.\overline{\kappa})$ and $\varepsilon_{\overline{\alpha}<\overline{w}}P(\overline{\alpha}.\overline{\kappa})_1$.

We will now give the semantical interpretation of the closed parametric syntactic ordinals, using (actual) ordinals of $[\![\mathcal{O}]\!]$. As syntactic ordinals contain predicate symbols, they will need to be interpreted as well.

Definition 2.6. 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.

$$[\![\infty]\!] = 2^{2^\omega} \qquad [\![\mathcal{O}]\!] = 2^{2^\omega} + \omega \qquad [\![\kappa + 1]\!] = [\![\kappa]\!] + 1 \qquad [\![o]\!] = o \qquad [\![\overline{\kappa}]\!] = ([\![\kappa_1]\!], \ldots, [\![\kappa_n]\!])$$

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

[6] We will in fact never use $\infty + 1$ (or other successors of ∞) in practice.

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

$$\frac{\gamma, \kappa \vdash \kappa \leq_{i-1} \tau \qquad w_j = \kappa \neq \mathcal{O}}{\gamma, \kappa \vdash \varepsilon_{\overline{\alpha}<\overline{w}} P(\overline{\alpha}.\overline{\upsilon})_j \leq_i \tau}\, \varepsilon \qquad \frac{\gamma \vdash \kappa \leq_i \tau \qquad w_j = \kappa \neq \mathcal{O}}{\gamma \vdash \varepsilon_{\overline{\alpha}<\overline{w}} P(\overline{\alpha}.\overline{\upsilon})_j \leq_i \tau}\, \varepsilon_w$$

Figure 1: Rules for ordinal ordering and strict ordering.

Here, $\overline{o} < [\![\overline{w}]\!]$ denotes point-wise ordering on vectors of ordinals, and $\overline{0}$ denotes a vector of 0 ordinals. Note that there may be several possible choices for $\overline{o}$ in the case of an ordinal witness. We will thus consider different *models*, for which the choice of $\overline{o}$ will be made differently. If $\mathcal{M}$ is such a model, we will denote $[\![\kappa]\!]^{\mathcal{M}}$ the induced interpretation.

Convention 2.7. We will most of the time omit to mention the model $\mathcal{M}$. In this case, we will assume that it is fixed, but arbitrary.

Lemma 2.8. *Let $\mathcal{M}$ be a model and $\overline{\varepsilon}_{\overline{\alpha}<\overline{w}} P(\overline{\alpha}.\overline{\kappa})$ be a vector of ordinal choice operators of size n. If $\overline{o} \in [\![\mathcal{O}]\!]^n$ is a vector of ordinals such that $\overline{o} < [\![\overline{w}]\!]^{\mathcal{M}}$ and $[\![P]\!](\overline{o}.[\![\overline{\kappa}]\!]^{\mathcal{M}}) = 1$, then there is a model $\mathcal{M}'$ such that $[\![\overline{w}]\!]^{\mathcal{M}'} = [\![\overline{w}]\!]^{\mathcal{M}}$, $[\![\overline{\kappa}]\!]^{\mathcal{M}'} = [\![\overline{\kappa}]\!]^{\mathcal{M}}$ and $[\![\overline{\varepsilon}_{\overline{\alpha}<\overline{w}} P(\overline{\alpha}.\overline{\kappa})]\!]^{\mathcal{M}'} = \overline{o}$.*

Proof. We define the height $h(\tau)$ of a syntactic ordinal τ as follows.

$$h(\infty) = 0 \qquad h(\mathcal{O}) = 0 \qquad h(o) = 0 \qquad h(\kappa + 1) = 1 + h(k)$$

$$h(\kappa_1, \ldots, \kappa_n) = \max(h(\kappa_1), \ldots, h(\kappa_n)) \qquad h(\varepsilon_{\overline{\alpha}<\overline{w}} P(\overline{\alpha}.\overline{\kappa})_i) = 1 + \max(h(\overline{w}), h(\overline{\kappa}))$$

We then define $[\![\tau]\!]^{\mathcal{M}'}$ by induction on $h(\tau)$ by first taking $[\![\tau]\!]^{\mathcal{M}'} = [\![\tau]\!]^{\mathcal{M}}$ for every τ such that $h(\tau) < h(\varepsilon^i_{\overline{\alpha}<\overline{w}} P(\overline{\alpha}.\overline{\kappa}))$ (including the elements of $\overline{w}$ and $\overline{\kappa}$). We then take $[\![\overline{\varepsilon}_{\overline{\alpha}<\overline{w}} P(\overline{\alpha}.\overline{\kappa})]\!]^{\mathcal{M}'} = \overline{o}$ and we complete the definition by marking arbitrary choices for other 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 the deduction rule system including *ordinal contexts*, which will contain ordinals assumed to be non-zero.

Definition 2.9. An *ordinal contexts* is a finite set of syntactic ordinals represented using lists generated by the following BNF grammar.

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

Note that it will never be useful to store syntactic ordinals of the form $\tau + 1$ or ∞ in an ordinal context as they are necessarily non-zero.

Definition 2.10. The syntactic ordinals are equipped with a family of relations $(\leq_i)$ with $i \in \mathbb{Z}$. Intuitively, $\kappa \leq_i \tau$ can be understood as "$\kappa + i \leq \tau$" when $i \geq 0$ and as "$\kappa \leq \tau + (-i)$" when $i \leq 0$. Given a context of positive ordinals γ, the relation $(\leq_i)$ is defined 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 (s_r) rule commutes with the (s_l), (ε) and (ε_w) rules. When both rules (ε) and (ε_w) may apply it is better to use (ε) as it yields a lower index, and thus proves more judgments according to Lemma 2.11, (2).

Lemma 2.11. *For every ordinal contexts γ and δ, every syntactic ordinals κ_1, κ_2 and κ_3, and for every integers i and j we have:*

(1) if $\gamma \vdash \kappa_1 \leq_i \kappa_2$ then $\gamma, \delta \vdash \kappa_1 \leq_i \kappa_2$,
(2) if $\gamma \vdash \kappa_1 \leq_i \kappa_2$ and $j \leq i$ then $\gamma \vdash \kappa_1 \leq_j \kappa_2$,
(3) 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 proofs of (1) and (2) are immediate by induction on the derivation. We prove (3) by induction on the sum of the sizes 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 conclude using (2). If the last rule used on the left is (s_l) then $\kappa_1 = \kappa + 1$. By induction hypothesis we have $\gamma \vdash \kappa \leq_{i+j+1} \kappa_3$ and thus $\gamma \vdash \kappa_1 \leq_{i+j} \kappa_3$. A similar argument can be used 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 $\kappa_1 = \varepsilon_{\overline{\alpha}<\overline{w}} P(\overline{\alpha}.\overline{v})_m$ and $w_m = \kappa \neq \mathcal{O}$. By induction hypothesis, we get $\gamma \vdash \kappa \leq_{i+j-1} \kappa_3$ if we applied the (ε) rule or $\gamma \vdash \kappa \leq_{i+j} \kappa_3$ if we applied the (ε_w) rule. In both cases this implies $\gamma \vdash \kappa_1 \leq_{i+j} \kappa_3$. If the last rule used on the right is the (ε) or (ε_w) then we must be in one of the previous cases. Indeed, the rules that can be applied on the left when κ_2 is an ordinal witness are $(=)$, (s_l), (ε) and (ε_w). □

Lemma 2.12. *Let γ be a closed context, κ_1, κ_2 be closed syntactic ordinals and i be an integer such that such that $\gamma \vdash \kappa_1 \leq_i \kappa_2$ is derivable. For any model, if $[\![\tau]\!] \neq 0$ for all $\tau \in \gamma$ then $[\![\kappa_1]\!] + i \leq [\![\kappa_2]\!]$ when $i \geq 0$ and $[\![\kappa_1]\!] \leq [\![\kappa_2]\!] + (-i)$ when $i \leq 0$.*

Proof. The proof is done by induction on the derivation of $\gamma \vdash \kappa_1 \leq_i \kappa_2$. The cases for the $(=)$, (s_l) and (s_r) rules are immediate. In the case of the (ε) rule we have $\kappa_1 = \varepsilon_{\overline{\alpha}<\overline{w}} P(\overline{\alpha}.\overline{v})_j$ with $w_j = \kappa \neq \mathcal{O}$. As a consequence, $[\![\kappa_1]\!]$ is either equal to some ordinal $o_j < [\![\kappa]\!]$ or to 0. Since $[\![\kappa]\!] \neq 0$, we have $[\![\kappa_1]\!] < [\![\tau_j]\!]$ in both cases and we can thus conclude by induction hypothesis. In the case of the (ε_w) rule the proof is similar, but it is possible that $[\![\kappa]\!] = 0$ so we only have $[\![\kappa_1]\!] \leq [\![\kappa]\!]$. □

3. Size change matrices

We will now consider the formalism that will be used to relate our syntactic ordinals to the size-change principle [23] in the following sections. The main idea will be to represent the size informations contained in the circular structure of our proofs using matrices. We will then be able to easily compose size informations using matrix product.

Definition 3.1. 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 below as its product. Note that the neutral element of (min) is -1 and that the neutral element of $(\circ)$ is ∞.

$$\begin{aligned} x \circ \infty &= \infty & \qquad -1 \circ x &= -1 \quad \text{if } x \neq \infty \\ \infty \circ x &= \infty & x \circ -1 &= -1 \quad \text{if } x \neq \infty \\ 0 \circ 0 &= 0 \end{aligned}$$

Intuitively, -1 will be used to indicate that the size of some object decreases, 0 will be used when the size does not increase and ∞ will be used when there is no size information.

Definition 3.2. 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 of A and B, denoted 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 exactly corresponds to the usual matrix product expressed with the operations of our semi-ring $(\{-1, 0, \infty\}, min, \circ)$.

Lemma 3.3. *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 $\min_{1 \leq k \leq p} L_{i,k} \circ C_{k,j} = \min_{1 \leq k \leq m} A_{i,k} \circ R_{k,j}$.

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

□

To conclude this section, we will now link the notion of size-change matrix to an order relation. In particular, we will show that the matrix product indeed corresponds to the composition of size informations. In other words, the product corresponds to the application of the transitivity of the order relation on vectors.

Definition 3.4. Let A be an $n \times m$ size-change matrix, $(X, \leq)$ be an ordered set and $\overline{x}$, $\overline{y}$ be two vectors of X with $|\overline{x}| = n$ and $|\overline{y}| = m$. We write $\overline{y} <_A \overline{x}$ if for all $1 \leq i \leq n$ and for all $1 \leq j \leq m$ we have $y_j < x_i$ when $A_{i,j} = -1$, and $y_j \leq x_i$ when $A_{i,j} = 0$.

Lemma 3.5. *Let $(X, \leq)$ be an ordered set and $\overline{x}$, $\overline{y}$ and $\overline{z}$ be three vectors of X with $|\overline{x}| = n$, $|\overline{y}| = m$ and $|\overline{z}| = p$. If A is an $n \times m$ size-change matrix such that $\overline{y} <_A \overline{x}$ and if B is an $m \times p$ size-change matrix such that $\overline{z} <_B \overline{y}$ then $\overline{z} <_{AB} \overline{x}$.*

Proof. Let us take $C = AB$. By definition, if $C_{i,j} = -1$ there must be k such that $A_{i,k} \circ B_{k,j} = -1$. This can only happen if $A_{i,k} = B_{k,j} = -1$, if $A_{i,k} = -1$ and $B_{k,j} = 0$, or if $A_{i,k} = 0$ and $B_{k,j} = -1$. In these three cases we respectively have $z_j < y_k < x_i$, $z_j < y_k \leq x_i$ and $z_j \leq y_k < x_i$, which all imply $z_j < x_i$. Now, if $C_{i,j} = 0$ then there must be k such that $A_{i,k} \circ B_{k,j} = 0$, which implies $z_j \leq y_k \leq x_i$. □

4. Circular proofs and size change principle

We will now introduce an abstract notion of circular proof, with a related notion of *well-foundedness*. The idea is to represent proofs as directed acyclic graphs, and to label their edges with size relations between syntactic ordinals. These size relations (expressed using size-change matrices) are then processed using the size change principle [23]. In this paper, it will first allow us to build circular subtyping proofs to handle inductive and coinductive types in Section 5. It will then be used to build circular typing proofs in Section 8 to ensure the termination of recursive programs.

Our notion of circular proof is parametrised by a notion of *abstract judgments*, their deduction rules and their semantics. They will correspond, for example, to typing judgments or to local subtyping judgments, with their respective deduction rules and interpretations. We believe that the framework presented here could be applied to other type systems involving a notion of size.

Definition 4.1. A language of *abstract judgments* is given by a set $\mathcal{J}$ of symbolic judgments, and an associated set Λ of *individuals*. Every symbol $J \in \mathcal{J}$ should depends on exactly one element of Λ and on $|J|$ syntactic ordinals (possibly 0). Optionally, for some $J \in \mathcal{J}$ and for all $\overline{\kappa} \in \mathcal{O}^{|J|}$ there may be a choice operator $\varepsilon_x \neg J(x, \overline{\kappa})$ in Λ, where x is a bound variable. It will be used as a counter-example to "for all $t \in \Lambda$, the judgment $J(t, \overline{\kappa})$ is valid". We denote $[\![\Lambda]\!] \subset \Lambda$ the set of all the individuals that do not contain choice operators.

Intuitively, an abstract judgments can be seen as a predicate, which validity depends on the truth of the denoted judgment. In the following, such predicates will be used to build syntactic ordinal witnesses according to Section 2. We will thus work with syntactic ordinals of the form $\varepsilon_{\overline{\alpha}<\overline{\kappa}} J(t, \overline{\alpha}.\overline{\upsilon})_i$ or $\varepsilon_{\overline{\alpha}<\overline{\kappa}} \forall x J(x, \overline{\alpha}.\overline{\upsilon})_i$, for example. However, note that we will only be able to quantify over all the individuals when a corresponding choice choice operator $\varepsilon_x \neg J(x, \overline{\kappa})$ is provided.

Definition 4.2. Given a language of abstract judgments $(\mathcal{J}, \Lambda)$, we can build a language of predicates $\mathcal{P}$ using the following BNF grammar, where $J \in \mathcal{J}$ and $t \in \Lambda$.

$$P, Q ::= J(t, \overline{\alpha}) \mid \neg J(t, \overline{\alpha}) \mid \forall x J(x, \overline{\alpha}) \mid \forall x \neg J(x, \overline{\alpha})$$

We then obtain a fixed language of (parametric) syntactic ordinals by instantiating Definitions 2.2 and 2.4 using $\mathcal{P}$.

We will now consider the interpretation of individuals and abstract judgments. Intuitively, an individual (potentially containing choice operators) will be interpreted by a *pure* individual (i.e., one that does not contain choice operators). An abstract judgment is then interpreted as predicates over a pure individual and (actual) ordinals.

Definition 4.3. Let $(\mathcal{J}, \Lambda)$ be a language of abstract judgments. Every individual $t \in \Lambda$ is interpreted by a pure individual $[\![t]\!] \in [\![\Lambda]\!]$, and every abstract judgment $J \in \mathcal{J}$ of arity n is interpreted by a function $[\![J]\!] : [\![\Lambda]\!] \times [\![\mathcal{O}]\!]^n \to \{0, 1\}$. The predicates over ordinals built according to the previous definition are then interpreted as follows.

$$[\![J(t, \overline{\alpha})]\!] = \overline{\alpha} \mapsto [\![J]\!]([\![t]\!], \overline{\alpha}) \qquad [\![\neg J(t, \overline{\alpha})]\!] = \overline{\alpha} \mapsto 1 - [\![J]\!]([\![t]\!], \overline{\alpha})$$

$$[\![\forall x J(x, \overline{\alpha})]\!] = \overline{\alpha} \mapsto \min_{t \in [\![\Lambda]\!]} [\![J]\!](t, \overline{\alpha}) \qquad [\![\forall x \neg J(x, \overline{\alpha})]\!] = \overline{\alpha} \mapsto 1 - \max_{t \in [\![\Lambda]\!]} [\![J]\!](t, \overline{\alpha})$$

Moreover, we require that for every individual of the form $\varepsilon_x \neg J(x, \overline{\kappa})$ with $J \in \mathcal{J}$ and $\overline{\kappa} \in \mathcal{O}$, we have $[\![\varepsilon_x \neg J(x, \overline{\kappa})]\!] = u \in [\![\Lambda]\!]$ such that $[\![J]\!](u, [\![\overline{\kappa}]\!]) = 0$ if such a u exists, otherwise u is chosen to be an arbitrary element of $[\![\Lambda]\!]$.

Definition 4.4. An *abstract sequent* $\gamma \vdash J(t, \overline{\kappa})$ is built using an ordinal context $\gamma \subseteq \mathcal{O}$, an abstract judgment $J \in \mathcal{J}$, an individual $t \in \Lambda$ and syntactic ordinals $\overline{\kappa} \in \mathcal{O}^{|J|}$. We say that the abstract sequent $\gamma \vdash J(t, \overline{\kappa})$ is true if we have $[\![J]\!]([\![t]\!], [\![\overline{\kappa}]\!]) = 1$ whenever $[\![\tau]\!] \neq 0$ for all $\tau \in \gamma$.

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

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

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

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

Figure 2: Generalisation and induction rules for general abstract sequents.

To relate the notion of size-change matrices 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 4.5. A list of *ordinal constraints* C of arity n is given by a function C from $\{1,\ldots,n\}$ to $\{0,1,\ldots,n\}$. Given a vector of ordinals $\overline{o} \in [\![\mathcal{O}]\!]^n$, we denote $C(\overline{o})$ the vector of size n defined as $C(\overline{o})_i = [\![\mathcal{O}]\!]$ if $C(i) = 0$ and as $C(\overline{o})_i = o_j$ if $C(i) = j \neq 0$. We say that C is satisfied by $\overline{o} \in [\![\mathcal{O}]\!]^n$ when $o_i < C(\overline{o})_i$ for all $1 \leq i \leq n$.

Building circular proofs will require the generalisation of abstract 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 4.6. A *general abstract sequent* is an abstract sequent that is quantified over. It may be of the form $\forall\overline{\alpha}\,(\gamma \vdash C(\overline{\alpha}) \Rightarrow J(t,\overline{\alpha}.\overline{\upsilon}))$ or $\forall\overline{\alpha}\forall x\,(\gamma \vdash C(\overline{\alpha}) \Rightarrow J(x,\overline{\alpha}.\overline{\upsilon}))$, where γ is an ordinal context only containing variables of $\overline{\alpha}$, C is a list of ordinal constraints of arity $|\overline{\alpha}|$, J is an abstract judgement and t is an individual. We say that the general abstract sequent $\forall\overline{\alpha}\,(\gamma \vdash C(\overline{\alpha}) \Rightarrow J(t,\overline{\alpha}.\overline{\upsilon}))$ (resp. $\forall\overline{\alpha}\forall x\,(\gamma \vdash C(\overline{\alpha}) \Rightarrow J(x,\overline{\alpha}.\overline{\upsilon}))$) is true if $[\![J]\!]([\![t]\!],\overline{o}.[\![\overline{\upsilon}]\!]) = 1$ (resp. $\min_{t \in [\![\Lambda]\!]}[\![J]\!](t,\overline{o}.[\![\overline{\upsilon}]\!]) = 1$) for all $\overline{o} \in [\![\mathcal{O}]\!]^n$ such that $o_i \neq 0$ if $\alpha_i \in \gamma$ and such that C is satisfied by $\overline{o}$.

Note that in a general abstract sequent, a judgement J may use ordinals $\overline{\upsilon}$ 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 $C(\overline{\alpha})$ cannot use ordinals of $\overline{\upsilon}$ (therefore, they can only use ordinals of $\overline{\alpha}$). This restriction is not essential, but simplifies the definitions to come.

Definition 4.7. A *circular deduction system* is given by a set of deduction rules defined over abstract sequents (i.e., their conclusions and premises are abstract sequents), together

$$
\cfrac{\cfrac{\cfrac{[\forall()(\vdash () \Rightarrow J(t,\overline{v}))]_0 \qquad \varnothing}{\vdash J(t,\overline{v})}\,\mathrm{G}}{\forall()(\vdash () \Rightarrow J(t,\overline{v}))}\,\mathrm{I}_0 \qquad \varnothing}{\gamma \vdash J(t,\overline{v})}\,\mathrm{G}
$$

Figure 3: Example of a circular proof that is not well-founded.

with the rules of Figure 2. The aim of the *generalisation rules* (G) and (G^+) is to prove an abstract sequent using a general abstract sequent. In particular, the ordinal constraints used in their first premise should be satisfied in the conclusion (see their second premise). The *induction rules* (I_k) and (I_k^+) may be used to prove a general abstract sequent using itself as an hypothesis (this is the meaning of the square brackets).[7] Note that a natural number k (unique in a proof) is used to keep track of the originating induction rule.

The rules (I_k), (G), (I_k^+) and (G^+) are the only ones allowed to manipulate general abstract sequents. The induction rules alone are responsible for the circular structure of the proofs in a circular deduction system. In particular, they allow for clearly invalid proofs such as the on of Figure 3, which can be used to prove an arbitrary abstract sequent. After applying a generalisation rule over the empty vector of ordinals (), the invalid proof is constructed by using an induction rule and by applying the new hypothesis directly.

As a circular deduction system can be used to build incorrect circular proofs, we will need to rely on a well-foundedness criterion. In other words, a derivable (general) abstract sequent will only be considered correct if its derivation is well-founded. In this paper, we will rely on the size-change principle [23] to obtain a sufficient condition for a given proof to be well-founded. To this aim, circular proofs first need to be decomposed into *blocks*.

Definition 4.8. 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 require blocks to be minimal, which means that they should not contain general abstract sequents (except in their conclusions 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 of ordinals $\overline{\alpha}$ in the conclusion of B otherwise.

Definition 4.9. 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, which may be directly above B in Π, B itself, or even a block below 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 in Figure 4, if we only

[7]Note that the (I_k^+) rule relies on the individuals of the form $\varepsilon_x\neg J(x,\overline{\kappa})$ required by Definition 4.1.

$$\dfrac{\dfrac{\vdots\ B_2}{\forall\overline{\beta}(\delta \vdash D(\overline{\beta}) \Rightarrow K(u,\overline{\beta}))} \qquad (\delta[\overline{\beta} := \overline{\tau}], \delta' \vdash \tau_i < D(\overline{\tau})_i)_{1 \leq i \leq |\overline{\beta}|}}{\delta[\overline{\beta} := \overline{\tau}], \delta' \vdash K(u, \overline{\tau})}\,\mathrm{G} \qquad B_1$$

$$\vdots$$

$$\dfrac{\gamma[\overline{\alpha} := \overline{\kappa}] \vdash J(t, \overline{\kappa}) \quad \text{where } \overline{\kappa} = \varepsilon_{\overline{\alpha} < C(\overline{\alpha})} \neg J(t, \overline{\alpha})}{\forall \overline{\alpha}(\gamma \vdash C(\overline{\alpha}) \Rightarrow J(t, \overline{\alpha}))}\,\mathrm{I}_k$$

Figure 4: Construction of an edge of the call graph (see Definition 4.9).

include the premises involved in the definition of the edge (B_1, B_2).[8] The $|\overline{\alpha}| \times |\overline{\beta}|$ matrix M attached to the edge (B_1, B_2) is then defined as $M_{i,j} = -1$ when $\delta[\overline{\beta} := \overline{\tau}], \delta' \vdash \tau_j < \kappa_i$ is derivable, $M_{i,j} = 0$ when only $\delta[\overline{\beta} := \overline{\tau}], \delta' \vdash \tau_j \leq \kappa_i$ is derivable, and $M_{i,j} = \infty$ otherwise.

As the edges of a call graph are labeled with matrices, any path in its transitive closure can also be assigned a label using the matrix product of the labels along the path. In particular, if there is a path from B_1 to B_2 with label M and a path from B_2 to B_3 with label N, then there is a path from B_1 to B_3 with label MN. Since a call graph has finitely many vertices and edges, the number of possible labels for a path in the transitive closure of the graph is also finite. If we consider two paths with the same label to be equal, then there can only be finitely many distinct paths in the transitive closure of a call graph. It can hence be computed in finite time by composing edges until saturation.

Definition 4.10. 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 of its label. Note that such loops are necessarily labeled with square matrices.

Example 4.11. We now consider the example of circular proof given in the upper part of Figure 5. For simplicity, the individuals and ordinals are not given explicitly. We will however assume that (besides reflexivity) it is possible to derive $\kappa_2 \vdash \upsilon_2 < \kappa_2$ 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 5. Its transitive closure contains five idempotent loops. There are none on the block B_0, two on the block B_1 with labels $\left(\begin{smallmatrix} 0 & \infty \\ \infty & -1 \end{smallmatrix}\right)$ and $\left(\begin{smallmatrix} -1 & \infty \\ \infty & -1 \end{smallmatrix}\right)$, and three on block 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 thus conclude that our proof example is indeed well-founded since every idempotent loop is labeled with a matrix having at least one -1 on its diagonal[9].

Example 4.12. The circular proof of Figure 3 is built using only two blocks. The upper block has one loop labelled with the empty matrix. It is therefore not well-founded.

Lemma 4.13. *The four deduction rules of Figure 2 are correct. In other words, if the premises of such a rule are semantically valid, then so is its conclusion.*

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

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

[9] The proof example of Figure 5 corresponds to the block decomposition of Figure 10 page 23.

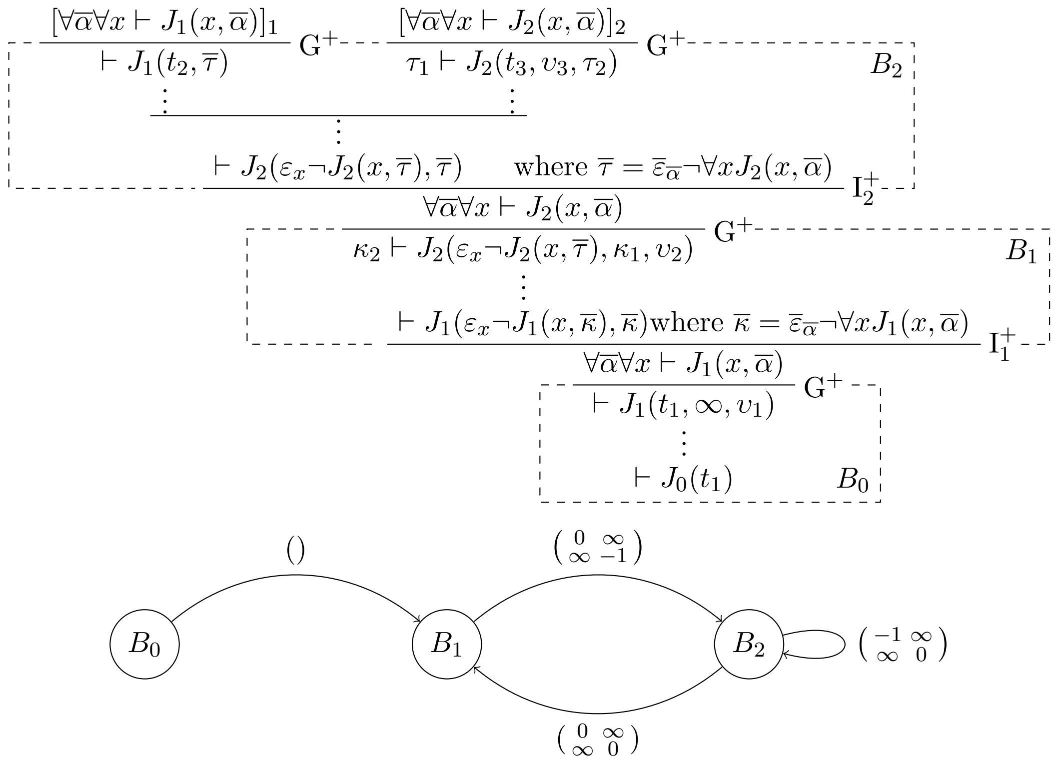


Figure 5: Example of a circular proof and the corresponding call graph.

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 implies the correctness by contraposition. □

Note that the correctness of the (I_k) and (I_k^+) rules rely on the fact that we ignore the hypothesis they introduce. They justification for such hypotheses are handled globally by our notion of well-founded proof (Definition 4.10).

Theorem 4.14. *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 sequent 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 sequent is false. As all the deduction rules are supposed correct (by hypothesis and by Lemma 4.13), 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 $(B_i, \overline{o}_i, \mathcal{M}_i)_{i \in \mathbb{N}}$ 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, $\overline{o}_0$ 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 that $\overline{o}_i$ is a counterexample (thus $|\overline{o}_i| = |B_i|$). We will also require that the call-graph contains an edge linking B_i to B_{i+1} labeled with a matrix M_i such that $\overline{o}_{i+1} <_{M_i} \overline{o}_i$ (the conclusion of B_{i+1} is thus a general abstract sequent).

Note that the first element of the sequence $(B_0, \overline{o}_0, \mathcal{M}_0)$ 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 be empty.

Let us now suppose that the sequence has been constructed for all $0 \leq j \leq i$, and define $(B_{i+1}, \overline{o}_{i+1}, \mathcal{M}_{i+1})$. 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 I_k or I_k^+. Without loss of generality we can assume that it is I_k^+, and thus B_i ends with the following rule.

$$\dfrac{\gamma[\overline{\alpha} := \overline{\kappa}] \vdash J(\varepsilon_x \neg J(x, \overline{\kappa}), \overline{\kappa}) \qquad \text{where } \overline{\kappa} = \overline{\varepsilon}_{\overline{\alpha} < C(\overline{\alpha})} \neg \forall x J(x, \overline{\alpha})}{\forall \overline{\alpha} \forall x (\gamma \vdash C(\overline{\alpha}) \Rightarrow J(x, \overline{\alpha}))}\ \mathrm{I}_k^+$$

By construction, we know that $\forall \overline{\alpha} \forall x (\gamma \vdash C(\overline{\alpha}) \Rightarrow J(x, \overline{\alpha}))$ is false in the model $\mathcal{M}_i$ and that $\overline{o}_i$ is a counterexample. This means that $[\![\gamma[\overline{\alpha} := \overline{o}_i]]\!]^{\mathcal{M}_i}$ contains only positive ordinals, C is satisfied by $\overline{o}$ and $[\![J]\!]^{\mathcal{M}_i}(t, \overline{o}_i) = 0$ for all $t \in \Lambda$. Thus, using Lemma 2.8 we can define $\mathcal{M}_{i+1}$ to be a model such that $[\![\overline{\kappa}]\!]^{\mathcal{M}_{i+1}} = \overline{o}_i$. By definition 4.3 the individual $t = [\![\varepsilon_x \neg J(x, \overline{\kappa})]\!]^{\mathcal{M}_{i+1}}$ satisfies $[\![J(t, \overline{o}_i)]\!] = 0$. 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.

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

As the conclusion of this rule is false is the model $\mathcal{M}_{i+1}$, we know that $[\![\delta[\overline{\beta} := \overline{\tau}], \delta']\!]^{\mathcal{M}_{i+1}}$ only contains positive ordinals and that $[\![K]\!]^{\mathcal{M}_{i+1}}([\![u]\!]^{\mathcal{M}_{i+1}}, [\![\overline{\tau}]\!]^{\mathcal{M}_{i+1}}) = 0$. By Proposition 2.12, the right premises of our G^+ rule cannot be false. Therefore, $\forall \overline{\beta} \forall y (\delta \vdash D(\overline{\beta}) \Rightarrow K(y, \overline{\beta}))$ 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 $\overline{o}_{i+1}$ to be $[\![\overline{\tau}]\!]^{\mathcal{M}_{i+1}}$, 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 $\overline{o}_{i+1} <_{M_i} \overline{o}_i$ to conclude the construction of our sequence. Let us take $1 \leq m \leq |\overline{o}_i|$ and $1 \leq n \leq |\overline{o}_{i+1}|$ and consider $(M_i)_{m,n}$. If it is equal to -1 then there is a proof of $\delta[\overline{\beta} := \overline{\tau}], \delta' \vdash \tau_n < \kappa_m$ and hence proposition 2.12 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 $\overline{o}_{i+1}$ 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 the same argument as in the proof of [23, 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 3.5 to obtain $\overline{o}_j <_{M_{i,j}} \overline{o}_i$ 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, $\overline{o}_{u_{i+1}} <_M \overline{o}_{u_i}$ 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. □

5. 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 normalising in Section 7. Surprisingly, recursion is still possible (for specific algebraic data types) using λ-calculus recursors that are typable thanks to subtyping (see Section 6). The language is formed using three syntactic entities: terms, types and syntactic ordinals (see Section 2). Syntactic 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 8 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 are defined mutually inductively due the choice operators that are contained in their syntax.

Definition 5.1. 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 (or individuals) Λ, the set of types (or formulas) $\mathcal{F}$ and the set of syntactic ordinals $\mathcal{O}$ are defined mutually inductively. The terms and types are defined using the following two BNF grammars.

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

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

The syntactic ordinals are build according to Definitions 2.2 and 4.2 using abstract judgments of the form $J(t, \overline{\tau}) = t : A$ and $J(t, \overline{\tau}) = t \in A \subset B$, where the ordinals of $\overline{\tau}$ may appear in the formulas A and B. Note that choice operators for individuals are provided for all the abstract judgments of the second form. Formally, Definition 4.3 requires terms of the form $\varepsilon_x \neg(x : A)$ and $\varepsilon_x \neg(x \in A \subset B)$. The former will not be provided in the syntax since we will never use the (G^+) and (I_k^+) rules on typing judgments. The latter will be syntactically encoded as $\varepsilon_{x\in A}(x \notin B)$, which will have the intended semantics. Note that in general, we require terms of the form $\varepsilon_{x\in A}(t \notin B)$ not to contain any free λ-variable (e.g., $\lambda y.\varepsilon_{x\in A}(y\, x \notin B)$ is not valid).

The term language contains the usual syntax of the λ-calculus extended with records, projections, constructors and pattern matching (see the reduction rules of Figure 6). A term of the form $\varepsilon_{x\in A}(t \notin B)$ corresponds to a choice operator denoting a closed term u of type A such that $t[x := u]$ does not have type B.[10] The restriction to closed choice is absolutely necessary for their interpretation in the semantics.

[10] Note that in a choice operator like $\varepsilon_{x\in A}(t \notin B)$, the variable x is bound in the term t.

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

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

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

Figure 6: Reduction rules of the language (without general recursion).

Convention 5.2. In our meta-language, we use the notation $\{(l_i = t_i)_{i\in I}\}$ (where I is a finite subset of $\mathbb{N}$) to denote a record. For example, if $I = \{1, 2\}$ then $\{(l_i = t_i)_{i\in I}\}$ corresponds to $\{l_1 = t_1; l_2 = t_2\}$. Similar notations are used for pattern matchings, 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 (or variant).

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. Note that our product types may be either *strict* or *extensible*. A record having an extensible product type (marked with an ellipsis) will be allowed to contain more fields than those explicitly specified, while records with a strict product type will only contain the specified fields. From a subtyping point of view, extensible records are obviously more interesting. However, strict product types will allow us to express a stronger type safety result based on a semantic proof (Theorem 7.27). Our inductive and coinductive types carry size information in the form of a syntactic ordinals κ. The ordinal ∞ is supposed to be large enough so that the construction of $\mu_\infty F$ and $\nu_\infty F$ converges. In particular, when F is covariant then correspond to the least and greatest fixpoints of F. Choice operators $\varepsilon_X(t \in A)$ and $\varepsilon_X(t \notin A)$ are also provided for types.[11] As for our term choice operators, they correspond to witnesses of the property they denote, and they will be interpreted as such in the semantics. However, contrary to term choice operators, they do not need to be closed to be given a semantical interpretation.

Convention 5.3. To lighten the syntax and reduce the need for parentheses we will use some syntactic sugars. We will sometimes group binders and write $\lambda x\, y.t$ for $\lambda x.\lambda y.t$, and $\forall X\, Y.A$ for $\forall X.\forall Y.A$. Morever, we will consider that binders have the lowest priority, which means that $\lambda x.x\, x$ is to be read as $\lambda x.(x\, x)$, and $\forall X.A \Rightarrow B$ as $\forall X.(A \Rightarrow B)$. We will write $\mu X.A$ for $\mu_\infty X.A$ and $\nu X.A$ for $\nu_\infty X.A$, and we will sometimes use the letter F to denote a type with one parameter $X \mapsto A$ so that we can write $F(\mu_\kappa F)$ for $A[X := \mu_\kappa X.A]$. In pattern matchings, we will use the notation $C_k x \to t$ to denote $C_k \to \lambda x.t$. Finally, we will will write $t.C_k$ for the term $[t \mid C_k \to \lambda x.x]$, also written $[t \mid C_k\, x \to x]$.

We now define the reduction relation of our language, which contains β-reduction and rules for pattern matching and record projection. The terms corresponding to runtime errors are also reduced to a diverging term Ω for termination to subsume type safety.

Definition 5.4. The reduction relation $(\succ) \subseteq \Lambda \times \Lambda$ is defined as the contextual closure of the rules given in Figure 6. Its reflexive, transitive closure is denoted $(\succ^*)$.

[11] In the choice operators $\varepsilon_X(t \in A)$ and $\varepsilon_X(t \notin A)$ for types, the variable X is bound in A only.

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

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

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

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

Figure 7: Typing rules for the system without general recursion.

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

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

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

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

$$\frac{I_2 \subseteq I_1 \qquad (\gamma \vdash t.l_i \in A_i \subset B_i)_{i\in I_2}}{\gamma \vdash t \in \{(l_i : A_i)_{i\in I_1}\} \subset \{(l_i : B_i)_{i\in I_2}; \dots\}}\times_{se}$$

$$\frac{I_2 \subseteq I_1 \qquad (\gamma \vdash t.l_i \in A_i \subset B_i)_{i\in I_2}}{\gamma \vdash t \in \{(l_i : A_i)_{i\in I_1}; \dots\} \subset \{(l_i : B_i)_{i\in I_2}; \dots\}}\times_e$$

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

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

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

Figure 8: Subtyping rules for the system without general recursion.

As our system relies on choice operators, usual typing contexts assigning a type to free variables are not required. In particular, open terms will never appear in typing and subtyping rules.

Definition 5.5. 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 Figures 7 and 8 respectively. Both forms of judgments can be used as abstract sequents (in the sense of Definition 4.4) to build well founded circular proofs (see Section 4). In fact, we will only use the (G^+) and (I_k^+) rules[12], and only allow circularity on subtyping proofs.

Thanks to local subtyping judgements, quantifiers are exclusively handled in the subtyping part of the system. The use of choice operators enables many valid permutations of quantifiers with other connectives, while preserving the syntax-directed nature of the system. Let aside the (G^+) and (I_k^+) rules, only one typing rule applies for every term constructor, and essentially one local subtyping rule applies for every two type constructors (see the beginning of Section 10). In the context of our type system, the (G^+) and (I_k^+) rules can be written as follows.

$$\frac{\forall \overline{\alpha} \forall x (\gamma \vdash C(\overline{\alpha}) \Rightarrow x \in A \subset B) \qquad (\gamma[\overline{\alpha} := \overline{\kappa}], \delta \vdash \kappa_i < C(\overline{\kappa})_i)_{1 \leq i \leq |\overline{\alpha}|}}{\gamma[\overline{\alpha} := \overline{\kappa}], \delta \vdash t \in A[\overline{\alpha} := \overline{\kappa}] \subset B[\overline{\alpha} := \overline{\kappa}]}\ \mathrm{G}^+$$

$$\frac{\begin{array}{l}[\forall \overline{\alpha} \forall x (\gamma \vdash C(\overline{\alpha}) \Rightarrow x \in A \subset B)]_k \\ \quad\quad\quad\quad\vdots \\ \gamma[\overline{\alpha} := \overline{\kappa}] \vdash A[\overline{\alpha} := \overline{\kappa}] \subset B[\overline{\alpha} := \overline{\kappa}] \qquad \text{where } \overline{\kappa} = \overline{\varepsilon}_{\overline{\alpha} < C(\overline{\alpha})}(A \not\subset B)\end{array}}{\forall \overline{\alpha} \forall x (\gamma \vdash C(\overline{\alpha}) \Rightarrow x \in A \subset B)}\ \mathrm{I}_k^+$$

Overall, our rules use syntactic ordinals of the forms $\varepsilon_{\alpha<\kappa}(t \in F(\nu_\tau F))$, $\varepsilon_{\alpha<\kappa}(t \notin F(\nu_\tau F))$ and $\overline{\varepsilon}_{\overline{\alpha}<C(\overline{\alpha})}(A \not\subset B)$. They are all built from our two forms of abstract judgments according to Definition 4.1 (up to notations). We respectively write $t \in A$ and $t \notin A$ for $t : A$ and $\neg(t : A)$, and we also write $A \not\subset B$ for $\neg \forall x (x \in A \subset B)$.

Example 5.6. Mitchell's containment axiom. In our system, it is possible to derive Mitchell's containment axiom [27], as well as 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 derivation of the former is given in Figure 9 (it is not circular). Note that the choice operators for terms and types are all well defined (their definitions are not cyclic).

Example 5.7. Mixed inductive and coinductive types. Our system is suitable for handling types containing alternations of inductive and coinductive types. Let us consider the following two types S and L where $F(X,Y)$ is a predicate covariant in X and in Y.

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

The elements of S can be thought of as streams of A's and B's that only contain finitely many A's. The elements of L are streams that do not contain infinitely many consecutive A's. In our system, it is possible to prove $\mathrm{S} \subset \mathrm{L}$ using the circular proof displayed in Figure 10. Note that the block decomposition of the proof is given in Example 4.11. We can thus conclude that it is well-founded (and thus valid).

[12]The (G) and (I_k) will be used in Section 8 to handle general recursion.

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

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 9: Derivation of Mitchell's containment axiom.

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

$$
\begin{aligned}
\mathrm{F}(X) &= \nu Y.[\mathrm{A\ of\ } X \mid \mathrm{B\ of\ } Y] & \kappa_3 &= \varepsilon_{\alpha<\kappa_2}(x_1 \in \mathrm{F}(S_\alpha)) \\
S_\alpha &= \mu_\alpha X \nu Y[A \text{ of } X | B \text{ of } Y] = \mu_\alpha X F(X) & \kappa_4 &= \overline{\varepsilon}_{\alpha_1,\alpha_2<\mathcal{O},\mathcal{O}}(\mathrm{F}(S_{\alpha_2}) \not\subset \mathrm{G}(L_{\alpha_1}))_1 \\
\mathrm{G}(Y) &= \mu X.[\mathrm{A\ of\ } X \mid \mathrm{B\ of\ } Y] & \kappa_5 &= \overline{\varepsilon}_{\alpha_1,\alpha_2<\mathcal{O},\mathcal{O}}(\mathrm{F}(S_{\alpha_2}) \not\subset \mathrm{G}(L_{\alpha_1}))_2 \\
L_\alpha &= \nu_\alpha Y \mu X[A \text{ of } X | B \text{ of } Y] = \nu_\alpha Y G(Y) & \kappa_6 &= \varepsilon_{\alpha<\kappa_4}(x_2.B \notin \mathrm{G}(L_\alpha)) \\
\kappa_0 &= \varepsilon_{\alpha<\infty}(x_0 \notin \mathrm{G}(L_\alpha)) & x_0 &= \varepsilon_{x\in\mathrm{S}}(x \notin \mathrm{L}) \\
\kappa_1 &= \overline{\varepsilon}_{\alpha_1,\alpha_2<\mathcal{O},\mathcal{O}}(S_{\alpha_2} \not\subset \mathrm{G}(L_{\alpha_1}))_1 & x_1 &= \varepsilon_{x\in S_{\kappa_2}}(x \notin \mathrm{G}(L_{\kappa_1})) \\
\kappa_2 &= \overline{\varepsilon}_{\alpha_1,\alpha_2<\mathcal{O},\mathcal{O}}(S_{\alpha_2} \not\subset \mathrm{G}(L_{\alpha_1}))_2 & x_2 &= \varepsilon_{x\in\mathrm{F}(S_{\kappa_5})}(x \notin \mathrm{G}(L_{\kappa_4}))
\end{aligned}
$$

Figure 10: Example of circular proof involving inductive and coinductive types.

6. Fixpoint-less recursion for Scott encoding

In this section, we are going to demonstrate the expressivity of our system by exhibiting typable, pure λ-calculus recursors for Scott encoded data types. Scott encoding is similar to Church encoding, but it relies on (co-)inductive types as well as polymorphism. As first examples, we are going to consider the Church and Scott encodings of natural numbers.

Although they have little (if any) practical interest, they demonstrate well the use of polymorphism and fixpoints. The type of Church numerals $\mathbb{N}_C$ and the type of Scott numerals $\mathbb{N}_S$ 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, we are able to define (and of course type-check using our implementation) the usual terms for predecessor P_C, recursor R_C, but also the less well-known Maurey infimum $(\leq)$, which requires inductive type [22]. The latter requires some type annotations for our implementation to guess the correct instantiation of unifications variables. In particular, the type $N_T = (\mathrm{T} \to \mathrm{T}) \to \mathrm{T} \to \mathrm{T}$ where $\mathrm{T} = \mu X.((X \to \mathbb{B}) \to \mathbb{B})$ must be used for natural numbers. Note that $T, F : \mathbb{B}$ denote booleans.

$$\begin{aligned}
P_C : \mathbb{N}_C \to \mathbb{N}_C &= \lambda n.n\ (\lambda p\, x{:}\mathbb{N}_C\, y{:}\mathbb{N}_C.p\ (S_C\ x)\ x)\ (\lambda x\, y.y)\ 0_C\ 0_C \\
R_C : \forall P.(P \to \mathbb{N}_C \to P) \to P \to \mathbb{N}_C \to P &= \lambda f\, a\, n.n\ (\lambda x\, p{:}\mathbb{N}_C.f\ (x\ (S_C\ p))\ p)\ (\lambda p.a)\ 0_C \\
(\leq) : \mathbb{N}_C \to \mathbb{N}_C \to \mathbb{B} &= \lambda n\, m.(n : N_T)\ (\lambda f\, g.g\ f)\ (\lambda i.T)\ ((m : N_T)\ (\lambda f\, g.g\ f)\ (\lambda i.F))
\end{aligned}$$

Scott numerals were initially introduced because they admit a constant time predecessor, whereas Church numerals do not. Usually, programming using Scott numerals requires the use of a recursor similar to that of Gödel's System T. Such a recursor can be easily programmed using general recursion, however this would require introducing typable terms that are not strongly normalising. In our type system, we can typecheck a strongly normalisable recursor due to Michel Parigot[30]. It is displayed below together with several terms and types involved in its definition.

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

It is easy to check that the term R_S is indeed a recursor for Scott numerals. It is similar to a λ-calculus fixpoint combinator but it only allows a limited number of unfoldings. As the recuror is typable, Theorem 7.25 implies that it is strongly normalising. The crutial point for typing the recursor is the subtyping relation $\mathbb{N}_S \subset \mathbb{N}'$, which is derivable in our system. It is however not clear what are the terms of type $\mathbb{N}'$ (that are not in $\mathbb{N}_S$). Note that the type annotation $n : \mathbb{N}'$ is required for type-checking R_S using our implementation, but we do not need to give the type of ζ or δ.

The recursor for 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.

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

Here, the existentially quantified type can be seen as the representation of the internal state of the stream. In particular, it must be provided to compute the first element or the tail of the stream. The order in which the fixpoint and the existential type is essential to allow the typing of "cons". Note that the internal state is also used to keep strong normalisation by introducing some laziness into the data type. In other words, a function call is required to compute the head or the tail of the stream. The definition of our stongly normalising coiterator coiter for streams is given bellow.

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

Note that in the definition of coiter we deliberately used the same names as in the definitions of R_S to highlight their similarities. The minimum type annotation for our implementation to type-check coiter involve the subtyping relation $\mathbb{S}'(\text{A}, \text{P}) \subset \mathbb{S}(\text{A})$. The let-binding syntax in coiter is used to name universally quantified types (see Section 10). It is only used in the implementation and it is not part of the theoretical type system. In particular, the types of ζ and δ are not required. As for Scott numeral, a question arise about the inhabitants of the type $\exists P.\mathbb{S}'(\text{A}, P)$.

The main difference between the encoding of Scott numerals and the encoding of streams is the use of native records. It is in fact possible to use native sums for encoding Scott numerals, but a function type is still required to program a strongly normalising recursor. We were not able to program a strongly normalisable recursor for the usual type of unary natural numbers $\mu N.[\text{Z} \mid \text{S of } N]$, and we conjecture that this is not possible. However, if we encode the sum type using a record type, a recursor can be given. The type of unary natural numbers then becomes $\mathbb{N} = \mu X.\forall Y.\{\text{z} : Y; \text{s} : X \to Y\} \to Y$, which is very similar to the type of Scott numeral.

7. Realisability semantics

In this section, we build a realisability model that is shown adequate with our type system. In particular, a formula A is interpreted as a set of strongly normalising *pure terms*

$\llbracket A\rrbracket$. Consequently, if $\vdash t : A$ is derivable then we will have $\llbracket t\rrbracket \in \llbracket A\rrbracket$, where $\llbracket t\rrbracket$ is the interpretation of t as a pure term.

Definition 7.1. A term is said to be *pure* if it does not contain subterms of the form $\varepsilon_{x\in A}(t \notin B)$. We denote $\llbracket\Lambda\rrbracket \subset \Lambda$ the set of pure terms (or pure individuals according to the terminology of Definition 4.3). A pure term $t \in \llbracket\Lambda\rrbracket$ is said to be strongly normalising if there is no infinite sequence of reduction starting from t using the rules of Figure 6. We denote $\mathcal{N} \subset \llbracket\Lambda\rrbracket$ the set of strongly normalising pure terms.

Definition 7.2. The set $\mathcal{H}$ of head contexts (i.e. terms with a hole in head position) is generated by the following grammar.

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

Given a term $t \in \Lambda$ and a context $H \in \mathcal{H}$, we denote $H[t]$ the term formed by plugging t into the hole of H. We extend naturally the notion of reduction to context by writing $H \succ H'$ when $H[t] \succ H'[t]$ for any term $t \in \Lambda$ (including, for instance, λ-variables). We denote $(\succ_H)$ the *head reduction* relation defined as the contextual closure of the rules of Figure 6, restricted to contexts of $\mathcal{H}$. We say that a term is in *head normal form* if it cannot be reduced using $(\succ_H)$.

Definition 7.3. We say that a set of pure terms $\Phi \subset \llbracket\Lambda\rrbracket$ is *saturated* if it is closed by head reduction[13] and if the following conditions hold.

(1) If $H[t[x := u]] \in \Phi$ and $u \in \mathcal{N}$ then $H[(\lambda x.t)\ u] \in \Phi$.
(2) If $H[t\ u] \in \Phi$, then $H[[D\ u \mid D \to t]] \in \Phi$.
(3) If $H[t] \in \Phi$ then $H[\{l = t; (l_i = t_i)_{i\in I}\}.l] \in \Phi$ provided that $t_i \in \mathcal{N}$ for all $i \in I$.
(4) If $H[t \mid (C_i \to t_i)_{i\in I}] \in \Phi$ and $I \subset J$ then $H[t \mid (C_i \to t_i)_{i\in J}] \in \Phi$ provided that $t_j \in \mathcal{N}$ for all $j \in J \setminus I$.

Lemma 7.4. *If $\Phi \subset \llbracket\Lambda\rrbracket$ is saturated, then for all $t \in \mathcal{N}$ and $u \in \Phi$, $t \succ_H u$ implies $t \in \Phi$.*

Proof. Immediate by the definitions of saturated sets and head reduction. □

Lemma 7.5. *The set $\mathcal{N}$ is saturated.*

Proof. The set $\mathcal{N}$ is obviously closed under head reduction, so it remains to show that it satisfies the four conditions of Definition 7.3.

(1) Let us take $H[t[x := u]] \in \mathcal{N}$ and suppose, by contradiction, that $H[(\lambda x.t)\ u] \notin \mathcal{N}$. There cannot be an infinite reduction of H, t or u. Hence, an infinite reduction of $H[(\lambda x.t)\ u]$ must start with $H[(\lambda x.t)\ u] \succ^* H'[(\lambda x.t')\ u'] \succ H'[t'[x := u']]$, where $H \succ^* H'$, $t \succ^* t'$ and $u \succ^* u'$. We then contradict $H[t[x := u]] \in \mathcal{N}$ by transforming this reduction into $H[(\lambda x.t)\ u] \succ H[t[x := u]] \succ^* H'[t'[x := u']]$.
(2) Let us take $H[t\,u] \in \mathcal{N}$ and suppose, by contradiction, that $H[[D\ u \mid D \to t]] \notin \mathcal{N}$. As in the previous case, there cannot be an infinite reduction of H, u or t. As a consequence, an infinite reduction of $H[[D\ u \mid D \to t]]$ necessarily starts with $H[[D\ u \mid D \to t]] \succ^* H'[[D\ u' \mid D \to t']] \succ H'[t'\,u']$, where $H \succ^* H'$, $t \succ^* t'$ and $u \succ^* u'$. This can be transformed into $H[[D\ u \mid D \to t]] \succ H[t\,u] \succ^* H'[t'\,u']$, which contradicts $H[t\,u] \in \mathcal{N}$.

[13] Requiring closure under head reduction is unusual, but necessary for subtyping on sum types.

(3) Let us take $H[t] \in \mathcal{N}$ and $t_i \in \mathcal{N}$ for all $i \in I$, and suppose, by contradiction, that $H[\{l = t; (l_i = t_i)_{i\in I}\}.l] \notin \mathcal{N}$. There cannot be an infinite reduction of H, t nor of any of the t_i. Consequently, an infinite reduction of $H[\{l = t; (l_i = t_i)_{i\in I}\}.l]$ must start with $H[\{l = t; (l_i = t_i)_{i\in I}\}.l] \succ^* H[\{l = t'; (l_i = t'_i)_{i\in I}\}.l]$, where $H \succ^* H'$, $t \succ^* t'$ and $t_i \succ^* t'_i$ for all $i \in I$. We then obtain a contradiction with $H[t] \in \mathcal{N}$ by transforming this reduction into $H[\{l = t; (l_i = t_i)_{i\in I}\}.l] \succ^* H[t] \succ H'[t']$.

(4) Let us take $H[[t \mid (C_i \to t_i)_{i\in I}]] \in \mathcal{N}$, a set of index J with $I \subset J$ and for all $j \in J \setminus I$ a term $t_j \in \mathcal{N}$. We suppose, by contradiction, that $H[[t \mid (C_i \to t_i)_{i\in J}]] \notin \mathcal{N}$. There cannot be an infinite sequence of reduction of H, t nor any of the t_j for $j \in J$. As a consequence, an infinite reduction of $H[[t \mid (C_i \to t_i)_{i\in J}]]$ necessarily starts with $H[[t \mid (C_i \to t_i)_{i\in J}]] \succ^* H'[[C_k u \mid (C_i \to t'_i)_{i\in J}]] \succ H'[t'_k\, u]$, where $H \succ^* H'$, $t \succ^* C_k u$ for some $k \in J$ and $t_i \succ^* t'_i$ for all $i \in J$. We can then obtain a contradiction using $H[[t \mid (C_i \to t_i)_{i\in I}]] \succ^* H'[[C_k u \mid (C_i \to t'_i)_{i\in I}]] \succ H'[t'_k\, u]$ if $k \in I$, and $H[[t \mid (C_i \to t_i)_{i\in I}]] \succ^* H'[[C_k u \mid (C_i \to t'_i)_{i\in I}]] \succ H'[\Omega]$ otherwise. □

Definition 7.6. The set of *neutral terms* $\mathcal{N}_0 \subset [\![\Lambda]\!]$ 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 $i \in \mathbb{N}$ and $t \in \mathcal{N}_0$ we have $t.l_i \in \mathcal{N}_0$,
(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 $[t \mid (C_i \to t_i)_{i\in I}] \in \mathcal{N}_0$.

Note that $\mathcal{N}_0$ is not saturated.

Definition 7.7. Given a set of pure valus $\Phi \subseteq [\![\Lambda]\!]$, we denote $\overline{\Phi} \subseteq [\![\Lambda]\!]$ the smallest saturated set containing Φ.

Lemma 7.8. *We have $\mathcal{N}_0 \subset \overline{\mathcal{N}}_0 \subset \mathcal{N}$.*

Proof. We obviously have $\mathcal{N}_0 \subset \overline{\mathcal{N}}_0$ and $\mathcal{N}_0 \subset \mathcal{N}$. Moreover, it is clear that the saturation operation is covariant. As a consequence, we have $\overline{\mathcal{N}}_0 \subset \overline{\mathcal{N}} = \mathcal{N}$. □

Definition 7.9. Given two sets Φ_1, $\Phi_2 \subset [\![\Lambda]\!]$ 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 7.10. *Let Φ_1, Φ_2, Ψ_1, $\Psi_2 \subseteq [\![\Lambda]\!]$ be sets of pure terms such that $\Phi_2 \subseteq \Phi_1$ and $\Psi_1 \subseteq \Psi_2$. We have $(\Phi_1 \Rightarrow \Psi_1) \subseteq (\Phi_2 \Rightarrow \Psi_2)$.*

Proof. Immediate by definition. □

Lemma 7.11. *We have $\mathcal{N}_0 \subseteq (\mathcal{N} \Rightarrow \mathcal{N}_0) \subseteq (\mathcal{N}_0 \Rightarrow \mathcal{N}) \subseteq \mathcal{N}$.*

Proof. By Lemma 7.8 we know that $\mathcal{N}_0 \subseteq \mathcal{N}$ and hence we obtain $(\mathcal{N} \Rightarrow \mathcal{N}_0) \subseteq (\mathcal{N}_0 \Rightarrow \mathcal{N})$ using Lemma 7.10. If we take $t \in \mathcal{N}_0$, then by definition $t\,u \in \mathcal{N}_0$ for all $u \in \mathcal{N}$. Therefore we obtain $\mathcal{N}_0 \subseteq (\mathcal{N} \Rightarrow \mathcal{N}_0)$. 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 $(\mathcal{N}_0 \Rightarrow \mathcal{N}) \subseteq \mathcal{N}$. □

In the semantics, a closed term $t \in \Lambda$ will be interpreted as a pure term $[\![t]\!] \in [\![\Lambda]\!]$ with the same structure. The choice operators t will be replaced by (possibly open) pure terms in $[\![t]\!]$. A formula $A \in \mathcal{F}$ will be interpreted by a saturated set of pure terms $[\![A]\!]$ such that $\overline{\mathcal{N}}_0 \subseteq [\![A]\!] \subseteq \mathcal{N}$. Note that a syntactic ordinals $\kappa \in \mathcal{O}$ will be interpreted by an actual ordinal $[\![\kappa]\!] \in [\![\mathcal{O}]\!]$ according to Section 2. Of course, the interpretation of syntactic ordinals will involve the interpretation of terms and formulas through abstract judgments. The interpretation of our three syntactic entities is thus defined mutually inductively, as was their syntax.

Definition 7.12. The set of every type interpretations $[\![\mathcal{F}]\!]$ is defined as follows, its elements will be called *reducibility candidates* (or simply *candidates*).

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

To simplify the definition of the semantics, we will extend the syntax of formulas with the elements of their domain of interpretation. We already used this technique in Section 2 for syntactic ordinals, and it will allow us to work only with closed syntactic elements. Most notably, we will use substitutions with elements of the semantics instead of relying on a semantical map for interpreting free variables.

Definition 7.13. The sets of *parametric terms* Λ^* and the set of *parametric formulas* $\mathcal{F}^*$ are formed by extending the syntax of formulas with the elements of $[\![\mathcal{F}]\!]$. Terms do not need to be extended directly, however, the definition of $\mathcal{F}^*$ impacts the definition of Λ^* since terms and formulas are defined mutually inductively.[14] A closed parametric term (resp. formula, resp. syntactic ordinal) is a parametric term (resp. formula, resp. syntactic ordinal) that does not contain free propositional variables nor free ordinal variables. Note however that λ-variables are allowed. This is due to the definition of $\mathcal{N}_0$.

Definition 7.14. The interpretation of a closed parametric term $t \in \Lambda^*$ (resp. closed parametric formula $A \in \mathcal{F}^*$) is defined to be a pure term $[\![t]\!] \in [\![\Lambda]\!]$ (resp. a set of pure terms $[\![A]\!] \in [\![\mathcal{F}]\!]$) defined inductively according to Figure 11 and Definition 2.6. Note that the semantics of terms, types and syntactic ordinals should be defined mutually inductively due to choice operators (or witnesses). In particular, the abstract judgments used in the definition of choice operators for ordinals are interpreted in the obvious way according to Definition 4.3.

In the interpretation of choice operators of the form $\varepsilon_{x\in A}(t \notin B)$, it is important that no λ-variable other that x is bound in t. This is enforced by a syntactic restriction given in Definition 5.1. Without this restriction, a term $[\![\lambda y.\varepsilon_{x\in A}(t \notin B)]\!]$ with y free in t would correspond to a function that is not always definable using a pure term. Thus, our model would have circular (and hence invalid) definitions. Note that the axiom of choice is required to interpret the choice operators.

It is also worth noting that the interpretation of the types of the form $\mu_\kappa F$ (resp. $\nu_\kappa F$) involves a union with $\overline{\mathcal{N}}_0$ (resp. an intersection with $\mathcal{N}$). It is required as otherwise we would obtain $[\![\mu_0 F]\!] = \emptyset$ (resp. $[\![\nu_0 F]\!] = \Lambda$) for the zero ordinal, and these sets are not proper candidates for the interpretation of formulas.

Lemma 7.15. *The semantical interpretation of terms, formulas and syntactic ordinals commutes with the substitution of the three kinds of variables. We thus have, for example,* $[\![t[X := A]]\!] = [\![t[X := [\![A]\!]]]\!]$ *or* $[\![A[\alpha := \kappa]]\!] = [\![A[\alpha := [\![\kappa]\!]]]\!]$.

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

Lemma 7.16. *For all candidates* $\Phi, \Psi \in [\![\mathcal{F}]\!]$*, we have* $(\Phi \Rightarrow \Psi) \in [\![\mathcal{F}]\!]$.

Proof. Since Φ and Ψ are candidates, we know $\overline{\mathcal{N}}_0 \subseteq \Phi \subseteq \mathcal{N}$ and $\overline{\mathcal{N}}_0 \subseteq \Psi \subseteq \mathcal{N}$. As a consequence, we can use Lemma 7.10 to obtain $(\mathcal{N} \Rightarrow \overline{\mathcal{N}}_0) \subseteq (\Phi \Rightarrow \Psi)$ (using $\Phi \subseteq \mathcal{N}$ and $\overline{\mathcal{N}}_0 \subseteq \Psi$) and $(\Phi \Rightarrow \Psi) \subseteq (\overline{\mathcal{N}}_0 \Rightarrow \mathcal{N})$ (using $\overline{\mathcal{N}}_0 \subseteq \Phi$ and $\Psi \subseteq \mathcal{N}$). We then obtain $\overline{\mathcal{N}}_0 \subseteq (\Phi \Rightarrow \Psi) \subseteq \mathcal{N}$ with Lemma 7.11. It remains to show that $(\Phi \Rightarrow \Psi)$ is saturated, so

[14] The set of parametric syntactic ordinals $\mathcal{O}^*$ of Definition 2.4 should also be impacted.

$$
\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]\!] \\ \text{any } t \in \mathcal{N}_0 \text{ if there is no such } u \end{cases} \\
[\![\Phi]\!] &= \Phi \ \text{ if } \Phi \in [\![\mathcal{F}]\!] \\
[\![A \to B]\!] &= ([\![A]\!] \Rightarrow [\![B]\!]) \\
[\![\{(l_i : A_i)_{i\in I}; \ldots\}]\!] &= \{t \in \mathcal{N} \mid \forall i \in I, t.l_i \in [\![A_i]\!]\} \\
[\![\{(l_i : A_i)_{i\in I\neq\emptyset}\}]\!] &= \{t \in \mathcal{N} \mid \forall i \in I, t.l_i \in [\![A_i]\!] \text{ and } \forall i \notin I, t.l_i \succ_H^* \Omega\} \\
[\![\{\}]\!] &= \overline{\mathcal{N}_0 \cup \{\{\}\}} \\
[\![[(C_i : A_i)_{i\in I}]]\!] &= \{t \in \mathcal{N} \mid \forall \Phi \in [\![\mathcal{F}]\!], \forall (t_i \in ([\![A_i]\!] \Rightarrow \Phi))_{i\in I}, [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]\!] &= \left(\cup_{o<[\![\kappa]\!]}[\![A[X := \mu_o X A]]\!]\right) \cup \overline{\mathcal{N}}_0 \\
[\![\nu_\kappa X.A]\!] &= \left(\cap_{o<[\![\kappa]\!]}[\![A[X := \nu_o X A]]\!]\right) \cap \mathcal{N} \\
[\![\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 11: Semantical interpretation of closed parametric terms and types.

we will first show that it is closed under head reduction. Let us take $t \in (\Phi \Rightarrow \Psi)$ such that $t \succ_H t'$ and show that $t' \in (\Phi \Rightarrow \Psi)$. We take $u \in \Phi$ and show $t'\ u \in \Psi$. Since Ψ is closed under head reduction and $t\ u \succ_H t'\ u$ it is enough to show $t\ u \in \Psi$, which follows from the definition of $t \in (\Phi \Rightarrow \Psi)$. It remains to prove the four saturation conditions.

(1) Let us suppose that $H[t[x := u]] \in (\Phi \Rightarrow \Psi)$ and that $u \in \mathcal{N}$. We need to show that $H[(\lambda x.t)\ u] \in (\Phi \Rightarrow \Psi)$ so we take $v \in \Phi \subseteq \mathcal{N}$ and we prove $H[(\lambda x.t)\ u]\ v \in \Psi$. As we have $H[t[x := u]] \in (\Phi \Rightarrow \Psi)$ we know that $H[t[x := u]]\ v \in \Psi$. We can thus conclude using the saturation condition (1) on Ψ with the context $H\ v$.

(2) We now suppose $H[t\ u] \in (\Phi \Rightarrow \Psi)$ and show $H[[D\ u \mid D \to t]] \in (\Phi \Rightarrow \Psi)$. We thus take $v \in \Phi \subseteq \mathcal{N}$ and we prove $H[[D\ u \mid D \to t]]\ v \in \Psi$. As $H[t\ u] \in (\Phi \Rightarrow \Psi)$ we know that $H[t\ u]\ v \in \Psi$ and thus we can conclude using the saturation condition (2) of Ψ with the context $H\ v$.

(3) Let us now suppose that $H[t] \in (\Phi \Rightarrow \Psi)$ and that $t_i \in \mathcal{N}$ for all $i \in I$. We need to show that $H[\{l = t; (l_i = t_i)_{i\in I}\}.l] \in (\Phi \Rightarrow \Psi)$ so we take $v \in \Phi \subseteq \mathcal{N}$ and we prove

$H[\{l = t; (l_i = t_i)_{i\in I}\}.l]\ v \in \Psi$. As $H[t] \in (\Phi \Rightarrow \Psi)$ we know that $H[t]\ v \in \Psi$ and thus conclude with the saturation condition (3) of Ψ with the context $H\ v$.

(4) We now suppose $H[t \mid (C_i \to t_i)_{i\in I}] \in (\Phi \Rightarrow \Psi)$ and $I \subset J$ with $t_j \in \mathcal{N}$ for all $j \in J \setminus I$. We need to show $H[t \mid (C_i \to t_i)_{i\in J}] \in (\Phi \Rightarrow \Psi)$ so we take $v \in \Phi \subseteq \mathcal{N}$ and we prove $H[t \mid (C_i \to t_i)_{i\in J}]\ v \in \Psi$. As we have $H[t \mid (C_i \to t_i)_{i\in I}] \in (\Phi \Rightarrow \Psi)$ we know that $H[t \mid (C_i \to t_i)_{i\in I}]\ v \in \Psi$ and thus we can conclude using the saturation condition (4) of Ψ with the context $H\ v$. □

Lemma 7.17. *If for all $i \in I$ we have $\Phi_i \in [\![\mathcal{F}]\!]$ then $[\![[(C_i \text{ of } \Phi_i)_{i\in I}]]\!] \in [\![\mathcal{F}]\!]$.*

Proof. By definition, it is easy to see that $[\![[(C_i \text{ of } \Phi_i)_{i\in I}]]\!] \subseteq \mathcal{N}$. Let us take $t \in \overline{\mathcal{N}}_0$ and show that $t \in [\![[(C_i \text{ of } \Phi_i)_{i\in I}]]\!]$. We thus take $\Phi \in [\![\mathcal{F}]\!]$ and $t_i \in (\Phi_i \Rightarrow \Phi)$ for all $i \in I$, and we show $[t \mid (C_i \to t_i)_{i\in I}] \in \Phi$. This is immediate as we have $[t \mid (C_i \to t_i)_{i\in I}] \in \overline{\mathcal{N}}_0$ and $\overline{\mathcal{N}}_0 \subseteq \Phi$. It remains to show that $[\![[(C_i \text{ of } \Phi_i)_{i\in I}]]\!]$ is saturated, so we will first show that it is closed under head reduction. Let us take $t \in [\![[(C_i \text{ of } \Phi_i)_{i\in I}]]\!]$ such that $t \succ_H t'$ and show that $t' \in [\![[(C_i \text{ of } \Phi_i)_{i\in I}]]\!]$. We thus take $\Phi \in [\![\mathcal{F}]\!]$ and $t_i \in (\Phi_i \Rightarrow \Phi)$ for all $i \in I$, and we show $[t' \mid (C_i \to t_i)_{i\in I}] \in \Phi$. Since $t \in [\![[(C_i \text{ of } \Phi_i)_{i\in I}]]\!]$ we know that $[t \mid (C_i \to t_i)_{i\in I}] \in \Phi$. We thus conclude since Φ is saturated and $[t \mid (C_i \to t_i)_{i\in I}] \succ_H [t' \mid (C_i \to t_i)_{i\in I}]$. It remains to prove the four saturation conditions.

(1) Let us suppose that we have $H[t[x := u]] \in [\![[(C_i \text{ of } \Phi_i)_{i\in I}]]\!]$ and $u \in \mathcal{N}$, and show $H[(\lambda x.t)\ u] \in [\![[(C_i \text{ of } \Phi_i)_{i\in I}]]\!]$. We take $\Phi \in [\![\mathcal{F}]\!]$ and $t_i \in (\Phi_i \Rightarrow \Phi)$ for all $i \in I$, and show $[H[(\lambda x.t)\ u] \mid (C_i \to t_i)_{i\in I}] \in \Phi$. Since $H[t[x := u]] \in [\![[(C_i \text{ of } \Phi_i)_{i\in I}]]\!]$ we have $[H[t[x := u]] \mid (C_i \to t_i)_{i\in I}] \in \Phi$. We can thus conclude using the saturation condition (1) on Φ with the context $[H \mid (C_i \to t_i)_{i\in I}]$.

(2) We suppose $H[t\ u] \in [\![[(C_i \text{ of } \Phi_i)_{i\in I}]]\!]$ and show $H[[D\ u \mid D \to t]] \in [\![[(C_i \text{ of } \Phi_i)_{i\in I}]]\!]$. We thus take $\Phi \in [\![\mathcal{F}]\!]$ and $t_i \in (\Phi_i \Rightarrow \Phi)$ for all $i \in I$, and show that we have $[H[[D\ u \mid D \to t]] \mid (C_i \to t_i)_{i\in I}] \in \Phi$. As $H[t\ u] \in [\![[(C_i \text{ of } \Phi_i)_{i\in I}]]\!]$ we know that $[H[t\ u] \mid (C_i \to t_i)_{i\in I}] \in \Phi$ and thus we can conclude using the saturation condition (2) of Φ with the context $[H \mid (C_i \to t_i)_{i\in I}]$.

(3) Let us now suppose that $H[t] \in [\![[(C_i \text{ of } \Phi_i)_{i\in I}]]\!]$ and that $t_i \in \mathcal{N}$ for all $i \in I$. We need to show that $H[\{l = t; (l_i = t_i)_{i\in I}\}.l] \in [\![[(C_i \text{ of } \Phi_i)_{i\in I}]]\!]$ so we take $\Phi \in [\![\mathcal{F}]\!]$ and $t_i \in (\Phi_i \Rightarrow \Phi)$ for all $i \in I$, and show $[H[\{l = t; (l_i = t_i)_{i\in I}\}.l] \mid (C_i \to t_i)_{i\in I}] \in \Phi$. As $H[t] \in [\![[(C_i \text{ of } \Phi_i)_{i\in I}]]\!]$ we know that $[H[t] \mid (C_i \to t_i)_{i\in I}] \in \Phi$ and thus conclude with the saturation condition (3) of Φ with the context $[H \mid (C_i \to t_i)_{i\in I}]$.

(4) We now suppose $H[t \mid (C_i \to t_i)_{i\in I}] \in [\![[(C_i \text{ of } \Phi_i)_{i\in I}]]\!]$ and $I \subset J$ with $t_j \in \mathcal{N}$ for all $j \in J \setminus I$. We need to show $H[t \mid (C_i \to t_i)_{i\in J}] \in [\![[(C_i \text{ of } \Phi_i)_{i\in I}]]\!]$ so we take $\Phi \in [\![\mathcal{F}]\!]$ and $t_i \in (\Phi_i \Rightarrow \Phi)$ for all $i \in I$, and show $[H[t \mid (C_i \to t_i)_{i\in J}] \mid (C_i \to t_i)_{i\in I}] \in \Phi$. As $H[t \mid (C_i \to t_i)_{i\in I}] \in [\![[(C_i \text{ of } \Phi_i)_{i\in I}]]\!]$ we get $[H[t \mid (C_i \to t_i)_{i\in I}] \mid (C_i \to t_i)_{i\in I}] \in \Phi$ and we can then use the saturation condition (4) of Φ with the context $[H \mid (C_i \to t_i)_{i\in I}]$. □

Lemma 7.18. *If for all $i \in I$ we have $\Phi_i \in [\![\mathcal{F}]\!]$ then $[\![\{(l_i : \Phi_i)_{i\in I}; \dots\}]\!] \in [\![\mathcal{F}]\!]$.*

Proof. Similar to the proofs of Lemmas 7.16 and 7.17. □

Lemma 7.19. *If for all $i \in I$ we have $\Phi_i \in [\![\mathcal{F}]\!]$ then $[\![\{(l_i : \Phi_i)_{i\in I}\}]\!] \in [\![\mathcal{F}]\!]$.*

Proof. Immediate if $I = \emptyset$ and similar to the proofs of Lemmas 7.16 and 7.17 otherwise. □

Theorem 7.20. *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. For instance, if $u = [\![\varepsilon_{x\in A}(t \notin B)]\!]$ then we have $u \in [\![A]\!] \subseteq \mathcal{N} \subseteq [\![\Lambda]\!]$ by induction hypothesis, or $u \in \overline{\mathcal{N}}_0 \subseteq [\![\Lambda]\!]$. For ordinals, the proof is immediate by Definition 2.6 and using the induction hypothesis to interpret predicates in ordinal witnesses. For types of the form $\Phi \subseteq [\![\mathcal{F}]\!]$, $\varepsilon_X(t \in A)$ or $\varepsilon_X(t \notin A)$ the proof is immediate. For types of the form $A \Rightarrow B$, $[(C_i \text{ of } A_i)_{i\in I}]$, $\{(l_i : A_i)_{i\in I}; \dots\}$ or $\{(l_i : A_i)_{i\in I}\}$ then we respectively use Lemma 7.16, 7.17, 7.18 or 7.19 with the induction hypotheses and Lemma 7.15. The remaining four possible forms of types are treated bellow.

- For types of the form $\forall X.A$, the induction hypothesis gives $[\![A[X := \Phi]]\!] \in [\![\mathcal{F}]\!]$ for all $\Phi \in [\![\mathcal{F}]\!]$. We can then conclude using the fact that an intersection of candidates is itself a candidate.
- For types of the form $\exists X.A$, the proof is similar to the previous case, using the fact that a union of candidates is itself a candidate.
- For types of the form $\mu_\kappa X.A$, we show $[\![\mu_o X.A]\!] \in [\![\mathcal{F}]\!]$ for all $o \leq [\![\kappa]\!]$ by induction on the ordinal o. This is enough as we can then conclude using Lemma 7.15 to show $[\![\mu_\kappa X.A]\!] = [\![\mu_{[\![\kappa]\!]} X.A]\!] = [\![\mu_o X.A]\!] \in [\![\mathcal{F}]\!]$. If $o = 0$ then we have $[\![\mu_0 X.A]\!] = \overline{\mathcal{N}}_0$ and the proof is thus immediate. Otherwise, we have $[\![\mu_o X.A]\!] = \cup_{o'<o}[\![A[X := \mu_{o'} X.A]]\!]$. Using the local induction hypothesis we get $[\![\mu_{o'} X.A]\!] \in [\![\mathcal{F}]\!]$ for all $o' < o$. Using Lemma 7.15, we then obtain $[\![A[X := \mu_{o'} X.A]]\!] = [\![A[X := [\![\mu_{o'} X.A]\!]]]\!]$ for all $o' < o$, which gives $[\![A[X := [\![\mu_{o'} X.A]\!]]]\!] \in [\![\mathcal{F}]\!]$ for all $o' < o$ using the global induction hypothesis. We can then conclude using again the fact that a union of candidates is itself a candidate.
- For types of the form $\nu_\kappa X.A$, we proceed in a similar way as in the previous case, using again the fact that an intersection of candidates is itself a candidate. Note that we have $[\![\nu_0 X.A]\!] = \mathcal{N} \in [\![\mathcal{F}]\!]$ in the case of the zero ordinal. □

Before going into our main soundness theorem, we need to show that the elements of sum types behave in the expected way. In other words, such a term sould reduce to either a neutral term (i.e., a term in $\overline{\mathcal{N}}_0$) or to a constructor. Although the semantics of our sum types involve arrows, we still obtain this result thanks to parametricity. This is why the codomain of the arrows is quantified over universally in the interpretation of sum types.

Lemma 7.21. *Every strongly normalising pure term $t \in \mathcal{N}$ has a head normal form that is either a λ-abstraction, a record, a constructor or a term in $\mathcal{N}_0$.*

Proof. The head normal form of a pure term can be written $H[u]$ where u is either a λ-abstraction, a record, a constructor or a λ-variable. If $H = [\,]$ then we can conclude immediately. If $H \neq [\,]$ then we must have $u = x$, which implies $H[u] \in \mathcal{N}_0$, as in every other cases $H[u]$ can be reduced. □

Lemma 7.22. *If $[\![A_i]\!] \in [\![\mathcal{F}]\!]$ for all $i \in I$, then we have $t \in [\![[(C_i \text{ of } A_i)_{i\in I}]]\!]$ if and only if $t \in \mathcal{N}$ and either $t \succ_H^* v$ with $v \in \mathcal{N}_0$ or $t \succ_H^* C_k\, v$ with $k \in I$ and $v \in [\![A_k]\!]$.*

Proof. ($\Rightarrow$) Let us suppose that $t \in [\![[(C_i \text{ of } A_i)_{i\in I}]]\!]$. By definition, we immediately have $t \in \mathcal{N}$, so according to Lemma 7.21 there is a head normal form v such that $t \succ_H^* v$, and we only need to show that v cannot be a λ-abstraction, a record, a term of the form $C_k\, u$ with $k \notin I$, or a term of the form $C_k\, u$ with $k \in I$ and $u \notin [\![A_k]\!]$. To rule out the first three possibilities, we apply the definition of $[\![[(C_i \text{ of } A_i)_{i\in I}]]\!]$ using the fact that $\lambda x.x \in ([\![A_i]\!] \Rightarrow \mathcal{N})$ for all $i \in I$ to obtain $[t \mid (C_i \to \lambda x.x)_{i\in I}] \in \mathcal{N}$. We thus have $[v \mid (C_i \to \lambda x.x)_{i\in I}] \in \mathcal{N}$ since $t \succ^* v$, but this term diverges if v has one of the first three forms. Let us now suppose that there is $k \in I$ such that $v = C_k\, u$. We consider the term $u_k = [t \mid C_k \to \lambda x.x \mid (C_i \to \lambda x.y)_{i\in I\setminus\{k\}}]$ where y is a fresh variable. Obviously, we have $\lambda x.x \in ([\![A_k]\!] \Rightarrow [\![A_k]\!])$ and $\lambda x.y \in ([\![A_i]\!] \Rightarrow \mathcal{N}_0) \subseteq ([\![A_i]\!] \Rightarrow [\![A_k]\!])$ for all $i \in I \setminus \{k\}$. Therefore, we can use the definition of $[\![[(C_i \text{ of } A_i)_{i\in I}]]\!]$ to obtain $u_k \in [\![A_k]\!]$. We can then conclude that $C_k\, u \in [\![A_k]\!]$ as $[\![A_k]\!]$ is saturated and $u_k \succ_H^* C_k\, u$.

($\Leftarrow$) Let us now suppose that $t \in \mathcal{N}$ and that $t \succ_H^* v$ with either $v \in \mathcal{N}_0$ or $v = C_k\, u$ with $k \in I$ and $u \in [\![A_k]\!]$. We need to show $t \in [\![[(C_i \text{ of } A_i)_{i\in I}]]\!]$, so we take a set $\Phi \in [\![\mathcal{F}]\!]$, terms $t_i \in ([\![A_i]\!] \Rightarrow \Phi)$ for all $i \in I$, and we show $[t \mid (C_i \to t_i)_{i\in I}] \in \Phi$. Since $t \succ_H^* v$ we also have $[t \mid (C_i \to t_i)_{i\in I}] \succ_H^* [v \mid (C_i \to t_i)_{i\in I}]$ and thus it is enough to show $[v \mid (C_i \to t_i)_{i\in I}] \in \Phi$ according to Lemma 7.4. Now, if $v \in \mathcal{N}_0$ then we have $[v \mid (C_i \to t_i)_{i\in I}] \in \mathcal{N}_0$ and we can conclude immediately. If $v = C_k\, u$ with $k \in I$ and $u \in [\![A_k]\!]$, then we need to show $t_k\, u \in [\![A_k]\!]$, which follows from $t_k \in ([\![A_k]\!] \Rightarrow \Phi)$. □

We will now prove our main soundness theorem, the so-called *adequacy lemma*. Note that the definition of saturation and the previous lemmas give exactly the properties required for the proof of this theorem. In fact, it is possible to gather the required properties by attempting to construct the proof.

Theorem 7.23. *Let γ be an ordinal context such that $[\![\tau]\!] > 0$ for all $\tau \in \gamma$.*

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

Proof. According to Theorem 4.14 we only have to prove that our typing and subtyping rules are correct. Note that the truth of our abstract judgments $[\![t : A]\!] = 1$ and $[\![t \in A \subset B]\!] = 1$ is defined according to the statement of the current theorem. We thus consider all the rules of Figure 12 and 8.

- ($\to_i$) We need to show $[\![\lambda x.t]\!] \in [\![C]\!]$. However, according to the second induction hypothesis, it is enough to show $[\![\lambda x.t]\!] \in [\![A \to B]\!] = ([\![A]\!] \Rightarrow [\![B]\!])$. Using the second induction hypothesis we have $[\![t[x := \varepsilon_{x\in A}(t \notin B)]]\!] \in [\![B]\!]$. By definition of the choice operator, this means that we have $[\![t[x := u]]\!] = [\![t]\!][x := u] \in [\![B]\!]$ for all $u \in [\![A]\!]$. By Theorem 7.20 we know that $[\![B]\!]$ is saturated and that $[\![A]\!] \subseteq \mathcal{N}$. We then use the saturation condition (1) to get $[\![\lambda x.t]\!]\, u \in [\![B]\!]$ for all $u \in [\![A]\!]$.
- ($\to_e$) We need to show $[\![t]\!]\ [\![u]\!] \in [\![B]\!]$. By induction hypothesis we have $[\![t]\!] \in [\![A \to B]\!]$ and $[\![u]\!] \in [\![A]\!]$, so we can conclude by definition of $[\![A \to B]\!] = ([\![A]\!] \Rightarrow [\![B]\!])$.
- (ε) We need to show $[\![\varepsilon_{x\in A}(t \notin B)]\!] \in [\![C]\!]$. However, according to the induction hypothesis, it is enough to show $[\![\varepsilon_{x\in A}(t \notin B)]\!] \in [\![A]\!]$. This follows immediately from the definition of $[\![\varepsilon_{x\in A}(t \notin B)]\!]$. In particular, $\mathcal{N}_0 \subseteq [\![A]\!]$ by Theorem 7.20.
- ($\times_i$) We need to show that $[\![\{(l_i = t_i)_{i\in I}\}]\!] \in [\![B]\!]$. According to the first induction hypothesis, it is enough to show $[\![\{(l_i = t_i)_{i\in I}\}]\!] \in [\![\{(l_i : A_i)_{i\in I}\}]\!]$. By definition, we need to take $k \in I$ and show $\{(l_i = [\![t_i]\!])_{i\in I}\}.l_k \in [\![A_k]\!]$. By induction hypothesis we know that $[\![t_k]\!] \in [\![A_k]\!]$, hence we can use the saturation condition (3) on $[\![A_k]\!]$ since it is saturated by Theorem 7.20. Note that if $k \notin I$ then we immediately have $\{(l_i = [\![t_i]\!])_{i\in I}\}.l_k \succ_H^* \Omega$.
- ($\times_e$) We need to show $[\![t.l_k]\!] = [\![t]\!].l_k \in [\![A]\!]$. As we have $[\![t]\!] \in [\![\{l_k : A; ...\}]\!]$ by induction hypothesis, we can conclude by definition of $[\![\{l_k : A; ...\}]\!]$.

($+_i$) We need to show $[\![C_k\,t]\!] \in [\![B]\!]$. According to the first induction hypothesis, it is enough to show $[\![C_k\,t]\!] = C_k\,[\![t]\!] \in [\![[C_k \text{ of } A]]\!]$. By definition, we need to take $\Phi \in [\![\mathcal{F}]\!]$, $t_k : ([\![A]\!] \Rightarrow \Phi)$ and show $[C_k\,[\![t]\!] \mid C_k \to t_k] \in \Phi$. Using the saturation condition (2) on Φ, it is enough to show $t_k\,[\![t]\!] \in \Phi$. This follows by definition of $([\![A]\!] \Rightarrow \Phi)$ since $[\![t]\!] \in [\![A]\!]$ according to the second induction hypothesis.

($+_e$) We need to show $[\![[t \mid (C_i \to t_i)_{i\in I}]]\!] \in [\![B]\!]$. By the first induction hypothesis, we know that $[\![t]\!] \in [\![[(C_i \text{ of } A_i)_{i\in I}]]\!]$. We can thus conclude by definition of $[\![[(C_i \text{ of } A_i)_{i\in I}]]\!]$, using the remaining induction hypotheses.

($\to$) Let us suppose that $[\![t]\!] \in [\![A_1 \to B_1]\!]$, and assume that $[\![t]\!] \notin [\![A_2 \to B_2]\!]$ by contradiction. By definition of $[\![A_2 \to B_2]\!] = ([\![A_2]\!] \Rightarrow [\![B_2]\!])$ there must be $u \in [\![A_2]\!]$ such that $[\![t]\!]\ u \notin [\![B_2]\!]$. As a consequence, the term $v = [\![\varepsilon_{x\in A_2}(t\ x \notin B_2)]\!]$ must satisfy $v \in [\![A_2]\!]$ and $[\![t]\!]\ v \notin [\![B_2]\!]$ by definition of the choice operator. By the first induction hypothesis we have $v \in [\![A_1]\!]$, and hence $[\![t]\!]\ v \in [\![B_1]\!]$ by definition of $t \in [\![A_1 \to B_1]\!]$. Using the second induction hypothesis this gives $[\![t]\!]\ v \in [\![B_2]\!]$, which is a contradiction.

($=$) This is a trivial implication.

($\forall_l$) We assume $[\![t]\!] \in [\![\forall X.A]\!]$, and we show $[\![t]\!] \in [\![B]\!]$. Using the induction hypothesis, it is enough to show $[\![t]\!] \in [\![A[X := U]]\!]$, which is equivalently to $[\![t]\!] \in [\![A[X := [\![U]\!]]]\!]$ according to Lemma 7.15. By definition of $[\![\forall X.A]\!]$, we have $[\![t]\!] \in [\![A[X := \Phi]]\!]$ for all $\Phi \in [\![\mathcal{F}]\!]$. We can thus conclude as $[\![U]\!] \in [\![\mathcal{F}]\!]$ by Theorem 7.20.

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

($\exists_r$) Similar to the ($\forall_l$) case.

($\exists_l$) Similar to the ($\forall_r$) case.

($\times_s$) We assume $[\![t]\!] \in [\![\{(l_i : A_i)_{i\in I}\}]\!]$ and we show $[\![t]\!] \in [\![\{(l_i : B_i)_{i\in I}\}]\!]$. We can assume that $I \neq \emptyset$ as otherwise the proof is trivial. By definition of $[\![\{(l_i : A_i)_{i\in I}\}]\!]$, we know that $[\![t]\!].l_i \succ_H^* \Omega$ for all $i \notin I$. Thus, by definition of $[\![\{(l_i : B_i)_{i\in I}\}]\!]$, it only remains to take $k \in I$ and show $[\![t]\!].l_k \in [\![B_k]\!]$. This follows from the induction hypothesis since $[\![t]\!].l_k \in [\![A_k]\!]$ by definition of $[\![\{(l_i : A_i)_{i\in I}\}]\!]$.

($\times_{se}$) Similar to the ($\times_s$) case.

($\times_e$) Also similar to the ($\times_s$) case.

($+$) We assume $[\![t]\!] \in [\![[(C_i \text{ of } A_i)_{i\in I_1}]]\!]$ and we show $[\![t]\!] \in [\![[(C_i \text{ of } B_i)_{i\in I_2}]]\!]$. According to Lemma 7.22, we know that $t \succ_H^* v$ with only two possibilities for v. In the case where $v \in \overline{\mathcal{N}}_0$ then we can conclude directly using Lemma 7.22 in the other direction. Otherwise, we know that $t \succ_H^* C_k\,u$ with $k \in I_1 \subseteq I_2$ and $u \in [\![A_k]\!]$. We now consider the term $t.C_k$, which reduces as $t.C_k \succ_H^* [C_k\,u \mid C_k \to \lambda x.x] \succ_H u$. Since $t \in \mathcal{N}$, we can use Lemma 7.4 to deduce that $t.C_k \in [\![A_k]\!]$. Hence, we obtain $t.C_k \in [\![B_k]\!]$ by induction hypothesis. By Theorem 7.20 we know that $[\![B_k]\!]$ is saturated (and thus closed under head reduction). As a consequence, we can deduce $u \in [\![B_k]\!]$. We can then conclude using (the right to left direction of) Lemma 7.22.

(μ_r) We assume $[\![t]\!] \in [\![A]\!]$ and we show $[\![t]\!] \in [\![\mu_\kappa F]\!]$. By the first induction hypothesis we obtain that $[\![t]\!] \in [\![F(\mu_\tau F)]\!]$, so it only remains to show $[\![F(\mu_\tau F)]\!] \subseteq [\![\mu_\kappa F]\!]$. According to the second induction hypothesis, using Lemma 2.12, we know that $[\![\tau]\!] < [\![\kappa]\!]$. We thus obtain $[\![F(\mu_{[\![\tau]\!]} F)]\!] \subseteq [\![\mu_{[\![\kappa]\!]} F]\!]$ by definition of $[\![\mu_{[\![\kappa]\!]} F]\!]$. We then obtain $[\![F(\mu_\tau F)]\!] = [\![F(\mu_{[\![\tau]\!]} F)]\!] \subseteq [\![\mu_{[\![\kappa]\!]} F]\!] = [\![\mu_\kappa F]\!]$ using Lemma 7.15 twice.

(ν_l) Similar to the (μ_r) case.

(μ_r^∞) We assume $[\![t]\!] \in [\![A]\!]$ and we show $[\![t]\!] \in [\![\mu_\infty F]\!]$. By induction hypothesis, we obtain $[\![t]\!] \in [\![F(\mu_\infty F)]\!]$ so we only need to show $[\![F(\mu_\infty F)]\!] \subseteq [\![\mu_\infty F]\!]$. Since the cardinal of the ordinal $[\![\infty]\!]$ is 2^{2^ω}, it is larger than the cardinal of $[\![\mathcal{F}]\!]$ which is 2^ω. Hence the inductive definition of $[\![\mu_{[\![\infty]\!]} F]\!]$ must reach its stationary point strictly before $[\![\infty]\!]$.[15] As a consequence, we have $[\![\mu_{[\![\infty]\!]} F]\!] = [\![\mu_{[\![\infty]\!]+1} F]\!] \supseteq [\![F(\mu_{[\![\infty]\!]} F)]\!]$ by definition. We can thus conclude using Lemma 7.15 on both sides.

(ν_l^∞) Similar to the (μ_r^∞) case.

(μ_l) Let us suppose that $[\![t]\!] \in [\![\mu_\kappa F]\!]$ and show that $[\![t]\!] \in [\![B]\!]$. If $[\![\kappa]\!] = 0$ then this is immediate since in this case we have $[\![\mu_\kappa F]\!] = \overline{\mathcal{N}}_0$, and thus $[\![t]\!] \in [\![B]\!]$ since $\overline{\mathcal{N}}_0 \subseteq [\![B]\!]$ according to Theorem 7.20. If $[\![\kappa]\!] \neq 0$ then by definition there must be $o < [\![\kappa]\!]$ such that $[\![t]\!] \in [\![F(\mu_o F)]\!]$. By definition of the choice operator, this means that $o = [\![\varepsilon_{\alpha<\kappa}(t \in F(\mu_\alpha F))]\!]$ does verify $o < [\![\kappa]\!]$ and $[\![t]\!] \in [\![F(\mu_o F)]\!]$. We can thus conclude using the induction hypothesis.

(ν_r) Similar to the (μ_l) case. □

Intuitively, the adequacy lemma establishes the compatibility of our semantics with our type system. We will now rely on this theorem to obtain results such as consistency, strong normalisation or weak forms of type safety.

Theorem 7.24. *There is no closed, pure term t such that $\vdash t : \forall X.X$ or $\vdash t : [\,]$ is derivable.*

Proof. Let us assume that there is such a term t. According to the adequacy lemma (Theorem 7.23), we have $[\![t]\!] \in \overline{\mathcal{N}}_0$ since $[\![\forall X.X]\!] = [\![[\,]]\!] = \overline{\mathcal{N}}_0$ by definition. This is a contradiction since $\overline{\mathcal{N}}_0$ only contains open terms. □

Theorem 7.25. *Given a closed, pure term $t \in [\![\Lambda]\!]$ and a closed type $A \in \mathcal{F}$, if $\vdash t : A$ is derivable then t is strongly normalising.*

Proof. Using the adequacy lemma (Theorem 7.23), we know that $[\![t]\!] \in [\![A]\!]$. However, since t is pure we have $[\![t]\!] = t$. Moreover, according to Theorem 7.20 we have $[\![A]\!] \subseteq \mathcal{N}$, and thus we obtain $t \in \mathcal{N}$. □

Note that, as a direct consequence of strong normalisation, we know that a well-typed term cannot produce a runtime error. Indeed, the reduction rules of Figure 6 introduce a non-terminating term in case of an error (e.g., the projection of a λ-abstraction). We will now consider a stronger safety result, which will apply to so-called *simple* data types. They will cover most of the common inductive datatypes such as lists or binary trees.

Definition 7.26. We say that a type $A \in \mathcal{F}$ is *simple* if it is closed, and if it only contains sums, strict products and least fixpoints carrying the ∞ ordinal. Moreover, we will assume that a simple type A does not have two consecutive least fixpoints, and that the body of fixpoints is not limited to a variable (like in $\mu X.Y$ or $\mu X.X$).

Theorem 7.27. *If $\vdash t : A$ is derivable for a closed, pure term t and a simple type A, then t reduces to a normal form u such that $\vdash u : A$ is derivable.*

Proof. According to Theorem 7.25, we know that t must reduce to a normal form u. Moreover, u is closed since no free variables are introduced by our reduction rules. We proceed by induction on the size of u. In the case where $A = \mu X.B$ we know that $[\![B[X := A]]\!] \subseteq [\![A]\!]$. Let us define $A' = B[X := A]$ if $A = \mu X.B$ and $A' = A$ otherwise. The hypotheses on least

[15] This stationary point is not a fixpoint if F is not covariant, but we do not need this information.

fixpoints are still true in A' since B cannot be equal to X by hypothesis. Moreover, since pure types may not contain two consecutive fixpoints and A cannot be $\mu X.X$, A' is either a sum type or a strict product type.

If $A' = [(C_i : A_i)_{i\in I}]$ then, by Lemma 7.22 we know that $u = C_k\, v$ with $k \in I$ and $v \in [\![A_k]\!]$. In particular, u is in normal form (and thus in head normal form) and it cannot be open, which means that $u \notin \overline{\mathcal{N}}_0$. Since u is in normal form, we know that v is also in normal form. The induction hypothesis provides us with a derivation of $\vdash v : A_k$. In the case where $A' = A = [(C_i : A_i)_{i\in I}]$ then we can conclude using the follwing derivation.

$$\dfrac{\dfrac{\{k\} \subseteq I \qquad \vdash t.C_k \in A_k \subset A_k}{\vdash C_k\, v \in [C_k : A_k] \subset [(C_i : A_i)_{i\in I}]}\,+ \qquad \vdash v : A_k}{\vdash C_k\, v : [(C_i : A_i)_{i\in I}]}\,+_i$$

Otherwise, if we have $A = \mu X.[(C_i : A_i)_{i\in I}]$ then $A' = [(C_i : A_i[X := A])_{i\in I}]$ and we can construct the following derivation. Note that in this case, A_k is rather of the form $A_k[X := A]$, so we in fact have a proof of $\vdash v : A_k[X := A]$.

$$\dfrac{\dfrac{\dfrac{\{k\} \subseteq I \qquad \vdash t.C_k \in A_k[X := A] \subset A_k[X := A]}{\vdash C_k\, v \in [C \text{ of } A_k] \subset [(C_i : A_i[X := A])_{i\in I}]}\,+}{\vdash C_k\, v \in [C \text{ of } A_k] \subset \mu X.[(C_i : A_i)_{i\in I}]}\,\mu_r^\infty \qquad \vdash v : A_k[X := A]}{\vdash C_k\, v : \mu X.[(C_i : A_i)_{i\in I}]}\,+_i$$

Now, if $A' = \{(l_i : A_i)_{i\in I}\}$ is a strict product type then the proof is similar. However, we first need to remark that $v = \{(l_i = v_i)_{i\in I}\}$ with $v_i \in [\![A_i]\!]$ for all $i \in I$. Note that all the other possible forms of normal forms can be ruled our using similar techniques as in the proof of Lemma 7.22. By induction hypothesis, we can obtain a proof of $\vdash v_i : A_i$ for all $i \in I$ and the reconstruct proofs as in the case of the sum types. □

To conclude this section, we will discuss the closure by head reduction imposed in our definition of saturation. This condition is not usually required, but it is needed here for a subtle reason. Although it is used in the proof of Theorem 7.27, the main aim of this condition is to allow for the correctness of the subtyping rule for sums recalled bellow.

$$\dfrac{I_1 \subseteq I_2 \qquad (\gamma \vdash t.C_i \in A_i \subset B_i)_{i\in I_1}}{\gamma \vdash t \in [(C_i : A_i)_{i\in I_1}] \subset [(C_i : B_i)_{i\in I_2}]}\,+$$

Indeed, closure under head reduction is necessary to accommodate the simple witnesses of the form $t.C_i$. It would be possible to use more complex witnesses similar to those introduced by the following encoding of sums as products.

$$[(C_i \text{ of } A_i)_{i\in I}] = \forall X\{(C_i : A_i \Rightarrow X)_{i\in I}\} \Rightarrow X$$

However, there is a fundamental problem with this encoding as the witnesses would mention all the types A_i and B_i due to subtyping on the arrow types. As a consequence, such witnesses would prevents the derivation of subtyping relations like $\forall X[C \text{ of } A] \subset [C \text{ of } \forall XA]$ or $[C \text{ of } \exists XA] \subset \exists X[C \text{ of } A]$. The simple witnesses mention none of these types, and thus give a workaround to this problem.

8. Fixpoint and termination

We will now extend the system with general recursion using a fixpoint combinator $Yx.t$, while preserving a termination property on programs. Obviously, strong normalisation is compromised by the reduction rule $Yx.t \succ t[x := Yx.t]$ of the fixpoint. Nonetheless, we will prove normalisation for all the weak reduction strategies, (i.e., those that do not reduce under λ-abstractions, and hence under the right members of case analyses).

Moreover, to prove the termination of certain programs, we will need to express the fact that some functions are size-preserving. For example, proving the termination of quicksort will require the partition function to return two lists that are no bigger than the input list. To this aim, we provide quantification over ordinals in types. We will thus be able 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 on lists, where $L_\alpha(A) = \mu_\alpha X.[\text{Nil of } \{\}, \text{Cons of } \{car : A, cdr : X\}]$. It is important to note that this is a subtype of $\forall A.\forall B.(A \Rightarrow B) \Rightarrow L_\infty(A) \Rightarrow L_\infty(B)$.

Finally, proving the termination of recursive programs will generally require us to extend our typing judgments with ordinal contexts. We will then be able to assume that certain ordinals are positive while building typing proofs. For example, if we know that $l : L_\alpha(A)$ and we want to type the case analysis $[l \mid \text{Nil} \rightarrow u \mid \text{Cons} \rightarrow v]$, then we can assume that $\alpha > 0$ when typing u and v. Indeed, if $\alpha = 0$ then we know that l is a neutral term and the typing proof is trivial. Without this technique, we would for example not be able to use the previously size-preserving type for the map function on lists. To transfer positivity hypotheses from subtyping judgments to typing judgements, we will rely on new connectives $A \wedge \alpha$ and $A \vee \alpha$. The former will be interpreted as A if $\alpha \neq 0$ and as $\forall X.X$ otherwise, and the latter will be interpreted as A id $\alpha \neq 0$ and as $\exists X.X$ otherwise. They will appear in the premises of our typing rules, and they will be handled using new subtyping rules.

Definition 8.1. We extend the syntax of terms and types given in Definition 5.1 with a fixpoint combinator and new connectives as follows.

$$t, u ::= \cdots \mid Yx.t$$

$$A, B ::= \cdots \mid \forall \alpha.A \mid \exists \alpha.A \mid A \wedge \alpha \mid A \vee \alpha$$

Note that this new definition also impacts abstract judgments and syntactic ordinals. However, we will still work with abstract judgments of the form $t : A$ and $t \in A \subseteq B$. As for λ-abstractions, terms of the form $Yx.t$ are not allowed to bind variables through choice operators of the form $\varepsilon_{x \in A}(t \notin B)$.

Convention 8.2. We will use the abbreviations $A \wedge \gamma$ and $A \vee \gamma$, where $\gamma = \kappa_1, \ldots, \kappa_n$ is an ordinal context, to denote $A \wedge \kappa_1 \ldots \wedge \kappa_n$ and $A \vee \kappa_1 \ldots \vee \kappa_n$ respectively. In particular, if $\gamma = \emptyset$ then we have $A \wedge \gamma = A \vee \gamma = A$. We will also use the notation γ_1, γ_2 for the union of the ordinals contexts γ_1 and γ_2.

Before going into the typing and subtyping rules of the extended system, we first need to consider a syntactic condition on terms. It will be used to strengthen several typing rules by allowing us to assume the positivity of syntactic ordinals in some cases.

Definition 8.3. We say that a term $t \in \Lambda$ is *weakly normal* and we write $t\downarrow$ if either $t = \varepsilon_{x \in A} u \notin B$, $t = \lambda x.u$, $t = Cu$ and $u\downarrow$, or $t = \{(l_i = u_i)_{i \in I}\}$ and $u_i\downarrow$ for all $i \in I$.

Definition 8.4. Our typing judgments now have the form $\gamma \vdash t : A$, where γ is an ordinal context. The typing rules of the extended system are given in Figure 12. Its subtyping

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

$$\frac{\gamma \vdash t : (A \to B) \wedge \gamma_0 \qquad \gamma, \gamma_0 \vdash u : A}{\gamma \vdash t\, u : B} \to_e \qquad \frac{\gamma \vdash \varepsilon_{x \in A}(t \notin B) \in A \subset C}{\gamma \vdash \varepsilon_{x \in A}(t \notin B) : C} \varepsilon$$

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

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

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

$$\frac{\gamma \vdash t : \{l_k : A; \dots\}}{\gamma \vdash t.l_k : A} \times_e \qquad \frac{\gamma \vdash t[x := Yx.t] : A}{\gamma \vdash Yx.t : A} Y$$

Figure 12: Typing of the system extended with general recursion.

$$\frac{\gamma \vdash t \in A[\alpha := \kappa] \subset B}{\gamma \vdash t \in \forall \alpha.A \subset B} \forall_l^o \qquad \frac{\gamma \vdash t \in A \subset B[\alpha := \varepsilon_{\alpha < \mathcal{O}}(t \notin B)]}{\gamma \vdash t \in A \subset \forall \alpha.B} \forall_r^o$$

$$\frac{\gamma \vdash t \in B \subset A[\alpha := \kappa]}{\gamma \vdash t \in B \subset \exists \alpha.A} \exists_r^o \qquad \frac{\gamma \vdash t \in B[X := \varepsilon_{\alpha < \mathcal{O}}(t \in B)] \subset A}{\gamma \vdash t \in \exists \alpha.B \subset A} \exists_l^o$$

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

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

Figure 13: Extra subtyping rules for the extended system.

rules still include those of Figure 8, but the rules of Figure 13 are added to handle the new connectives. Note that we allow circular subtyping proofs using the (G^+) and (I_k^+) rules as in Section 5, and circular typing rules using the (G) and (I_k) rules.

The typing rules of the system need to be changed completely to account for the ordinal contexts. Note that they are strongly linked to the new connectives $A \wedge \alpha$ and $A \vee \alpha$ in types. Moreover, the ($\times_i$) and ($+_i$) rules require some terms to be weakly normal to learn the positivity of certain syntactic ordinals. Furthermore, the system now includes circular typing proofs to handle general recursion. Note that the typing rule of the fixpoint is very simple as it only performs an unfolding. In practice, we will only need to allow circularity on typing judgments of the form $\gamma \vdash Yx.t : A$.. In this context, the (G) and (I_k) rules can be written as in Figure 14, where we write $\varepsilon_{\overline{\alpha} < C(\overline{\alpha})}(Yx.t \notin A)_i$ for the ordinal $\varepsilon_{\overline{\alpha} < C(\overline{\alpha})} \neg(Yx.t : A)_i$ (see Section 4).

$$
\dfrac{\forall\overline{\alpha}(\gamma \vdash C(\overline{\alpha}) \Rightarrow Yx.t : A) \qquad (\gamma[\overline{\alpha} := \overline{\kappa}], \delta \vdash \kappa_i < C(\overline{\kappa})_i)_{1 \leq i \leq |\overline{\alpha}|}}{\gamma[\overline{\alpha} := \overline{\kappa}], \delta \vdash Yx.t : A[\overline{\alpha} := \overline{\kappa}]}\ \mathrm{G}
$$

$$
\begin{array}{l}
[\forall\overline{\alpha}(\gamma \vdash C(\overline{\alpha}) \Rightarrow Yx.t : A)]_k \\
\qquad\quad\vdots
\end{array}
$$

$$
\dfrac{\gamma[\overline{\alpha} := \overline{\kappa}] \vdash Yx.t : A[\overline{\alpha} := \overline{\kappa}] \qquad \text{where } \overline{\kappa} = \overline{\varepsilon}_{\overline{\alpha} < C(\overline{\alpha})}(Yx.t \notin A)}{\forall\overline{\alpha}(\gamma \vdash C(\overline{\alpha}) \Rightarrow Yx.t : A)}\ \mathrm{I}_k
$$

Figure 14: Specialised circular typing rules for general recursion.

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

$$
\begin{aligned}
\mathrm{F}(X) &= [\mathrm{Z} \mid \mathrm{S} \text{ of } X] \\
\mathbb{N}_\alpha &= \mu_\alpha X.\mathrm{F}(X) \\
\mathbb{N} &= \mathbb{N}_\infty \\
\mathrm{z} : \mathbb{N} &= Z \\
\mathrm{s} : \mathbb{N} \to \mathbb{N} &= \lambda n.Sn
\end{aligned}
\qquad\qquad
\begin{aligned}
\mathrm{id} &= Y id.\lambda n.[n \mid Z \to \mathrm{z} \mid Sp \to \mathrm{s}\ (id\ p)] \\
\kappa_0 &= \varepsilon_{\alpha_1 < \mathcal{O}}(\mathrm{id} \notin \mathbb{N}_{\alpha_1} \to \mathbb{N}) \\
\kappa_1 &= \varepsilon_{\alpha < \kappa_0}(n_0 \in \mathrm{F}(\mathbb{N}_\alpha)) \\
n_0 &= \varepsilon_{n \in \mathbb{N}_{\kappa_0}}([n \mid Z \to \mathrm{z} \mid Sp \to \mathrm{s}\ (\mathrm{id}\ p)] \notin \mathbb{N}) \\
p_0 &= \varepsilon_{p \in \mathbb{N}_{\kappa_1}}(\mathrm{s}\ (\mathrm{id}\ p) \notin \mathbb{N})
\end{aligned}
$$

Figure 15: Typing proof of the recursive identity function on $\mathbb{N}$.

Example 8.5. We consider the identity function on unary natural numbers. It can be typed using the derivation given in Figure 15, which is the simplest possible example of a circular typing proof. Follownig the terminology of Section 4), the proof is formed using two blocks. The former, that we will call B_0, starts at the root of the proof and only contains one typing rule. The latter, that we will call B_1, contains all the rest of the proof. The call graph corresponding to the proof contains one edge from B_0 to B_1, labelled with the empty matrix, and one edge from B_1 to itself, labelled with the 1×1 matrix (-1) since we can prove $\kappa_0 \vdash \kappa_1 < \kappa_0$.

It is important to note that the positivity of κ_0 must be known to obtain $\kappa_1 < \kappa_0$. It is thus essential to use the type $F(\mathbb{N}_{\kappa_1}) \wedge \kappa_0$ (and not $F(\mathbb{N}_{\kappa_1})$) for the first premisse of the $(+_e)$ rule. This allows us to assume that κ_0 is positive when typing its other premisses. There would be no way of building a typing proof withoug doing so.

We will now modify the semantics that was given in Section 7 to account for the fixpoint combinator and the new connectives. Here, we will not be able to interpret types as subsets of $\mathcal{N}$, since the reduction rule of the fixpoint will break strong normalisation. We will however be able to preserve normalisation for all the weak reduction strategies (i.e., those that do not reduce under λ-abstractions, and thus case analyses as well).

Definition 8.6. We denote $(\succ_w) \subseteq \Lambda \times \Lambda$ the one step weak reduction relation. It is defined as the least relation containing the rules of Figure 6 and $Yx.t \succ_w t[x := Yx.t]$, and that is contextually closed for weak contexts (i.e., contexts formed without a λ-abstraction constructor). Its reflexive, transitive closure is denoted $(\succ_w^*)$.

Definition 8.7. We denote $\mathcal{W} \subseteq [\![\Lambda]\!]$ the set of all the pure terms that are strongly normalising for the $(\succ_w)$ reduction relation. In other words, we have $t \in \mathcal{W}$ if and only if there is no infinite sequence of reduction of t using $(\succ_w)$.

Using the set $\mathcal{W}$ we can define a notion of saturated set, as well as a set $\mathcal{W}_0$ like in Section 7. We are then able to prove corresponding lemmas using the same techniques.

Definition 8.8. A set of pure terms $\Phi \subseteq [\![\Lambda]\!]$ is said to be *weakly saturated* if it satisfies the conditions of Definition 7.3, where every occurrence of $\mathcal{N}$ is replaced by $\mathcal{W}$, plus the following condition.

(5) If $H[t[x := Yx.t]] \in \Phi$, then $H[Yx.t] \in \Phi$.

Definition 8.9. The set $\mathcal{W}_0$ is defined as $\mathcal{N}_0$ (see Definition 7.6), but using $\mathcal{W}$ instead of $\mathcal{N}$. We denote by $\overline{\mathcal{W}_0}$ the least weakly saturated set containing $\mathcal{W}_0$.

Example 8.10. The term $y\ (Yr.\lambda x.r)$ is in $\mathcal{W}_0$, but not in $\mathcal{N}_0$.

Using the above definitions, we can obtain similar properties as in Section 7. This is mainly due to the fact that the proof of theses lemmas do not considers reductions which are allowed for $(\succ)$ but forbidden for $(\succ_w)$. We will first show that $\mathcal{W}$ is weakly saturated, but this requires a small lemma that was immediate in Section 7.

Lemma 8.11. *For any terms $t \in \Lambda$ and $u \in \mathcal{W}$ such that $u \succ_w^* u'$, if $t[x := u'] \in \mathcal{W}$ has an infinite weak reduction then $t[x := u]$ also has one.*

Proof. We reason coinductively. We first distinguish the occurrences of x in t that appear under an abstraction by denoting them x_0, while denoting the others x_1. We hence obtain $t[x := u] = (t[x_1 := u])[x_0 := u]$ and $t[x := u'] = (t[x_1 := u'])[x_0 := u']$ up to a renaming of x. Let us now consider the first step of an infinite reduction of $t[x := u'] \succ_w t'[x_0 := u']$, with $t[x_1 := u'] \succ_w t'$ (there cannot be any weak reduction for the occurrences of u' replacing x_0). We thus have $t[x := u] = (t[x_1 := u])[x_0 := u] \succ_w^* (t[x_1 := u'])[x_0 := u] \succ_w t'[x_0 := u]$. This step being productive, we can apply the coinduction hypothesis with t' to get an infinite weak reduction of $t'[x_0 := u]$ from the infinite weak reduction of $t'[x_0 := u']$. □

Lemma 8.12. *The set $\mathcal{W}$ is weakly saturated.*

Proof. The proof is exactly the same as that of Lemma 7.5, except for condition (1). In this case, we need to prove that if $H[t[x := u]] \in \mathcal{W}$ and $u \in \mathcal{W}$, then $H[(\lambda x.t)\ u] \in \mathcal{W}$. We thus suppose, by contradiction, that $H[(\lambda x.t)\ u]$ has an infinite weak reduction. Such a reduction must start with $H[(\lambda x.t)\ u] \succ_w^* H'[(\lambda x.t)\ u'] \succ_w^* H'[t[x := u']]$, where $H \succ_w^* H'$ and $u \succ_w^* u'$. It can hence be transformed into $H[t[x := u]] \succ_w^* H'[t[x := u]]$ and we can use Lemma 8.11 to obtain an infinite reduction of $H'[t[x := u]]$ from the infinite reduction of $H'[t[x := u']]$. This gives a contradiction with $H[t[x := u]] \in \mathcal{W}$. □

We will now consider the interpretation of terms, types and syntactic ordinals to handle the fixpoint and the new connectives. However, let us first give the new domain of interpretation for our types.

Definition 8.13. The set of every type interpretaton $[\![\mathcal{F}]\!]$ is now defined as follows.

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

Definition 8.14. We modify the definition of the interpretation of terms and formulas given in Figure 11 by replacing every occurence of $\mathcal{N}$ and $\overline{\mathcal{N}_0}$ with $\mathcal{W}$ and $\overline{\mathcal{W}_0}$ respectively. The new syntactic elements are interpreted as follows.

$$[\![Yx.t]\!] = Yx.[\![t]\!] \qquad [\![A \wedge \kappa]\!] = \begin{cases} [\![A]\!] \text{ if } [\![\kappa]\!] \neq 0 \\ \overline{\mathcal{W}_0} \text{ otherwise} \end{cases}$$

$$[\![\forall \alpha.A]\!] = \cap_{o \in [\![\mathcal{O}]\!]} [\![A[\alpha := o]]\!]$$

$$[\![\exists \alpha.A]\!] = \cup_{o \in [\![\mathcal{O}]\!]} [\![A[\alpha := o]]\!] \qquad [\![A \vee \kappa]\!] = \begin{cases} [\![A]\!] \text{ if } [\![\kappa]\!] \neq 0 \\ \mathcal{W} \text{ otherwise} \end{cases}$$

Theorem 8.15. *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. The proof is similar as for Theorem 7.20. The cases for the four new type constructors are immediate by induction hypothesis. □

We will now give the adequacy lemma for the new system, which will be similar to that of Section 7. For this reason, we will not state all require lemmas (e.g., the equivalent of Theorem 7.15), as their proof does not change much. We however need a small lemma for handling the strong normality condition in some of our new typing rules.

Lemma 8.16. *If $t \in \Lambda$ be a term such that $t \downarrow$ (i.e., t is weakly normal), then $[\![t]\!] \in \mathcal{W}$.*

Proof. Immediate by induction, using Theorem 8.15 when $t = \varepsilon_{x \in A}(t \notin B)$. □

Theorem 8.17. *Let γ be an ordinal context such that $[\![\tau]\!] > 0$ for all $\tau \in \gamma$.*

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

Proof. The proof is similar to that of Theorem 7.23, using Theorem 4.14. For the local subtyping rules of Figure 8, the proof remains essentially the same. Occurrences of $\mathcal{N}$ and $\overline{\mathcal{N}_0}$ need to be replaced by $\mathcal{W}$ and $\overline{\mathcal{W}_0}$, and lemmas need to be modified according to the new definitions (their proofs are mostly unchanged). Similarly, the cases of the (ε) and $(\times_e)$ typing rules are unchanged (up to the transmission of the context in the induction hypothesis). Hence, we only consider the cases of the remaining typing rules of Figure 12, and the local subtyping rules of Figure 13.

- $(\to_i)$ We need to show $[\![\lambda x.t]\!] \in [\![C]\!]$. According to the first induction hypothesis, it is enough to show $[\![\lambda x.t]\!] \in [\![(A \to B) \vee \gamma_0]\!]$. If there is $\kappa \in \gamma_0$ such that $[\![\kappa]\!] = 0$ then we have $[\![(A \to B) \vee \gamma_0]\!] = \mathcal{W}$ and we can conclude immediately by Lemma 8.16. We can thus assume that $[\![\kappa]\!] \neq 0$ for all $\kappa \in \gamma_0$ and that the positivity context of the second induction hypothesis is valid to obtain $[\![t[x := \varepsilon_{x \in A}(t \notin B)]]\!] \in [\![B]\!]$. By definition of the choice operator, this means that $[\![t[x := u]]\!] \in [\![B]\!]$ for all $u \in [\![A]\!]$. Hence we can conclude $[\![\lambda x.t]\!] \in [\![A \to B]\!] = [\![(A \to B) \vee \gamma_0]\!]$ since we know that $[\![A \to B]\!]$ is weakly saturated.
- $(\to_e)$ We need to show $[\![t\ u]\!] \in [\![B]\!]$. By the first induction hypothesis $[\![t]\!] \in [\![(A \to B) \wedge \gamma_0]\!]$. If $[\![\kappa]\!] = 0$ for some $\kappa \in \gamma_0$, then $[\![(A \to B) \wedge \gamma_0]\!] = \overline{\mathcal{W}_0}$ and thus we have $[\![t]\!] \in \overline{\mathcal{W}_0}$, which implies $[\![t\ u]\!] \in \overline{\mathcal{W}_0} \subseteq [\![B]\!]$. Otherwise, we have $[\![\kappa]\!] \neq 0$ for all $\kappa \in \gamma_0$, and thus $[\![t]\!] \in [\![A \to B]\!]$. We can hence use the second induction hypothesis to get $[\![u]\!] \in [\![A]\!]$ and conclude by definition of $[\![A \to B]\!]$.

($\times_i$) We only need to prove $[\![\{(l_i : A_i)_{i\in I}\}]\!] \in [\![\{(l_i : A_i)_{i\in I}\} \vee \gamma_0]\!]$ according to the first induction hypothesis. If $[\![\kappa]\!] = 0$ for some $\kappa \in \gamma_0$ and if $t_i \downarrow$ for all $i \in I$, then we can conclude immediately using Lemma 8.16 as $[\![\{(l_i : A_i)_{i\in I}\} \vee \gamma_0]\!] = \mathcal{W}$. Otherwise, we can use the remaining induction hypotheses to get $[\![t_i]\!] \in [\![A_i]\!]$ for all $i \in I$. From this we obtain $[\![\{(l_i = t_i)_{i\in I}\}]\!] \in [\![\{(l_i : A_i)_{i\in I}\}]\!]$ using weak saturation. We can then conclude since $[\![\{(l_i : A_i)_{i\in I}\} \vee \gamma_0]\!] = [\![\{(l_i : A_i)_{i\in I}\}]\!]$ by definition.

($+_i$) We only need to prove $[\![C\,t]\!] \in [\![[C \text{ of } A] \vee \gamma_0]\!]$ according to the first induction hypothesis. It $[\![\kappa]\!] = 0$ for some $\kappa \in \gamma_0$ and if t is weakly normal, then we can conclude immediately using Lemma 8.16. Otherwise, we must have $[\![\kappa]\!] \neq 0$ for all $\kappa \in \gamma_0$. Therefore, we can use the second induction hypothesis to get $[\![t]\!] \in [\![A]\!]$. From this we obtain $[\![C\,t]\!] \in [\![[C \text{ of } A]]\!]$ by saturation.

($+_e$) We need to show $[\![[t \mid (C_i \to t_i)_{i\in I}]]\!] \in [\![B]\!]$. By the first induction hypothesis, we have $[\![t]\!] \in [\![[(C_i : A_i)_{i\in I}] \wedge \gamma_0]\!]$. If $[\![\kappa]\!] = 0$ for some $\kappa \in \gamma_0$ then we obtain $[\![t]\!] \in \overline{\mathcal{W}_0}$, and thus $[\![[t \mid (C_i \to t_i)_{i\in I}]]\!] \in \overline{\mathcal{W}_0} \subset [\![B]\!]$. Otherwise, the result follows from the right induction hypotheses and the definition of $[\![[(C_i : A_i)_{i\in I}]]\!]$.

(Y) By definition, we have $(Yx.t) \succ_w t[x := Yx.t]$. As a consequence, the validity of the rule follows from the weak saturation condition (5) on $[\![A]\!]$.

($\forall_l^o$) If $[\![t]\!] \in [\![\forall\alpha.A]\!]$ then $[\![t]\!] \in [\![A[\alpha := [\![\kappa]\!]]]\!] = [\![A[\alpha := \kappa]]\!]$ by the substitution lemma. Hence, the induction hypothesis gives $[\![t]\!] \in [\![B]\!]$.

($\forall_r^o$) Let us suppose that $[\![t]\!] \in [\![A]\!]$ and assume $[\![t]\!] \notin [\![\forall\alpha.B]\!]$ by contradiction. There must be $o \in [\![\mathcal{O}]\!]$ such that $[\![t]\!] \notin [\![B[x := o]]\!]$. By definition of the choice operator, this means means that $[\![t]\!] \notin [\![B[x := \varepsilon_{\alpha<\mathcal{O}}(t \notin B)]]\!]$. We hence obtain a contradiction with $[\![t]\!] \in [\![A]\!]$ using the induction hypothesis.

($\exists_r^o$) Similar to the ($\forall_l^o$) case.

($\exists_l^o$) Similar to the ($\forall_r^o$) case.

($\wedge_l$) We assume that $[\![t]\!] \in [\![A \wedge \kappa]\!]$. If $[\![\kappa]\!] = 0$ then $[\![A \wedge \kappa]\!] = \overline{\mathcal{W}_0}$ and hence $[\![t]\!] \in [\![B]\!]$. If $[\![\kappa]\!] \neq 0$ then $[\![A \wedge \kappa]\!] = [\![A]\!]$ and we can thus conclude by induction hypothesis.

($\wedge_r$) Since $\kappa \in \gamma$ we know that $[\![\kappa]\!] \neq 0$ and thus we have $[\![A \wedge \kappa]\!] = [\![A]\!]$. We can thus conclude by induction hypothesis.

($\vee_r$) Similar to the ($\wedge_l$) case.

($\vee_l$) Similar to the ($\wedge_r$) case. □

Theorem 8.18. *As for the initial system, we get termination (typed terms are normalising for every weak reduction strategy), type safety for simple data types and consistency.*

Proof. The proofs are similar to those of Theorems 7.25, 7.27 and 7.24 respectively (using the results of the current section). □

9. Terminating examples

We will now consider several examples of functions that are typable in our system, and accepted by our implementation. We will start with examples on lists, as the usual functions on unary natural numbers are not more difficult to handle than the recursive identity function of Figure 15.

The type of lists of *size* α given at the top of Figure 16 is straight forward. It allows us to define the traditional map function, which is decorated with the information that it preserves size. Note that its type does not guarantee that the input and output lists have

$$\begin{aligned}
\mathrm{F}(A, X) &= [\mathrm{Nil} \mid \mathrm{Cons\ of}\ \{\mathrm{hd} : A; \mathrm{tl} : X\}] \\
\mathbb{L}_\alpha(A) &= \mu_\alpha X.\mathrm{F}(A, X) \\
\mathbb{L}(A) &= \mathbb{L}_\infty(A) \\
\mathrm{map}\ &: \forall A.\forall B.\forall \alpha.(A \to B) \to \mathbb{L}_\alpha(A) \to \mathbb{L}_\alpha(B) \\
&= Y map.\lambda f\, l. \left[l \left| \begin{array}{l} [] \to [] \\ x :: l \to f\ x{::}map\ f\ l \end{array} \right. \right] \\
\mathrm{map}_2\ &: \forall A.\forall B.\forall C.\forall \alpha.(A \to B \to C) \to \mathbb{L}_\alpha(A) \to \mathbb{L}_\alpha(B) \to \mathbb{L}_\alpha(C) \\
&= Y map_2.\lambda f\, l_1\, l_2. \left[l_1 \left| \begin{array}{l} [] \to [] \\ x :: l_1 \to \left[l_2 \left| \begin{array}{l} [] \to [] \\ y :: l_2 \to f\ x\ y{::}map_2\ f\ l_1\ l_2 \end{array} \right. \right] \end{array} \right. \right] \\
\mathrm{flatten}\ &: \forall A.\mathbb{L}(\mathbb{L}(A)) \to \mathbb{L}(A) \\
&= Y flatten.\lambda l_s. \left[l_s \left| \begin{array}{l} [] \to [] \\ l :: l_s \to \left[l \left| \begin{array}{l} [] \to flatten\ l_s \\ x :: l \to x{::}flatten\ (l{::}l_s) \end{array} \right. \right] \end{array} \right. \right] \\
\mathrm{insert}\ &: \forall \alpha.\forall A.(A \to A \to \mathbb{B}) \to A \to \mathbb{L}_\alpha(A) \to \mathbb{L}_{\alpha+1}(A) \\
&= Y insert.\lambda f\, a\, l. \left[l \left| \begin{array}{l} [] \to a{::}[] \\ x :: l \to [f\ a\ x \mid T \to a{::}l \mid F \to x{::}insert\ f\ a\ l] \end{array} \right. \right] \\
\mathrm{sort}\ &: \forall \alpha.\forall A.(A \to A \to \mathbb{B}) \to \mathbb{L}_\alpha(A) \to \mathbb{L}_\alpha(A) \\
&= Y sort.\lambda f\, l. \left[l \left| \begin{array}{l} [] \to [] \\ x :: l \to \mathrm{insert}\ f\ x\ (sort\ f\ l) \end{array} \right. \right]
\end{aligned}$$

Figure 16: Examples of functions on lists (map, flatten and insertion sort).

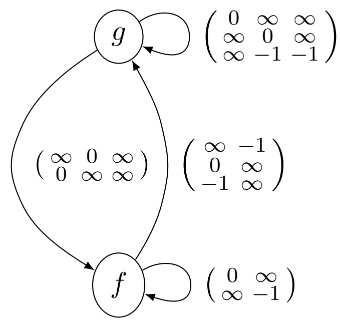
the same size, but rather that the output list is at most as long as the input list. More surprisingly, the map_2 function can also be typed with some size information. However, the type it is given here is not enough as it forbids using map_2 on input lists of unrelated sizes, while still preserving size information about the result. A more precise and useful type for map_2 would require extending our syntactic ordinals with a min symbol. Indeed, we could then us the type $\forall A.\forall B.\forall C.\forall \alpha.\forall \beta, (A \to B \to C) \to \mathbb{L}_\alpha(A) \to \mathbb{L}_\beta(B) \to \mathbb{L}_{\min(\alpha,\beta)}(C)$. Nonetheless, it is important to note that the types of map and map_2 are subtypes of their usual type (with no size information). For example, we can derive

$$\forall A.\forall B.\forall \alpha.(A \to B) \to \mathbb{L}_\alpha(A) \to \mathbb{L}_\alpha(B) \subset \forall A.\forall B.(A \to B) \to \mathbb{L}(A) \to \mathbb{L}(B)$$

in our system. As a consequence, the map and map_2 functions of Figure 16 are suitable for all applications. In particular, we do not need to provide two different versions (one with size information, and one without).

We will now consider the flatten function, which is also given in Figure 16. On this particular example, proving termination requires unrolling the fixpoint twice. Indeed, if we only unroll it once then our algorithm infers the general abstract sequent $\forall \alpha_0, \alpha_1 (\vdash \mathrm{f} : \forall A.\mathbb{L}_{\alpha_1}(\mathbb{L}_{\alpha_0}(A)) \to \mathbb{L}(A))$, which is not sufficient for proving termination. However, if we unroll the second recursive call twice we obtain two different induction hypotheses, and the algorithm succeeds in proving termination. This amounts to typing the program given at the top of Figure 17 using the abstract sequents given at its bottom. We will now give

$$Y f.\lambda l_s.\left[l_s \left| \begin{array}{l} [] \to [] \\ l :: l_s \to \left[l \left| \begin{array}{l} [] \to f\ l_s \\ x :: l \to x{::}Y g.\lambda l_s.\left[l_s \left| \begin{array}{l} [] \to [] \\ l :: l_s \to \left[l \left| \begin{array}{l} [] \to f\ l_s \\ x :: l \to x{::}g\ (l{::}l_s)\end{array}\right.\right]\end{array}\right.\right]\ (l{::}l_s)\end{array}\right.\right]\end{array}\right.\right]$$



$$\forall \alpha_0, \alpha_1(\vdash \mathrm{f} : \forall A.\mathbb{L}_{\alpha_1}(\mathbb{L}_{\alpha_0}(A)) \to \mathbb{L}(A))$$
$$\forall \beta_0, \beta_1, \beta_2(\beta_1 \vdash \beta_2 < \beta_1 \Rightarrow \mathrm{g} : F(\mathbb{L}_{\beta_2}(A_0), \mathbb{L}_{\beta_0}(\mathbb{L}_{\beta_1}(A_0))) \to \mathbb{L}(A_0))$$

Figure 17: Unrolling of flatten and the corresponding call graph.

some explanations about the call graph of the function, and in particular the size change matrices labeling its edges.

($f \to f$) The loop on f corresponds to the first recursive call. The 2×2 matrix is justified because in this call the size of the inner list α_0 is constant, while α_1 decreases.

($f \to g$) The edge from f to g represents the definition of g inside f, which must be seen as f calling g. In this call, the first line of the matrix is justified by $\beta_0 < \alpha_1$ because β_0 is the size of the tail of the outer list. The second line is justified because β_1, the size of the inner list, is equal to α_0. The last line is justified because β_2, the size of the first element of the outer list decreases (it is smaller than α_0).

($g \to g$) The loop on g corresponds to the last recursive call, where β_0 and β_1 are constant (which justifies the first two lines). The first element of the list is decreasing, so β_2 decreases. Moreover, as we keep in the general abstract sequent the information that $\beta_2 < \beta_1$, we also have a -1 in the middle of the last line.

($g \to f$) Finally, the edge from g to f corresponds to the third recursive call where we have $\alpha_0 = \beta_1$, $\alpha_1 = \beta_0$ and β_2 become useless (hence the two ∞ on the last column).

The size change principle yields a positive answer on this call graph. This means that the typing derivation is well-founded, and thus correct.

The last example given in Figure 16 is insertion sort, for which our implementation is able to derive both termination and size preservation. The system is also able to derive the termination of quicksort and merge sort, but in both cases we are unable to obtain size preservation. However, it might be possible to obtain size preservation on such a program by first enriching our language of syntactic ordinals with an addition symbol for example. For instance, this would allow us to give a precise type to the partition function required for quicksort.

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

Figure 18: Append lists as a supertype of lists.

To illustrate the use of subtyping, a simple example implementing *append lists* is provided in Figure 18. Roughly, an append list is formed like a list, but an additional constructor is provided for concatenation (we thus obtain constant time concatenation). Thanks to subtyping, a list is an append list, and thus the conversion function fromList is just the identity. A recursive function toList is however required in the other direction to effectively concatenate the lists contained in App nodes.

To conclude this section, we will now give an example mixing inductive and coinductive types. We consider the type of streams $\mathbb{S}(A)$ and the type of filter on streams $\mathbb{F}$ defined at the top of Figure 20. In the type of filters, the variant R indicates that one element of the stream should be *removed*, while the variant K indicates that one element should be kept. Note that in the type $\mathbb{F}$, the inner type $\mu Y.(\{\} \to [\mathrm{R\ of}\ Y \mid \mathrm{K\ of\ X}])$ imposes that we can only have finitely many R constructors between K constructors. As a filter must contain infinitely many K constructors, this ensures the productivity of the filter function, applying a filter to a stream, and the cmp function composing two filters.

As in the example of the flatten function on lists, both filter and cmp require some unrolling. To avoid this, we may replace the type $\mathbb{F}$ with $\mathbb{F}' = \mu Y.(\{\} \to [\mathrm{R\ of}\ Y \mid \mathrm{K\ of}\ \mathbb{F}])$. Indeed, although $\mathbb{F} \subset \mathbb{F}'$ and $\mathbb{F}' \subset \mathbb{F}$ are both derivable, $\mathbb{F}'$ carries an ordinal representing the initial number of R constructors in the type. The call-graph for cmp is given in Figure 19 and gives an example of a non trivial instance of the size change principle.

Note also that $\mathbb{F}$ is isomorphic to the type of streams over natural numbers, and that we can prove the termination of this isomorphism while keeping size information about the streams. The isomorphism is given by the s2f and f2s functions.

More examples are provided with the implementation of our prototype [24]. They contain, for example, the GCD function for binary natural numbers, and the basic operations for exact real arithmetic (using the signed digits representation). In particular, all of these examples are proved terminating by our implementation.

10. Type-checking Algorithm

Our system can be implemented by transforming the deduction rule systems given in this paper into recursive functions. This can be done relatively easily because the system is mostly syntax-directed. For instance, only one typing rule applies for each term constructor, and at most two subtyping rules apply for each pair of type constructors. It is easy to see that when two subtyping rules may apply (one left rule and one right rule), then they commute (e.g., quantifier rules). This is due to the fact that they do not modify the term

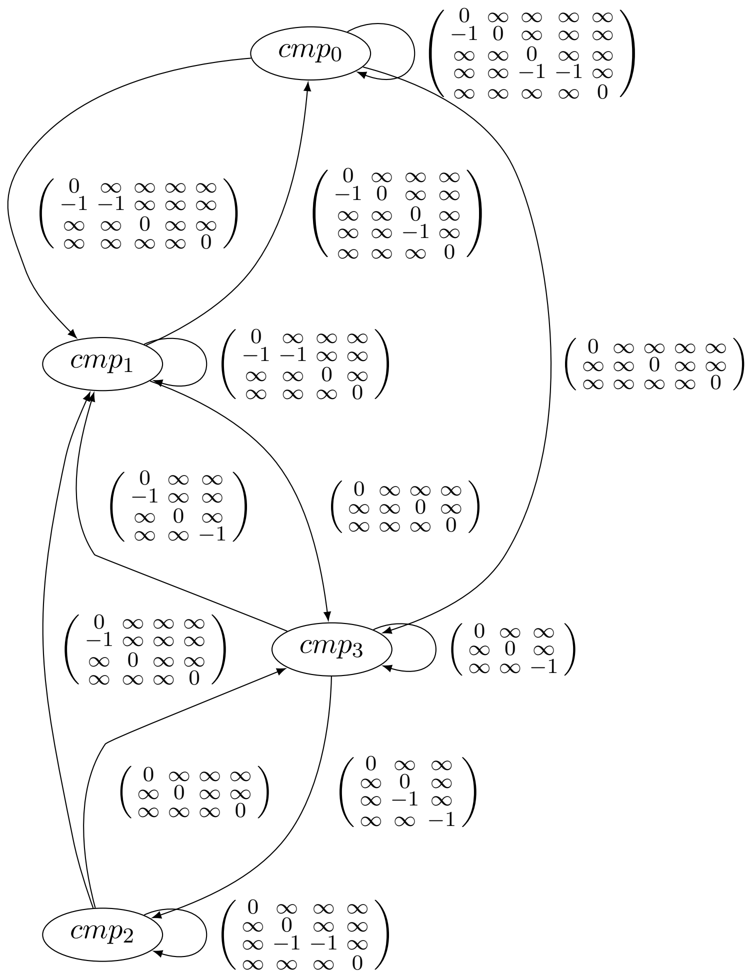


Figure 19: Call-graph of the cmp function.

carried by the judgment, and that choice operators are constructed using only the term and the type on the side where it is applied. This means that the order in which such rules are applied does not matter. Moreover, if the rule for implication, product or sum can be applied, then it is easy to see that no other rule can be applied (except generalisation).

Another important remark about the system is that if we limit the unrolling depth for fixpoints in typing rules, then the only possible place where an implementation may loop is in the subtyping function. Indeed, every typing rule (except fixpoint unrolling) decreases the size of the term, if we consider choice operators to have size zero (we will come back to this point when we discuss type errors).

Nonetheless, several subtle details need further discussion. We will here give some guidelines explaining parts of our implementation. We encourage the reader to look at the code of our prototype [24], which should be relatively accessible (at least to readers familiar with the implementation of type systems). According to the previous remarks, the only implementation freedom is in the management of the rules introducing unknown types or ordinals (namely $(\forall_l)$, $(\exists_r)$, $(\forall_l^o)$, $(\exists_r^o)$, (μ_r) and (ν_l)), in the management of the ordinal

$$
\begin{aligned}
\mathbb{S}(A) &= \nu X.(\{\} \to A \times X) \\
\mathbb{F}_\alpha &= \nu_\alpha X.\mu Y.(\{\} \to [\mathrm{R}\ \text{of}\ Y \mid \mathrm{K}\ \text{of}\ X]) \\
\mathbb{F} &= \mathbb{F}_\infty \\
\text{filter } &: \forall A.\mathbb{F} \to \mathbb{S}(A) \to \mathbb{S}(A) \\
&= Y filter.\lambda f\, s.(\lambda(h,t).\left[f\ () \middle| \begin{array}{l} Rf' \to filter\ f'\ t \\ Kf' \to \lambda u.(h, filter\ f'\ t) \end{array}\right]) \ (s\ ()) \\
\text{cmp } &: \mathbb{F} \to \mathbb{F} \to \mathbb{F} \\
&= Y cmp.\lambda f_1\, f_2\, u. \left[f_2\ () \middle| \begin{array}{l} Kf_2' \to \left[f_1\ () \middle| \begin{array}{l} Kf_1' \to K(cmp\ f_1'\ f_2') \\ Rf_1' \to R(cmp\ f_1'\ f_2') \end{array}\right] \\ Rf_2' \to R(cmp\ f_1\ f_2') \end{array}\right] \\
\text{f2s } &: \forall \alpha.\mathbb{F}_\alpha \to \mathbb{S}_\alpha(\mathbb{N}) \\
&= Y f2s.\lambda s\, u. \left[s\ () \middle| \begin{array}{l} Rs \to (\lambda(n,r).(Sn, r))\ (f2s\ s\ ()) \\ Ks \to (Z, f2s\ s) \end{array}\right] \\
s2f_{aux} &: \forall \alpha.\mathbb{F}_\alpha \to \mathbb{N} \to \mathbb{F}_{\alpha+1} \\
&= Y s2f_a ux.\lambda s\, n. \left[n \middle| \begin{array}{l} Z \to \lambda u.Ks \\ Sp \to \lambda u.R(s2f_a ux\ s\ p) \end{array}\right] \\
\text{s2f } &: \forall \alpha.\mathbb{S}_\alpha(\mathbb{N}) \to \mathbb{F}_\alpha \\
&= Y s2f.\lambda s\, u.(\lambda(n,s).s2f_{aux}\ (s2f\ s)\ n\ ())\ (s\ ())
\end{aligned}
$$

Figure 20: Examples with streams and filters on streams.

contexts with the $A \wedge \gamma$ and $A \vee \gamma$ connective, and in the construction of circular typing and local subtyping proofs.

Unification variables. For handling unknown types and ordinals in subtyping, the natural solution is to extend their syntax with a set of unification variables. In types, we will use the letters U and V to denote unification variables, which correspond to unknown types until their value is inferred. In our prototype implementation [24], unification variables are handled as follows.

- If we encounter $\gamma \vdash t \in U \subset U$ then we use reflexivity.
- If we encounter $\gamma \vdash t \in U \subset V$, then we set $U := V$.
- If we encounter $\gamma \vdash t \in U \subset A$ or $\gamma \vdash t \in A \subset U$, then we decide that U is equal to A, provided that it does not occur in A. Note that it is essential to check occurrence inside choice operators for them to be well-defined (i.e., not cyclic). Moreover, when U occurs only positively in A we may use $\mu X.A[U := X]$ as a definition for U, thus allowing the system to build new recursive types.

In fact, this approach is a bit too naive in the case where we have a projection $t.l_k$ and the type of t is a unification variable. Indeed, it is usually not sufficient to fix the type of t to be a record type with only the field l_k (the dual problem arises with variants). To solve this issue, our unification variables carry a state keeping track of projected fields (or constructed variants). The state of a unification variable is initialised or updated when we encounter $\gamma \vdash t \in U \subset \{l_1 : A_1, \dots, l_n : A_n, \dots\}$ or $\gamma \vdash t \in [C_1 \text{ of } A_1, \dots, C_n \text{ of } A_n] \subset U$. This state can be seen 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.

Unification variables are also required for syntactic ordinals to handle the (μ_r), (ν_l), $(\forall_l^o)$ and $(\exists_r^o)$ rules. In syntactic ordinals, we will use the letters O and P to denote unifications variables. As for types, an ordinal unification variable O may carry constraints like $\tau \leq O < \kappa$, to delay instantiation until we have a constraint $O \leq \kappa$. Moreover, when we need to prove $\gamma \vdash t \in A \subset \mu_O F$ or $\gamma \vdash t \in \nu_O F \subset B$ and O is a unification variable, we define O to be the first ordinal in γ satisfying the constraints on O. If there is none, then we instantiate it with the successor of a unification variable or with ∞. We do this because we must fail if there is no positive solution for O. Otherwise, the subtyping procedure would often loop by building decreasing chains of unification variables.

Circular subtyping proofs. The generalisation rule used to build circular proof is the only one that is not directed by the syntax (or handled by unification variables). As a consequence, it cannot be implemented directly and requires a special treatment. In practice, we try to apply the generalisation rule to build an induction hypothesis each time we encounter a local subtyping judgment with an inductive or coinductive types on either side. In such an eventuality, we apply the generalisation rule (G^+) by quantifying over all the ordinals appearing in the types. The produced general abstract sequent is then looked up in the list of all the encountered induction hypotheses in an attempt to end the branch of the proof by induction. If the general abstract sequent has not been encountered before, then it is registered and the proof proceeds by applying the I_k^+ rule.

Note that when there are no quantifiers, only a finite number of distinct general abstract sequents can be produced, thus implying the termination of our algorithm. Indeed, when when proving a subtyping judgement $\gamma \vdash t \in A \subset B$, the formulas that appear in the proof can be uniquely identified by a pointer to a subformula of the original types A or B, and the value of the ordinals. When building a general abstract sequent, the ordinals are quantified over, and hence the general abstract sequent only depends on two pointers (for the involved types). This means that the number of distinct general abstract sequents appearing in a proof of $\gamma \vdash t \in A \subset B$ is less than $|A| \times |B|$ (where $|C|$ denotes the size of the type C). This property is similar to the finiteness of Kozen's closure for the propositional μ-calculus [21]. When quantification over types is allowed, subtyping may loop by instantiating unification variables with different types each time a given quantifier is eliminated. This does not happens very often in practice.

Circular typing proofs. The construction of circular typing proofs follows the same principle as for circular subtyping proofs. We create a general abstract sequent each time we encounter a fixpoint $Yx.t$, check whether it was already encountered before to end the proof, and if not we register the new hypothesis and continue the proof. Note however that the generalisation we preform for typing proofs is a bit more subtle. Indeed, if the type of $Yx.t$ does not contain any explicit quantifier on ordinals, we generalise infinite ordinals by decorating negative occurrences of types of the form $\mu X.A$ (and positive occurrences of types of the form $\nu X.A$). For example, this means that the sequent $\vdash Yx.t : \mu XA \rightarrow \nu Y \mu ZB$ is generalised to $\forall\alpha\forall\beta \vdash Yx.t : \mu_\alpha XA \rightarrow \nu_\beta Y \mu ZB$. However, when the type uses ordinal quantifiers we do not generalise infinite ordinals and only generalise ordinal variables (as for subtyping), assuming the given type already carries the proper ordinal annotation. In other words, if the user has not given explicit size information in the type of a program, then the first generalisation will have the effect of eliminating certain occurrences of ∞, intuitively replacing them with a smaller, finite ordinal.

Breadth-first search for typing fixpoint. As explained in the previous section, unrolling a fixpoint more than once is often necessary for building typing proofs. When mixed with unification, this requires a breadth-first proof search strategy. This means that when typing $Yx.t$, we first finish all the other branches of the proof, collecting as much as possible information about the type of $Yx.t$. By doing so, our experimentations have shown that we have more chances to instantiate unification variable in the expected way.

To implement the breadth-first strategy we first apply all the typing rules on the considered term, by delaying all the applications of the (Y) rule. In other words, we simply store the typing sequents corresponding to the (Y) rule in a list. We then iterate through all the stored sequents and first try to apply a possible induction hypothesis (there are none at the first stage of the search). For all the remaining sequents we perform a generalisation (as explained above) and store the general abstract sequent as an induction hypothesis. Finally, the next stage of breadth-first search can be launched. It consists in proving all the generalised sequents by first applying the I_k rule on them.

Generalisation and unification variables. In practice, the presence of unification variables in general abstract sequents often leads to failure or non-termination. Therefore, we instantiate constrained unification variables using their own constraints when we generalise a sequent to form a general abstract sequent. In particular, we fix type unification variables according to the set of variant constructors or record fields they carry in their states, and we instantiate ordinal unification variables with their lower bounds.

Nonetheless, unification variables that are not constrained are still kept in general abstract sequents. In this case, we need to introduce second order unification variables that may depend on the value of generalised ordinals. This is required as otherwise the unification variables would not be able to use the ordinals that are quantified over by the generalisation. For example, if a unification variable U occurs in a sequent $\vdash Yx.t : \mu X.A \to \nu Y.\mu Z.B$, then we introduce a new second order unification variable V with two ordinal parameters. The general abstract sequent is then $\forall\alpha\forall\beta \vdash Yt : (\mu_\alpha X.A \to \nu_\beta Y.\mu Z.B)[U := V(\alpha,\beta)]$, and U is instantiated with $V(\infty,\infty)$. Second order unification variables are dealt with in a very simple way, using projection when possible and imitation (i.e. constant value) when projection is not possible. For example, if we need to solve a constraint $\gamma \vdash V(\tau,\kappa) \leq \tau$ then we will only try to set V to the first projection and hence $V(\tau,\kappa) = \tau$.

Dealing with type errors. In our implementation, there are two different kinds of type errors: *clashes* which immediately stop the proof search, and loops that can be interrupted by the user. As only subtyping may loop, we can display the last encountered typing judgment in both cases, as well as the subtyping instance that failed to be proved. We can thus obtain a message like "t has type A and is used with type B".

For readability, it is important to note that it is never required to display choice operators in full. Indeed, we can limit ourselves to the name of the variable they bind, and the position of the variable it was substituted to in the source code. Note however that the error messages of the current prototype are not optimal. They have been optimised for the debugging of the prototype itself rather than for debugging programs written using the

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

Figure 21: Example involving dot projection (dual category).

prototype. We believe that we could improve error messages for it to be as easy (or as difficult) to debug type errors with our algorithm than with mainstream ML implementations. However, proving termination requires an extra effort for advanced examples.

11. Type annotations and dot notation.

Using the guidelines provided in the previous section, it is possible to build a satisfactory implementation. However, since the system is likely to be undecidable, we need to provide a way of annotating complex programs.

As we are considering a Curry style language, type annotations are not completely natural. Simple type coercions like $t : A$ can be added to the system without difficulty using the following rule.

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

However, such type annotations are often required to reference bound type variables, and a type abstraction constructor $\Lambda X.t$ is only natural in Church style calculi. A simple idea to solve the annotation problem in Curry style is to write annotations like the following.

$$\mathrm{let}\ \overline{X}\ \mathrm{such\ that}\ x : A(\overline{X})\ \mathrm{in}\ t$$

They allow the user to name a type (most of the time a choice operator) by pattern matching the type of the bound variable x. During type checking, x is replaced by a choice operator which carries its type T. It is thus possible to pattern match T against $A(\overline{X})$ to obtain the value of the variables of $\overline{X}$ (this is relatively simple to implement). For example, a fully annotated identity function can be written as follows.

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

Moreover, this kind of annotations may be used to define dot notation on existential types. It may be used to replace the usual dot notation for abstract types. Indeed, if a λ-variable x has type $\exists X \exists Y A(X,Y)$ then we can access X and Y using the following.

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

As we use local subtyping when matching type, the implementation can easily search X_0 and Y_0 such that $\gamma \vdash t \in A(X_0, Y_0) \subset \exists X \exists Y A(X, Y)$. 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 style is too heavy and in this particular case, we prefer writing $x.X$ 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 variable because we substitute closed terms, types or ordinals to variables and renaming is never necessary in this case. 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 (see Figure 21).

Note that the syntactic sugar defined here for dot notation is limited as it only applies to variables. A more general dot notation such as $(f\ t).X$ would be more difficult to obtain (in particular in presence of effects), because it denotes a type that may contain a computation. Nonetheless, it is always possible to name $f\ t$ using a let-binding.

12. 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 generalised into a form of higher-order pattern matching. However, our system allows us to avoid computing the variance of higher-order expressions (which is not completely trivial), thanks to the absence of syntactic covariance condition on our inductive and coinductive types.

Dependent types and proofs of programs. One of our motivations for this work is the integration of subtyping to the realisability models defined in a previous work by Rodolphe Lepigre [25]. 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 observationally 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. We also expect that such a feature will allow for a better type inference, and thus simplify the development of complex programs.

Completeness without quantifiers. Our algorithm seems terminating for the fragment without $\forall$ and $\exists$ quantifiers. We are actually able to prove its completeness 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.

A larger complete Subsystem. If we succeed in proving the completeness of the fragment of the system without quantifiers, the next step would be to see if we can gain completeness with some restriction on quantification (like ML style polymorphism). More generally, the cases leading to non-termination of subtyping should be better understood to avoid it as much as possible and try to produce better error messages when the system is interrupted.

References

[1] Martín Abadi, Luca Cardelli, and Gordon Plotkin. Types for the Scott numerals. `http://lucacardelli.name/papers/notes/scott2.ps`, 1993.

[2] Andreas Abel. *Foetus - Termination Checker for Simple Functional Programs*, 1998. `http://www2.tcs.ifi.lmu.de/~abel/foetus.pdf`.

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

[4] 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.

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

[6] A. Arnold and D. Niwiński. *Rudiments of μ-calculus*. Studies in Logic and the Found. Elsevier, 2001.

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

[8] 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.

[9] 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.

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

[11] Stephen Dolan and Alan Mycroft. Polymorphism, subtyping, and type inference in mlsub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 60–72. ACM, 2017.

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

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

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

[15] 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.

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

[17] 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.

[18] Pierre Hyvernat. The size-change termination principle for constructor based languages. *Logical Methods in Computer Science*, 10(1), 2014.

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

[20] 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, 2006.
[21] Dexter Kozen and Rohit Parikh. A decision procedure for the propositional μ-calculus. In *Logic of Programs*, volume 164 of *Lecture Notes in Computer Science*, pages 313–325. Springer, 1983.
[22] Jean-Louis Krivine. Un algorithme non typable dans le système F. *CRAS*, 304, 1987.
[23] 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.
[24] R. Lepigre and C. Raffalli. SubML implementation, 2015. `https://lama.univ-savoie.fr/subml`.
[25] Rodolphe Lepigre. A Classical Realizability Model for a Semantical Value Restriction. In Peter Thiemann, editor, *25th European Symposium on Programming, ESOP 2016*, volume 9632 of *Lecture Notes in Computer Science*, pages 476–502. Springer, 2016.
[26] The Coq development team. *The Coq proof assistant reference manual.* LogiCal Project, 2004.
[27] John C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76(2):211–249, 1988.
[28] John C. Mitchell, Sigurd Meldal, and Neel Madhav. An extension of standard ML modules with subtyping and inheritance. In *POPL*, pages 270–278. ACM Press, 1991.
[29] Ulf Norell. Dependently typed programming in agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, 2009.
[30] Miche Parigot. Un récurseur fortement normalisable et typable pour les entiers de Scott. Private communication, 1992.
[31] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
[32] François Pottier. *Synthèse de types en présence de sous-typage: de la théorie à la pratique*. PhD thesis, Université Paris 7, July 1998.
[33] C. Raffalli. Type checking in system F^η. In *Prépublication 98-05a du LAMA*, 1998.
[34] C. Raffalli. The PhoX proof assistant, 2008. `https://lama.univ-savoie.fr/~raffalli/phox.html`.
[35] C. Raffalli. *The PML programming language*, 2012. `https://lama.univ-savoie.fr/tracpml`.
[36] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 408–423. Springer-Verlag, 1974.
[37] 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.
[38] Jorge Luis Sacchini. Well-founded sized types in the calculus of (co)inductive constructions. 2015.
[39] Luigi Santocanale. A calculus of circular proofs and its categorical semantics. In *FoSSaCS*, volume 2303 of *Lecture Notes in Computer Science*, pages 357–371. Springer, 2002.
[40] Luigi Santocanale. From parity games to circular proofs. *Electr. Notes Theor. Comput. Sci.*, 65(1):305–316, 2002.
[41] Robert S. Streett and E. Allen Emerson. An automata theoretic decision procedure for the propositional mu-calculus. *Information and Computation*, 81(3):249 – 264, 1989.
[42] J. Tiuryn and P. Urzyczyn. The subtyping problem for second-order types is undecidable. *Inf. Comput.*, 179(1):1–18, 2002.
[43] 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.
[44] 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.