Practical Subtyping for Curry-Style Languages

RODOLPHE LEPIGRE, LAMA, CNRS, Univ. Savoie Mont Blanc and Inria, LSV, CNRS, Univ. Paris-Saclay
CHRISTOPHE RAFFALLI, LAMA, CNRS, Univ. Savoie Mont Blanc and IMERL, FING, UdelaR

We present a new, syntax-directed framework for Curry style type-systems with subtyping. It supports a rich set of features, and allows for a reasonably simple theory and implementation. The system we consider has sum and product types, universal and existential quantifiers, inductive and coinductive types. The latter two may carry size invariants that can be used to establish 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. One of the key ideas is to 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. Termination is then obtained using a standard (semantic) normalisation proof. To demonstrate the practicality of the system, we provide an implementation accepting all the examples discussed in the paper.

CCS Concepts: • **Theory of computation → Type theory**; • **Software and its engineering → Functional languages**;

Additional Key Words and Phrases: syntax-directed type system, Curry-style quantifiers, subtyping, polymorphism, existential types, inductive and coinductive sized types, choice operators, realizability semantics, reducibility candidates, circular proofs, size change principle, dot notation for abstract types.

ACM Reference format:
Rodolphe Lepigre and Christophe Raffalli. 2017. Practical Subtyping for Curry-Style Languages. *ACM Trans. Program. Lang. Syst.* 1, 1, Article 1 (July 2017), 58 pages.
https://doi.org/0000001.0000001

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 [42], or on sum types like polymorphic variants [24]. 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. They were simply not conceived with the aim of supporting a general form of subtyping.

In this paper, we propose a new framework for the design and for the implementation of type systems with subtyping. Our goal being the development 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 [29]. For example, we can express the fact that the usual *map* function on lists is size-preserving.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
0164-0925/2017/7-ART1 $15.00
https://doi.org/0000001.0000001

The system can be implemented using standard unification techniques, thanks to its syntax-directed typing and subtyping rules (Figures 6 and 7 page 25). In particular, only one typing rule applies for each term constructor, and at most one subtyping rule applies for every two type constructors (up to commutation). As a consequence, the only challenges that are faced when implementing the system are related to the handling of unification variables, and to the construction of circular proofs (heuristics are discussed in Section 9 page 50).

Church-style versus Curry-style quantification

There are two different approaches to quantifiers in type systems. The first one, called Church-style, is most widely used in proof systems such as Coq or Agda, as it usually preserves the decidability of type-checking. In particular, Church-style quantifiers are reflected in the syntax of terms using type abstraction and type application, as shown in the following typing rules.

$$\frac{\Gamma \vdash t : A \qquad X \notin FV(\Gamma)}{\Gamma \vdash \Lambda X.t : \forall X.A}\ \forall_i\text{-Church} \qquad\qquad \frac{\Gamma \vdash t : \forall X.A}{\Gamma \vdash t\ B : A[X := B]}\ \forall_e\text{-Church}$$

Despite its good properties, the Church-style discipline places the burden of writing type annotations on the user. That is why functional programming languages such as Haskell or OCaml usually rely on the second approach, called Curry-style, in which no annotation is required.

$$\frac{\Gamma \vdash t : A \qquad X \notin FV(\Gamma)}{\Gamma \vdash t : \forall X.A}\ \forall_i\text{-Curry} \qquad\qquad \frac{\Gamma \vdash t : \forall X.A}{\Gamma \vdash t : A[X := B]}\ \forall_e\text{-Curry}$$

However, this lighter syntax comes at the expense of the decidability of type-checking, at least in an unrestricted setting.[1] In this paper, we consider a language with full Curry-style quantifiers, and argue that decidability and practicality are two different questions.

Local subtyping for syntax-directed typing rules in Curry-style

Let aside the undecidability of type-checking, Curry-style systems are hard to implement because their typing rules are not syntax-directed. In particular, it is not possible to decide what typing rule should be applied by only looking at the shape of the term. To solve this issue, we rely on a new approach based on subtyping, which eliminates the problematic $\forall_e$-Curry and $\forall_i$-Curry rules. In general, subtyping already makes $\forall_e$-Curry redundant since it can be derived as follows.

$$\frac{\Gamma \vdash t : \forall X.A \qquad \dfrac{\dfrac{}{A[X := B] \subseteq A[X := B]}\ \subseteq\text{-refl}}{\forall X.A \subseteq A[X := B]}\ \forall_l}{\Gamma \vdash t : A[X := B]}\ \subseteq$$

However, $\forall_i$-Curry cannot be derived using the usual notion of subtyping. Indeed, it only allows for a very weak (and even useless) introduction rule, obtained with the following derivation.

$$\frac{\Gamma \vdash t : A \qquad \dfrac{\dfrac{}{A \subseteq A}\ \subseteq\text{-refl} \qquad X \notin FV(A)}{A \subseteq \forall X.A}\ \forall_r}{\Gamma \vdash t : \forall X.A}\ \subseteq$$

The problem with this rule is that the eigenvariable constraint $X \notin FV(\Gamma)$ cannot be expressed with subtyping, as it does not involve typing contexts. In other words, there is no hope of deriving the general $\forall_i$-Curry rule with the usual notion of subtyping and the following rule.

$$\frac{A \subseteq B \qquad X \notin FV(B)}{A \subseteq \forall X.B}\ \forall_r$$

[1] Languages like Haskell or OCaml usually restrict polymorphism or enforce type annotations to preserve decidability.

To solve this problem, we introduce *local subtyping* judgements of the form $\Gamma \vdash t \in A \subseteq B$. Thanks to the presence of a typing context, the eigenvariable constraints can be expressed with subtyping, and the $\forall_i$-Curry rule can thus be derived as follows.

$$\dfrac{\Gamma \vdash t : A \qquad \dfrac{\dfrac{}{\Gamma \vdash t \in A \subseteq A}= \qquad X \notin FV(\Gamma)}{\Gamma \vdash t \in A \subseteq \forall X.A}\forall_r}{\Gamma \vdash t : \forall X.A}\subseteq$$

As far as the authors know, there is no other system in which $\forall_i$-Curry can be derived with subtyping. That is however essential for obtaining syntax-directed typing rules.

Surprisingly, the terms that appear in our local subtyping judgements do not play any role in the syntax.[2] Their only purpose is to provide a natural semantics to local subtyping. Indeed, the judgement $\Gamma \vdash t \in A \subseteq B$ is interpreted as "$\Gamma \vdash t : A$ implies $\Gamma \vdash t : B$", and it is not clear what would be a suitable interpretation if t was omitted.[3] Therefore, we choose to keep the term t, and rather eliminate the typing context Γ by pushing the information it contains into t. This can be achieved using choice operators inspired by Hilbert's Epsilon and Tau functions [28], which keep the eigenvariable constraints implicit. As a consequence, our typing and local subtyping judgements are respectively of the forms $t : A$ and $t \in A \subseteq B$.

Replacing free variables with choice operators

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.[4] Intuitively, $\varepsilon_{x\in A}(t \notin B)$ is a counterexample to the fact that $\lambda x.t$ has type $A \to B$. It can thus be used to built the following unusual typing rule for λ-abstractions.

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

Note that it can be read as a proof by contradiction, since its premise is only valid when there is no term u of type A such that $t[x := u]$ does not have type B. This exactly corresponds to the usual realizability interpretation of the arrow type.

Thanks to this new approach, terms remain closed throughout typing derivations, thus suppressing the need for typing contexts.[5] 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 typing rule

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

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

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

In fact, the typing rules of our system (Figure 6 page 24) are presented in a slightly more general way. Indeed, most of our typing rules have a local subtyping judgement as a premise.

Similar choice operator techniques can be applied to free type variables for handling universal and existential quantifiers. To this aim, we introduce choice operators $\varepsilon_X(t \in A)$ and $\varepsilon_X(t \notin B)$, interpreted as types satisfying the denoted properties. In particular, $\varepsilon_X(t \notin B)$ denotes a type such that t does not have type $B[X := \varepsilon_X(t \notin B)]$. In other words, it is a counterexample to the fact that

[2] They are actually used to handle a specific membership type constructor $t \in A$ in the type system of PML_2 [39].
[3] This particular point will be discussed in relation with related work (see Section 1 page 10).
[4] Our model being based on reducibility candidates [25, 26], the interpretation of a type is never empty.
[5] We will still use a form of context to store ordinals assumed to be nonzero (see Section 4 page 22).

t has type $\forall X.B$. As a consequence, the typing rule for the introduction of the universal quantifier is subsumed by the following local subtyping rule.

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

The usual eigenvariable constraint is not required here, since t does not have any free variables thanks to choice operators. It is replaced by the fact that t cannot contain $\varepsilon_X(t \notin B)$, as this would lead to an invalid "cyclic" term, with t appearing in its own definition. Following our presentation, a typing rule corresponding to $\forall_i$-Curry may be derived as follows.

$$\frac{t : A[X := \varepsilon_X(t \notin A)] \qquad \dfrac{\dfrac{}{t \in A[X := \varepsilon_X(t \notin A)] \subseteq A[X := \varepsilon_X(t \notin A)]}\ =}{t \in A[X := \varepsilon_X(t \notin A)] \subseteq \forall X.A}\ \forall_r}{t : \forall X.A}\ \subseteq$$

Note that our system does not rely on the general ($\subseteq$) rule used above, since it is not syntax-directed (it applies to any term t). However, our typing rules (Figure 6 page 24) somehow incorporate the ($\subseteq$) rule, since some of them have a local subtyping judgement as a premise.

In conjunction with local subtyping, our choice operators for types enable the derivation of valid permutations of quantifiers and connectors. For instance, Mitchell's containment axiom [41]

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

can be easily derived in the system. Another important consequence of these innovations is that our system does not require a (non-syntax-directed) transitivity rule such as

$$\frac{t \in A \subseteq B \qquad t \in B \subseteq C}{t \in A \subseteq C}$$

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 $t : A$, $t : A \subseteq B$ and $t : B \subseteq C$, which may help the system to find the right instantiation for unification variables. As such annotations are seldom required in the implementation and no incompleteness appeared in practice due to the lack of a transitivity rule, we feel that this rule might be admissible in the system.[6]

Implicit covariance condition for (co-)inductive types

Inductive and coinductive types are generally handled using types $\mu X.F(X)$ and $\nu X.F(X)$, respectively denoting the least and the greatest fixed point 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, covariance is automatically enforced when traversing the types during the construction of local subtyping judgements. For instance, if F is not covariant, then it is not be possible to derive $\mu X.F(X) \subseteq \nu X.F(X)$ or $\mu X.F(X) \subseteq F(\mu X.F(X))$.[7] As far as the authors know, this is the first work in which covariance is not explicitly required for inductive and coinductive types.

Sized types and circular subtyping proofs

In this paper, our inductive and coinductive types carry an ordinal number κ to form sized types $\mu_\kappa X.F(X)$ and $\nu_\kappa X.F(X)$ [4, 29, 55]. They intuitively correspond to κ iterations of F on the types $\bot = \forall X.X$ and $\top = \exists X.X$ respectively. In particular, if t has type $\mu_\kappa X.F(X)$ then there must be $\tau < \kappa$ such that t has type $F(\mu_\tau X.F(X))$, and 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 $\nu_\kappa X.F(X)$ as the intersection of all the $F(\nu_\tau X.F(X))$ for $\tau < \kappa$. These definitions are monotonous in

[6] We did not attempt to prove the admissibility of the transitivity rule because we do not need this result here.

[7] We write $A \subseteq B$ for $\varepsilon_{x \in A}(x \notin B) \in A \subseteq B$, which encodes usual subtyping as a local subtyping judgement.

κ, even if F is not covariant. For cardinality reasons, this implies that there is an ordinal ∞ from which these constructions are stationary. As a consequence, we have $F(\mu_\infty X.F(X)) \subseteq \mu_\infty X.F(X)$ and $\nu_\infty X.F(X) \subseteq F(\nu_\infty X.F(X))$, which are sufficient for the correctness of our subtyping rules. Note that $\mu_\infty X.F(X)$ and $\nu_\infty X.F(X)$ only correspond to the least and greatest fixed points of F when it is covariant. If F is not covariant then these stationary points are not fixed points.

In this paper, we introduce a uniform induction rule for local subtyping, expressed in a new circular proof framework (Section 3 page 17). It is able to deal with many inductive and coinductive types at once, but accepts circular proofs that are not well-founded. To solve this problem, we rely on the size change principle [36], which allows us to check for well-foundedness a posteriori. Our system is able to deal with subtyping relations involving mixed inductive and coinductive types. For example, it is able to derive $\mu X.\nu Y.F(X,Y) \subseteq \nu Y.\mu X.F(X,Y)$ when F is covariant.

When we restrict ourselves to types without universal and existential quantifiers, our experiments tend to indicate that the implementation of our system is in some sense complete. We have a proof sketch for completeness in the case where arrow types are also removed, but we failed to prove it in their presence. The main problem is that we used semantics to prove the completeness of the implementation, and is is not clear for which semantics our system should be shown complete in the presence of function types. For instance, axioms like $A \subseteq (\forall X.X) \to B$ make sense in certain models, but they cannot be proved in our system.

Terminating recursion and circular typing proofs

In our system, recursive programs can be handled using circular typing proofs, in a similar way that sized types are handled with circular local subtyping proofs. General recursion is enabled by extending the language with a fixed point combinator $Yx.t$, reduced as $Yx.t \succ t[x := Yx.t]$. It is typed using the very simple unfolding rule displayed below.

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

This rule clearly induces circularity as a proof of $Yx.t : A$ will require a proof of $Yx.t : A$ (provided that x appears in t). Again, as there is no guarantee that the produced circular proofs are well-founded, we rely on the size change principle [36]. Given its simplicity, our system is surprisingly powerful. Note that to type-check certain complex programs, it is sometimes necessary to unfold a fixed point several times to obtain a well-founded circular proof (see Section 8 page 46).

One of the major advantages of our presentation is that it allows for a good integration of the termination check into the type system, both in the theory and in the implementation. Indeed, we do not prove the termination of programs directly, but rather show that their circular typing proofs are 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 apply the size change principle to size information extracted from the circular structure of our proofs in a precisely defined way (see Section 3 page 17). Contrary to usual approaches, we do not require the semi-continuity condition for recursion.[8]

Quantification over ordinals

As types can carry ordinal sizes, it is natural to allow quantification over the ordinals themselves. We can thus type the usual *map* function with

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

[8]This particular point will be discussed in relation with related work (see Section 1 page 10).

where $\mathrm{List}(A, \alpha)$ is the type of lists of size α with elements in A defined as $\mu_\alpha L.[\mathrm{Nil} \mid \mathrm{Cons}\ \mathrm{of}\ A \times L]$. 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 a partitioning function whose type is of the following form.

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

This ensures that the produced lists cannot be bigger that the input list, and we can then define the sorting function in the usual way. In our implementation, the termination of simple functions is automatically derived, without requiring specific type annotations from the user.

In this paper, the language of the ordinals that can be represented in the syntax is very limited. As in [54], 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 syntax of ordinals with function symbols like maximum or addition.

Implementation of the system

Typing and subtyping are likely to be undecidable in our system, as it contains John Mitchell's System F_η [13]. System F and System F_η both have undecidable typing and subtyping [61, 63–65], but it is an open problem whether this is also the case for all their normalising extensions.

Moreover, we believe that there are no practical, complete algorithms for extensions of System F or System F_η like ours. Instead, we propose an incomplete semi-algorithm that may fail, or even diverge on a typable program. In practice we almost never meet non termination, but even in such an eventuality, the user can still interrupt the program and obtain a relevant error message. Indeed, our algorithm can only diverge when checking a local subtyping judgement. 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 called SubML, which is available online [37]. Aside from a few subtleties (see Section 9 page 50), the implementation is straightforward and remains very close to the typing and subtyping rules of the system (Figure 7 page 25, Figure 12 page 41, and Figure 13 page 41). 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 SubML is complete (i.e., may accepts all typable programs), provided that enough type annotations are given. In practice, the required amount of annotations seems to be reasonably small (see Section 5 page 27, and Section 8 page 46). Overall, the system provides a similar user experience to languages like OCaml or Haskell. In fact, these languages also require type annotations for advanced features like polymorphic recursion.

SubML provides literate programming features inspired by the PhoX language [49]. They can notably be used to generate LaTeX documents. In particular, the examples presented in the paper (Sections 4 page 22, Section 5 page 27, and Section 8 page 46), including proof trees, have been generated by SubML and are therefore machine checked. Many other program examples (more than 4800 lines of code) are provided with the implementation, to support our claim that the system is indeed usable in practice. SubML can be compiled from source at https://github.com/rlepigre/subml, or tried online using the provided web editor[9] at https://rlepigre.github.io/subml.

Type annotations and abstract types

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. In particular, 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 the subterms of polymorphic terms with their types.

As our system relies on choice operators, it never actually 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 types, and make them available to the user for giving type annotations. As an example, we can fully annotate the polymorphic identity function in the following way.

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

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 it the dot notation for existential types [14]. It enables the encoding of a module system including abstract types, using records and existential quantification. For example, a signature for isomorphisms can be encoded as follows.

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

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

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

As our system never infers polymorphic or existential types, we can rely on the names that were chosen by the user for the bound variables. This approach to abstract types was already considered by Abadi, Gonthier and Werner [2], although they did not go as far as defining the dot notation. In any case, choice operators seem to yield a simpler system than previous work like [15].

Original applications

In addition to classical examples, our system allows for applications that we find very interesting (see Section 5 page 27, and Section 8 page 46). 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 combining 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 $\mathbb{N}_S \to \mathbb{N}_S$.

[9] This online version is compiled to Javascript using js_of_ocaml (https://ocsigen.org/js_of_ocaml).

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 λ-calculus recursors for $\mathbb{N}_S$, without relying on any native form of recursion. It was shown to the second author by Michel Parigot [44], and we adapted it to other algebraic data types. This shows that Scott encoding can be used to program in a strongly normalising system, with the expected asymptotic complexity.

We also discovered a surprising λ-calculus coiterator for streams, when they are encoded using an existentially quantified type S, representing an internal state.

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

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

As far as we known, the strongly normalising version of our system (see Section 4, page 22) is the first language based on the λ-calculus, without specific recursors (i.e., only using pure λ-terms), that can be used to encode inductive and coinductive data types with operations having the expected asymptotic complexity. Indeed, in languages such as System T, recursors that are encoded as pure λ-terms are usually neither directly typable nor strongly normalising.

List of contributions

This paper contains many contributions, which all play an important role in obtaining a system that is reasonably simple, and yet powerful enough to match (and in some aspects outmatch) existing implementations of statically typed functional programming languages. As the proposed system is certainly undecidable, we can only try to convince the reader of its practicality by providing a substantial number of examples. As a consequence, we chose to consider a system with many different features, in which relevant examples can be expressed. This led to the long list of contributions displayed below, which might seem too long for a single paper. We however argue that they are necessary for our practicality argument to be convincing enough.

- Syntax-directed framework for Curry-style languages with inductive and coinductive type.
- Local subtyping relation for handling quantifiers in subtyping.
- Type system without any specific typing rule for handling quantifiers.
- Syntax with choice operators allowing a clearer semantics and a simpler implementation.
- Circular proof framework with a correctness criterion based on the size change principle.
- Using circular proofs for handling inductive and coinductive types with subtyping.
- Using circular proofs to prove the termination of recursive programs.
- Semantic proof of normalisation by well-founded induction on circular typing derivations.
- No explicit covariance condition for inductive and coinductive types.
- Absence of an explicit semi-continuity condition for recursion.
- Use of choice operators for implementing the dot notation for abstract types in modules.

Outline of the paper

After discussing related work in Section 1 (Related work and other approaches, page 10), we start by considering a general framework for circular proofs. In Section 2 (Syntactic ordinals and size change matrices, page 13), we define a syntax for representing (vectors of) ordinals, whose sizes can be compared using matrices. This formalism is then used to give a correctness criterion for (potentially non-well-founded) circular proofs in Section 3 (Circular proofs and size change principle, page 17). This criterion is based on the size change principle, and ensures that we can reason by induction on the "circular" structure of proofs.

In a second part of the paper, we consider a strongly normalising version of the system, that does not have general recursion. It is defined in Section 4 (Language and type system, page 22), and relies on circular subtyping proofs for handling inductive and coinductive types. Surprising examples of programs are given in Section 5 (Recursion without fixed point for Scott encoding, page 27), and the properties of the system are established in Section 6 (Realizability semantics, page 30). A model based on reducibility candidates is constructed, and it is used to show that the system is logically consistent (Theorem 6.24 page 38), strongly normalising (Theorem 6.25 page 38), and type safe (Theorem 6.27 page 38). These results follow from the adequacy lemma (Theorem 6.23 page 36), which establishes that the type system is compatible with the model.

In a third part of the paper, we extend the language with general recursion. The new system is defined in Section 7 (Fixed point and termination, page 39), and it relies on circular typing proof to establish the termination of programs. The properties of the system are mostly preserved (Theorem 7.17 page 45, and Theorem 7.18 page 46). However, the definition of the model needs to be changed slightly because strong normalisation (in the usual sense) is compromised by the fixed point 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 or case analyses). Several examples of provably terminating, recursive programs are then given in Section 8 (Terminating examples, page 46), and a list of all the relevant examples that we implemented is available on the web page of our prototype [37]. Note that it can be used online at https://rlepigre.github.io/subml/.

The last part of the paper is about practical matters. In Section 9 (Type-checking Algorithm, page 50), we discuss the implementation of the system, and in particular the heuristics that were used. In Section 10 (Type annotations and dot notation, page 54), we discuss the issue of type annotations in Curry-style systems. We also show that choice operators yield a natural way of implementing the dot notation for abstract types in modules. Finally, Section 11 (Perspectives and Future Work, page 55) contains a list of possible improvements, and future explorations.

Table of contents

1 RELATED WORK AND OTHER APPROACHES.

The language presented in this paper is an extension of John Mitchell's System F_η [13], which itself extends Girard and Reynolds's System F [25, 53] with subtyping. Unlike previous work like [6, 46] that only dealt with inductive types, our system supports mixed inductive, coinductive, polymorphic and existential types. As mentioned in [16, Section 6.4], the handling of existentials is generally not straight-forward. Here however, their treatment is dual to that of polymorphic types, be it in the syntax or in the semantics. As there does not seem to exist any other system supporting all the features of ours, we consider related work for each individual component.

Type-checking algorithms for variants of System F

Let us first focus on the part of our language that deals with the types of system F, seeing our type system as a reformulation of System F_η. Since the work of Damas and Milner on rank-1 (or prenex) polymorphism [17], quite a few results have been obtained for arbitrary rank polymorphism. For example, the approach of Le Botlan and Rémy in [35] is to consider a system that is in between Church-style and Curry-style (terms contain type abstractions but no type applications). However, there are only few works on full Curry-style languages. We can cite the work of Rémy and Cretin [51], which requires type annotations for the arguments of applications and let definitions, the work of Peyton Jones et al. [45], and the work of Dunfield and Krishnaswami [21].

Overall, these approaches yield rather complex type systems, which seem to be less intuitive than ours. This is mainly due to the fact that they provide an algorithm (i.e., a terminating type checking procedure) for a decidable subsystem (or variant) of Curry-style System F. In this paper, we take an opposite direction, which is to trade the decidability of type checking for a simple semi-algorithm (i.e., a type checking procedure that may fail or even diverge on typable programs). Following this direction seems natural because the theoretical complexity of usual (decidable) type systems, which is generally (at least) exponential, is almost never met in practice. This could indicate that part of their complex machinery remains most of the time unused, and is hence not strictly necessary. The type system that is presented in this paper can be easily implemented, by introducing unification variables for unknown types in a naive way:[10]

- directly following the typing and subtyping rules of the system,
- introducing a fresh unification variables U for every unknown type,
- setting $U := A$ to prove local subtyping judgement like $t \in U \subseteq A$ or $t \in A \subseteq U$.

Our experiments showed that this approach works well in practice. Moreover, the user experience is not very different from working with meta-variables or implicit arguments in Coq [40] or Agda [43]. Although our type-checking procedure may diverge (in theory), it works very well in practice, and does not require more annotation than the systems mentioned above.

Type-checking algorithms for sized (co-)inductive types

Our type system relies on sized types [29], which means that our inductive and coinductive types carry ordinal numbers. Such a technique is widespread for handling induction [10, 11, 27, 32, 55] and even coinduction [4, 5, 54], in settings where termination must be enforced (e.g., in proof assistants). The most important difference between this paper and previous works precisely lies in the handling of recursion. In particular, size relations between ordinal parameters are usually checked when using recursion, which yields typing rules such as the following.

$$\frac{\lambda x.t : \forall\alpha.(\mu_\alpha X.F(X)) \to (\mu_{\alpha+1} X.F(X))}{Yx.t : \mu_\infty X.F(X)} \qquad \frac{\lambda x.t : \forall\alpha.(\nu_\alpha X.F(X)) \to (\nu_{\alpha+1} X.F(X))}{Yx.t : \nu_\infty X.F(X)}$$

[10] Most of the heuristics presented in Section 9 page 50 are not necessary for the System F fragment.

In these rules, F must be contravariant in its parameter for the first rule, and covariant for the other. But this is not enough for correctness, and a semi-continuity condition is also required. For example, if X is not free in B then types like $F(X) = X \to B$ or $F(X) = B \to X$ are accepted, but more complex types like $F(X) = (X \to B) \to B$ and $F(X) = ((X \to B) \to B) \to B$ are rejected.

In this paper, sized types and recursion are handled in a completely orthogonal way. Ordinal sizes are only manipulated in the subtyping rules for inductive and coinductive types, while recursion is handled separately, using the following simple typing rule which simply unfolds the fixed point combinator (in the same type).

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

This leads to a system with a simple presentation, which relies on a general concept of circular proofs. In particular, no semi-continuity condition is imposed. It is replaced by an implicit condition on the term t. Using this approach, we accept some examples that do not satisfy the usually required semi-continuity condition.

Moreover, the systems described in [5, 54] only provide a simple rule for recursion, while the implementations of systems like Agda [43], MiniAgda or Coq [40] use more general termination criterion (e.g., using the size change principle [36]) and handle mutually recursive definitions. This more general recursion scheme is often not formally described, with a few exceptions like [3, 30, 31]. When this is the case, we believe that the obtained systems are far more complex than ours. Indeed, our circular proof framework makes it easier to integrate the termination checker into the formal description of the system and its normalisation proof. This technique has also been applied to the implementation of the PML_2 language [50].

Nevertheless, our system is not completely satisfactory, and it could be improved 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 [31]. Another possible direction of improvement would be to extend the syntax of ordinals with function symbols like addition or maximum.

Systems involving circular proofs

In this paper, we consider a system with well-founded circular proofs, which means that our circular proofs need to satisfy an external criterion to be considered valid. Systems in which circular proofs are unrestricted exist, but they are not really related to our work.

Circular proofs in our sense were first studied in the context of temporal logic in [56–60], and were also applied to first-order logic [12]. However, using such techniques in the context of a subtyping relation seems completely new, although our subtyping rules have similarities with the deduction rules of the modal μ-calculus. In the context of program termination, there are some systems with circular proofs that have a cut elimination procedure [22, 23] (this can be seen as a form of termination). But as mentioned earlier, using circular typing proofs with a simple unfolding rule for the fixed point combinator seems also new.

Infinite proofs are also used in the context of linear logic by Baelde, Doumane and Saurin [7, 8, 20], but their presentation is not oriented toward the Curry-Howard correspondence. Moreover, their notion of infinite proof includes, but is not limited to circular proofs. This leads to an elegant theoretical presentation, but it is unclear how these techniques could be applied in practice.

Systems involving choice operators

Choice operators such as Hilbert's Epsilon and Tau [28] (which can encode each other in classical logic) are a well-known alternative to Skolemization for eliminating quantifiers. They are used in theorem proving. For instance, in [18] the authors show how to use them at a reduced cost. They are also used in linguistics [52, 62], which links them to the λ-calculus via Montague's approach of semantics. Choice operators were also considered for type theories [9], and they are standard in systems such as HOL or PVS.

However, choice operators are rarely used in typed programming languages, although this seems to be a very simple and natural approach. They were used to handle existential types (type abstraction for modules) in the work of Abadi, Gonthier and Werner [2]. However, we are not aware of any work (except ours) in which choice operators are used in the place of all free variables (including λ-variables). This allows us to only consider closed objects, which means that we never need to rename bound variables. For instance, this would simplify the formalisation of our system in a proof assistant. It also allows for a simple encoding of dot notation for abstract types, using syntactic sugar relying on the name of the bound variable (this was not noticed in [2]).

Systems with subtyping

Subtyping has been extensively studied in the context of ML-like languages, starting with the work of Amadio and Cardelli [6]. Recent work includes the MLsub system [19], 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 [47]. However, we are not aware of any system that is as expressive as ours for a Curry-style extension of System F with subtyping. 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 4 and 5).

The most important difference between our system and previous work is that we were able to handle quantifiers using subtyping exclusively. In particular, the usual introduction rule for the universal quantifier is not included in our system, which was not possible in previous approaches to subtyping. The main advantage of our approach using local subtyping, is that it can be easily extended to other computationally irrelevant connectives (i.e., connectives that are not reflected in the syntax of terms). Examples of such connectives include inductive and coinductive types, as well as the membership and restriction types of the PML_2 system [50] (used to encode a form of dependent types and a form of refinement types).

Alternative approaches to subtyping with polymorphism

To conclude the related work section, we will now discuss the necessity of local subtyping and choice operators in the handling of polymorphism with subtyping. First, there exist several way of handling quantifiers with subtyping. For examples, the approach used in [45] consist in defining the following subtyping rule, which can be used to derive Mitchell's containment axiom.

$$\frac{\Gamma \vdash A \subseteq C_1 \to \ldots \to C_n \to B \qquad X \notin A, C_1, \ldots, C_n}{\Gamma \vdash A \subseteq C_1 \to \ldots \to C_n \to \forall X.B}$$

However, this rule cannot be used to derive the usual introducing rule for the universal quantifier (called $\forall_i$-Curry on page 2), because of the constraint on A. This is also the case in [51], where types are transformed into prenex form to get Mitchell's containment axiom.

It is actually possible to use typing contexts in subtyping rules, in such a way that Mitchell's containment axiom and $\forall_i$-Curry are both derivable. In practice, such a context can simply be a list of types. The following two rules rely on this principle.

$$\frac{\Gamma \vdash A \subseteq B \qquad X \notin \Gamma}{\Gamma \vdash A \subseteq \forall X.B} \qquad\qquad \frac{\Gamma, C \vdash C \subseteq A \qquad \Gamma, C \vdash B \subseteq D}{\Gamma \vdash A \to B \subseteq C \to D}$$

This approach was used in an unpublished note of the second author [48]. Unfortunately, judgments of the form $\Gamma \vdash A \subseteq B$ have an intricate semantics of the form "$\Gamma \vdash t : A$ implies $\Gamma \vdash t : B$ for all term t", which involves a substitution to interpret the judgment $\Gamma \vdash t : A$.

Then, a simpler alternative is to use a form of local subtyping without choice operators (and thus with typing contexts), which would lead to subtyping judgments of the form $\Gamma \vdash t \in A \subseteq B$. However with this approach, the formulation of the semantics is still painful because it requires substitutions and valuations. For instance, the adequacy lemma must be states as follows.

LEMMA. *If the judgment $\Gamma \vdash t \in A \subseteq B$ is derivable, then for all type variable interpretations φ, if $t\sigma \in [\![A]\!]_\varphi$ for all substitutions $\sigma \in [\![\Gamma]\!]_\varphi$ (i.e., $\sigma(x) \in [\![\Gamma(x)]\!]_\varphi$ for all $x \in dom(\Gamma)$), then $t\sigma \in [\![B]\!]_\varphi$ for all substitutions $\sigma \in [\![\Gamma]\!]_\varphi$.*

On the contrary, when local subtyping is used in conjunction with choice operators, we obtain a very clear and elegant semantics: "$[\![t]\!] \in [\![A]\!]$ implies $[\![t]\!] \in [\![B]\!]$". This technique does not impose any syntactic overhead, apart from the choice operators themselves.

2 SYNTACTIC ORDINALS AND SIZE CHANGE MATRICES

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 [29]. 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 $(e_1, \dots, e_n)$ of length $|\overline{e}| = n$. The concatenation of two vectors $\overline{x}$ and $\overline{y}$ will be denoted $\overline{x}.\overline{y}$. Moreover, there will sometimes be implicit length constraints on 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 $\mathcal{V}_\mathcal{O} = \{\alpha, \beta, \dots\}$.*

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

In syntactic ordinals of the form $\varepsilon_{\overline{\alpha} < \overline{w}} P(\overline{\alpha})_i$, the variables of $\overline{\alpha}$ are bound in $P(\overline{\alpha})$ but not in $\overline{w}$. Moreover, we enforce $1 \le i \le |\overline{\alpha}|$ and $|\overline{\alpha}| = |\overline{w}| = |P|$, where $|P|$ is the arity of the predicate P.

Syntactic ordinals are built using the constant ∞, a successor symbol, and *ordinal choice operators* (or *witnesses*) of the form $\varepsilon_{\overline{\alpha} < \overline{w}} P(\overline{\alpha})_i$. Intuitively, the vector $\overline{\tau}$ defined as $\tau_i = \varepsilon_{\overline{\alpha} < \overline{w}} P(\overline{\alpha})_i$ denotes syntactic ordinals that are pointwise smaller than $\overline{w}$, and such that "$P(\overline{\tau})$ is true" (this will be made formal in Definition 2.6 page 14). In the upper bound $\overline{w}$, one can use the notation $\mathcal{O}$ to mean that there is no size constraint on the corresponding variable. In other words, $\mathcal{O}$ denotes an ordinal that is bigger than every syntactic ordinal, and hence is not a syntactic ordinal itself.

CONVENTION 2.3. *We will use the notation $\overline{\varepsilon}_{\overline{\alpha} < \overline{w}} P(\overline{\alpha})$ for the vector $(\varepsilon_{\overline{\alpha} < \overline{w}} P(\overline{\alpha})_i)_{1 \le i \le |\overline{\alpha}|}$. In the case where $|\overline{\alpha}| = |\overline{w}| = |P| = 1$ we will write $\varepsilon_{\alpha < w} P(\alpha)$ for both $\overline{\varepsilon}_{\overline{\alpha} < \overline{w}} P(\overline{\alpha})$ and $\varepsilon_{\overline{\alpha} < \overline{w}} P(\overline{\alpha})_1$.*

In the semantics, the symbol ∞ will be interpreted by an (actual) ordinal Ω. Its concrete value does not matter for the definition of our circular proof framework, and it may be chosen differently depending on the application. In this paper, Ω will eventually be instantiated with an ordinal large enough to ensure the convergence of all the fixed points corresponding to our inductive and coinductive types (see Section 6 page 30). Nonetheless, Ω will not be the biggest ordinal of our semantics, since larger ones may be represented in the syntax using the successor symbol.[11]

DEFINITION 2.4. *Let Ω be a fixed, but arbitrary ordinal. We denote $[\![\mathcal{O}]\!]$ the ordinal $\Omega + \omega$, which is also the set of all the (actual) ordinals of our semantics, and the interpretation of $\mathcal{O}$.*

[11] In practice, ordinals such as $\infty + 1$ and further successors of ∞ will never be considered.

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 2.5. *The set of* parametric syntactic ordinals $\mathcal{O}^*$ *is obtained by extending the language of syntactic ordinals given in Definition 2.2 with (actual) ordinals* $o \in [\![\mathcal{O}]\!]$.

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

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

Definition 2.6. *To interpret predicate symbols, we require a function (or valuation)* $[\![-]\!]$ *such that* $[\![P]\!] \in [\![\mathcal{O}]\!]^{|P|} \rightarrow \{0, 1\}$ *for all* $P \in \mathcal{P}$. *The semantics of closed (vectors of) parametric syntactic ordinals is then defined inductively as follows.*

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

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

In the ordinal witness case, $(<)$ *denotes pointwise ordering, and* $\overline{0}$ *is a vector of* 0 *ordinals. Moreover, as there may be several possible values for* $\overline{o}$*, we will sometimes distinguish* models *for which the choice is made in a specific way. If* $\mathcal{M}$ *is such a model, we will use the notation* $[\![\kappa]\!]^{\mathcal{M}}$ *for the corresponding interpretation. When no particular model is specified, it is considered fixed, but arbitrary.*

Lemma 2.7. *Let* $\mathcal{M}_0$ *be a model,* $\overline{\varepsilon}_{\overline{\alpha} < \overline{w}} P(\overline{\alpha})$ *be a vector of ordinal witnesses, and* $\overline{o}$ *be a vector of (actual) ordinals of the corresponding size. If* $\overline{o} < [\![\overline{w}]\!]^{\mathcal{M}_0}$ *and* $[\![P]\!](\overline{o}) = 1$*, then there is a model* $\mathcal{M}_1$ *such that* $[\![\overline{w}]\!]^{\mathcal{M}_1} = [\![\overline{w}]\!]^{\mathcal{M}_0}$ *and* $[\![\overline{\varepsilon}_{\overline{\alpha} < \overline{w}} P(\overline{\alpha})]\!]^{\mathcal{M}_1} = \overline{o}$.

Proof. We start by considering the height function $h : \mathcal{O}^* \rightarrow \mathbb{N}$ on parametric syntactic ordinals, defined inductively in the following way.

$$\begin{aligned} h(\infty) = h(\mathcal{O}) = h(o) &= 0 & h(\kappa + 1) &= 1 + h(k) \\ h(\kappa_1, \dots, \kappa_n) &= \max(h(\kappa_1), \dots, h(\kappa_n)) & h(\varepsilon_{\overline{\alpha} < \overline{w}} P(\overline{\alpha})_i) &= 1 + h(\overline{w}) \end{aligned}$$

We then construct the interpretation of the parametric syntactic ordinal τ in the model $\mathcal{M}_1$ by induction on $h(\tau)$. We take $[\![\tau]\!]^{\mathcal{M}_1} = [\![\tau]\!]^{\mathcal{M}_0}$ for every τ such that $h(\tau) < h(\overline{\varepsilon}_{\overline{\alpha} < \overline{w}} P(\overline{\alpha}))$, which includes the elements of $\overline{w}$. We then take $[\![\overline{\varepsilon}_{\overline{\alpha} < \overline{w}} P(\overline{\alpha})]\!]^{\mathcal{M}_1} = \overline{o}$ and complete the definition by marking arbitrary choices for the remaining ordinal witnesses. □

In the syntax, we will compare syntactic ordinals τ and κ using an ordering relation $\kappa \leq \tau$, and a strict ordering relation $\kappa < \tau$. They will be defined in terms of a third (ternary) relation $\kappa \leq_i \tau$ with $i \in \mathbb{Z}$. It will be specified by a deduction rule system involving *ordinal contexts*, which will gather syntactic ordinals assumed to be non-zero.

Definition 2.8. Ordinal contexts *are finite sets of syntactic ordinals. They are represented using lists generated by the following BNF grammar.*

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

$$\frac{i \le 0}{\gamma \vdash \kappa \le_i \kappa} = \qquad \frac{\gamma \vdash \kappa \le_{i+1} \tau}{\gamma \vdash \kappa + 1 \le_i \tau} s_l \qquad \frac{\gamma \vdash \kappa \le_{i-1} \tau}{\gamma \vdash \kappa \le_i \tau + 1} s_r$$

$$\frac{\gamma, w_j \vdash w_j \le_{i-1} \tau \qquad w_j \ne O}{\gamma, w_j \vdash \varepsilon_{\overline{\alpha} < \overline{w}} P(\overline{\alpha})_j \le_i \tau} \varepsilon \qquad \frac{\gamma \vdash w_j \le_i \tau \qquad w_j \ne O}{\gamma \vdash \varepsilon_{\overline{\alpha} < \overline{w}} P(\overline{\alpha})_j \le_i \tau} \varepsilon_w$$

Fig. 1. Rules for ordinal ordering and strict ordering.

In practice, it will never be useful to store syntactic ordinals of the form $\tau + 1$ or ∞ in ordinal contexts, as they are necessarily non-zero.

Definition 2.9. *For all $i \in \mathbb{Z}$, the relation $(\le_i)$ on syntactic ordinals is defined, under an ordinal context γ, using the deduction rules of Figure 1. We then use the notation $\kappa \le \tau$ for $\kappa \le_0 \tau$, and the notation $\kappa < \tau$ for $\kappa \le_1 \tau$.*

Intuitively, the relation $\kappa \le_i \tau$ can be understood as "$\kappa + i \le \tau$" if $i \ge 0$, and as "$\kappa \le \tau + (-i)$" otherwise. The corresponding deduction rule system 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 the rules (ε) and (ε_w) both apply, it is always better to use (ε) since it yields a lower index, and thus proves more judgements according to the second item of the following lemma.

Lemma 2.10. *For every ordinal contexts γ and δ, for every syntactic ordinals κ_1, κ_2 and κ_3, and for every integers i and j we have:*

(1) if $\gamma \vdash \kappa_1 \le_i \kappa_2$ then $\gamma, \delta \vdash \kappa_1 \le_i \kappa_2$,
(2) if $\gamma \vdash \kappa_1 \le_i \kappa_2$ and $j \le i$ then $\gamma \vdash \kappa_1 \le_j \kappa_2$,
(3) if $\gamma \vdash \kappa_1 \le_i \kappa_2$ and $\gamma \vdash \kappa_2 \le_j \kappa_3$ then $\gamma \vdash \kappa_1 \le_{i+j} \kappa_3$.

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 $\gamma \vdash \kappa_1 \le_i \kappa_2$ and $\gamma \vdash \kappa_2 \le_j \kappa_3$. If the last applied rule on either side is $(=)$, then we have $\kappa_1 = \kappa_2$ and $i \le 0$ or $\kappa_2 = \kappa_3$ and $j \le 0$. 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 $\gamma \vdash \kappa \le_{i+j+1} \kappa_3$ and thus $\gamma \vdash \kappa_1 \le_{i+j} \kappa_3$. 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 $\kappa_1 = \varepsilon_{\overline{\alpha} < \overline{w}} P(\overline{\alpha})_m$. By induction hypothesis, we get $\gamma \vdash w_m \le_{i+j-1} \kappa_3$ if we applied the (ε) rule, or $\gamma \vdash w_m \le_{i+j} \kappa_3$ if we applied the (ε_w) rule. In both cases this implies $\gamma \vdash \kappa_1 \le_{i+j} \kappa_3$. If the last rule used on the right is (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). The only remaining case, that is not covered by previous ones, is when the (s_r) rule is applied on the left and the (s_l) rule is applied on the right. It immediately follows from the induction hypothesis. □

We will now show that our ordering relations $(\le_i)$ are compatible with the semantical interpretation of syntactic ordinals. This is summarised in the following lemma.

Lemma 2.11. *Let γ be a closed ordinal context, κ_1 and κ_2 be closed syntactic ordinals, and i be an integer such that $\gamma \vdash \kappa_1 \le_i \kappa_2$ is derivable. If $[\![\tau]\!] \ne 0$ for all $\tau \in \gamma$, then $[\![\kappa_1]\!] + i \le [\![\kappa_2]\!]$ when $i \ge 0$ and $[\![\kappa_1]\!] \le [\![\kappa_2]\!] + (-i)$ when $i \le 0$.*

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

We are now going to consider the formalism that will be used to relate our syntactic ordinals to the size-change principle [36] in the following sections. The main idea will be to represent the size information contained in the circular structure of our proofs using matrices. We will then be able to easily compose size information using matrix product.

Definition 2.12. *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. The neutral elements of* (min) *and* $(\circ)$ *are respectively* ∞ *and* 0, *and the absorbing element of* $(\circ)$ *is* ∞.

$$0 \circ 0 \;=\; 0 \qquad\qquad x \circ \infty \;=\; \infty \circ x \;=\; \infty \qquad\qquad -1 \circ x \;=\; x \circ -1 \;=\; -1 \quad \textit{if} \quad x \neq \infty$$

Intuitively, the value -1 will indicate a size decrease, the value 0 will indicate a size stagnation (or rather, no size increase), and the value ∞ will denote the absence of size information.

Definition 2.13. *A* size-change matrix *is simply a matrix with coefficients 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 $C_{i,j} = \min_{1 \leq k \leq m} A_{i,k} \circ B_{k,j}$.*

Note that product on size-change matrices exactly corresponds to the usual matrix product, expressed with the operations of our semi-ring $(\{-1, 0, \infty\}, min, \circ)$. In particular, it enjoys the usual associativity property.

Lemma 2.14. *The size-change matrix product is associative.*

Proof. Immediate from the semi-ring axioms. □

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

Definition 2.15. *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 $A_{i,j} = 0$.*

In this paper, we will take $(X, \leq)$ to be the set of our syntactic ordinals $\mathcal{O}$, equipped with the relations $(\leq)$ and $(<)$ given in Definition 2.9 page 15.

Lemma 2.16. *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 $A_{i,k} = B_{k,j} = -1$, or 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 $A_{i,k} = B_{k,j} = 0$ and thus $z_j \leq y_k \leq x_i$. □

3 CIRCULAR PROOFS AND SIZE CHANGE PRINCIPLE

We will now introduce an abstract notion of *circular proof*, and a related notion of *well-foundedness* defined in terms of the size change principle of Lee, Jones and Ben Amram [36]. Although we rely on this well-known technique, this work requires a complete reformulation of the original size change principle, which is generally used as a criterion for program termination. As a consequence, we do not rely on the results of [36] directly, but we follow the same ideas.

In this paper, we propose to represent proofs as directed acyclic graphs, and to label their edges with size-change matrices. We can hence track the evolution of the size of our syntactic ordinals throughout proofs, and rely on a simple condition (the same as for the usual size change principle) to make sure that there is some structural decrease. Of course, a proof will only be considered correct if such a decrease can be established. We will later use circular subtyping proofs to handle inductive and coinductive types (Section 4 page 22), and circular typing proofs to ensure the termination of recursive programs (Section 7 page 39).

Our circular proof framework is parameterised by so-called *abstract judgements*, their deduction rules, and their semantics. In this paper, they will correspond to typing or local subtyping judgements, equipped with their respective deduction rules and interpretations. However, we believe that our framework could be applied to other systems involving a notion of size.

Definition 3.1. *A language of* abstract judgements *is given by a set $\mathcal{J}$ of judgements parameterised by syntactic ordinals. Every $J \in \mathcal{J}$ has an arity $|J|$ corresponding to the number of parameters it expects (possibly 0).*

Intuitively, an abstract judgement can be seen as a form of predicate, whose validity depends on the truth of the judgement it denotes. In the following, such predicates will be used to build ordinal witnesses according to the previous section. For instance, we will work with and manipulate syntactic ordinals of the form $\varepsilon_{\overline{\alpha}<\overline{\kappa}}\neg J(\overline{\alpha})_i$.

Definition 3.2. *Given a language of abstract judgements $\mathcal{J}$, we build a language of predicates $\mathcal{P}$ that contains a predicate $\neg J$ of arity $|J|$ for all $J \in \mathcal{J}$. We then obtain a fixed language of (parametric) syntactic ordinals by instantiating Definitions 2.2 and 2.5 page 13 using $\mathcal{P}$.*

Example 3.3. In this paper, we will consider abstract judgements of the forms $J(\overline{\alpha})$ = "$A \subseteq B$" and $J(\overline{\alpha})$ = "$t : A$", respectively corresponding to (usual) subtyping judgement and to typing judgements. Therefore, we will have predicates of the form "$\neg(A \subseteq B)$" and "$\neg(t : A)$", and syntactic ordinals will include choice operators of the forms "$\varepsilon_{\overline{\alpha}<\overline{\kappa}}(\neg(A \subseteq B))_i$" and "$\varepsilon_{\overline{\alpha}<\overline{\kappa}}(\neg(t : A))_i$". Note that the syntax of our type will include syntactic ordinals. They will thus be defined mutually inductively through Definition 3.2. Their semantics will also need to be mutually inductively defined to give an interpretation to predicates constructed using abstract judgments.

In the semantics, an abstract judgement will be interpreted by a predicate over (actual) ordinals. It will intuitively correspond to the provability of the abstract judgement, for every possible instantiation of the ordinal parameters.

Definition 3.4. *Let $\mathcal{J}$ be a language of abstract judgements. Every abstract judgement $J \in \mathcal{J}$ of arity n is interpreted by a function $[\![J]\!] : [\![\mathcal{O}]\!]^n \longrightarrow \{0, 1\}$. The predicates over ordinals built according to Definition 2.6 page 14 are then interpreted as $[\![\neg J]\!](\overline{\alpha}) = 1 - [\![J]\!](\overline{\alpha})$.*

Definition 3.5. *An* abstract sequent *$\gamma \vdash J(\overline{\kappa})$ is built using an ordinal context γ, an abstract judgement $J \in \mathcal{J}$ and syntactic ordinals $\overline{\kappa} \in \mathcal{O}^{|J|}$. We say that the abstract sequent $\gamma \vdash J(\overline{\kappa})$ is valid if we have $[\![J]\!]([\![\overline{\kappa}]\!]) = 1$ whenever $[\![\tau]\!] \neq 0$ for all $\tau \in \gamma$.*

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

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

Fig. 2. Generalisation rule and induction rule for general abstract sequents.

To relate size-change matrices to abstract sequents, we introduce a notion of *ordinal constraints*. They will allow us to concisely represent, in the form of a sequence of indices, a conjunction of strict relations between the ordinals of a given vector.

Definition 3.6. *A list of* ordinal constraints *of arity n is given by a partial function C from $\{1, \ldots, n\}$ to itself. For all ordinals $\overline{o} \in [\![O]\!]^n$, we denote $C(\overline{o})$ the vector of size n defined as*

$$C(\overline{o})_i = \begin{cases} o_j & \textit{when } i \in dom(C) \\ [\![O]\!] & \textit{otherwise} \end{cases}$$

for all $1 \leq i \leq n$. We say that C is satisfied by $\overline{o}$ if and only if $o_i < C(\overline{o})_i$ for all $i \in dom(C)$.

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

Definition 3.7. *A* general abstract sequent *is an abstract sequent whose syntactic ordinals have been quantified over. It is thus of the form $\forall\overline{\alpha}(\gamma \vdash C(\overline{\alpha}) \Rightarrow J(\overline{\alpha}))$, where γ is an ordinal context only containing variables of $\overline{\alpha}$, C is a list of ordinal constraints of arity $|\overline{\alpha}|$, and J is an abstract judgement such that $|\overline{\alpha}| = |J|$. We say that the general abstract sequent $\forall\overline{\alpha}(\gamma \vdash C(\overline{\alpha}) \Rightarrow J(\overline{\alpha}))$ is valid if $[\![J]\!](\overline{o}) = 1$ for all $\overline{o} \in [\![O]\!]^n$ such that $o_i \neq 0$ whenever $\alpha_i \in \gamma$, and such that C is satisfied by $\overline{o}$.*

Definition 3.8. *A* circular deduction system *is given by a set of deduction rules whose premises and conclusions are abstract sequents, together with the two rules of Figure 2.*

The aim of the *generalisation rule* (G) is to prove an abstract sequent using a general abstract sequent. In particular, the ordinal constraints used in its first premise should be satisfied in the conclusion (see the remaining premises). The *induction rule* (I_k) is used to prove a general abstract sequent using itself as an hypothesis (this is the meaning of the square brackets). A natural number k (unique in a proof) is used to keep track of the originating induction rule.

Note that (I_k) and (G) are the only rules that manipulate general abstract sequents. The induction rule alone is responsible for the circular structure of proofs. In particular, it allows for clearly invalid proofs, as shown in the following example.

Example 3.9. Every abstract sequent $\varnothing \vdash J$ with $|J| = 0$ can be proved by the (invalid) circular proof displayed bellow.

$$\dfrac{\dfrac{\dfrac{[\forall *(\varnothing \vdash * \Rightarrow J)]_0}{\varnothing \vdash J}\ \mathrm{G}}{\forall *(\varnothing \vdash * \Rightarrow J)}\ \mathrm{I}_0}{\varnothing \vdash J}\ \mathrm{G}$$

$$\dfrac{\begin{array}{c}\dfrac{\begin{array}{c}\vdots \qquad B_2 \\ \hline \forall\overline{\alpha_2}(\delta \vdash D(\overline{\alpha_2}) \Rightarrow K(\overline{\alpha_2}))\end{array} \qquad (\delta[\overline{\alpha_2} := \overline{\tau}], \delta' \vdash \tau_i < D(\overline{\tau})_i)_{i\in dom(D)}}{\delta[\overline{\alpha_2} := \overline{\tau}], \delta' \vdash K(\overline{\tau})}\,\mathrm{G} \quad B_1 \\ \vdots \\ \gamma[\overline{\alpha_1} := \overline{\kappa}] \vdash J(\overline{\kappa}) \qquad \text{where } \overline{\kappa} = \overline{\varepsilon}_{\overline{\alpha_1}<C(\overline{\alpha_1})}\neg J(\overline{\alpha_1})\end{array}}{\forall\overline{\alpha_1}(\gamma \vdash C(\overline{\alpha_1}) \Rightarrow J(\overline{\alpha_1}))}\,\mathrm{I}_k$$

Fig. 3. Illustration for the construction of an edge of the call graph (see Definition 3.11).

After generalising over the empty vector of ordinals (denoted $*$), the new hypothesis is directly applied. As there is no structural decrease along the loop, the proof is definitely invalid.

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 rely on the size-change principle 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 3.10. *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 general abstract sequents. Moreover, 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 3.11. *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_1 has one outgoing edge for each of its premises. Such a premise may be proved by a block B_2 directly above B_1, or by an hypothesis introduced by an I_k rule at the beginning of a block B_2 (possibly B_1 itself).*

Every edge (B_1, B_2) of a call graph is labelled by a size-change matrix M. To give its definition, we need to remark that a premise of a block necessarily uses the (G) rule, since it is 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 3, where only the relevant premises are depicted. The $|\overline{\alpha_1}| \times |\overline{\alpha_2}|$ matrix M attached to the edge (B_1, B_2) is then defined as $M_{i,j} = -1$ when $\delta[\overline{\alpha_2} := \overline{\tau}], \delta' \vdash \tau_j < \kappa_i$ is derivable, $M_{i,j} = 0$ when only $\delta[\overline{\alpha_2} := \overline{\tau}], \delta' \vdash \tau_j \leq \kappa_i$ is derivable, and $M_{i,j} = \infty$ otherwise.

As the edges of a call graph are labelled with matrices, any path in its transitive closure can 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 thus be computed in finite time by composing edges until saturation.

Definition 3.12. *We say that a proof is* well-founded *(or satisfies the size-change principle) if every idempotent loop (i.e., a closed path labelled with an idempotent matrix) in the transitive closure of its call graph has at least one -1 on the diagonal of its label.*

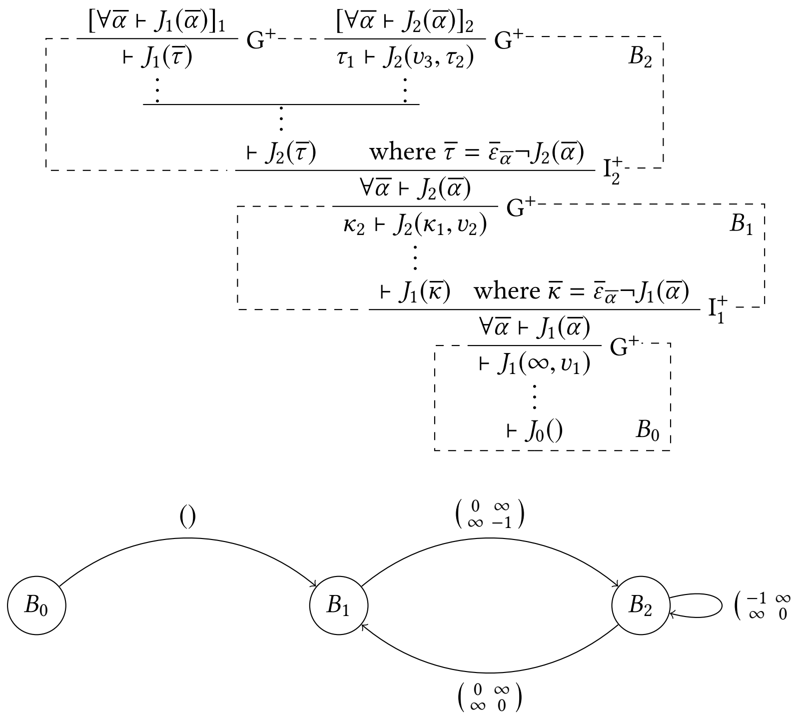


Fig. 4. Example of a circular proof with the corresponding call graph.

To illustrate the previous definitions, we will now consider two examples of circular proofs. The structure of the former is the same as the well-founded circular subtyping proof of Figure 10 page 28. The latter corresponds to the invalid circular proof of Example 3.9 page 18.

Example 3.13. We now consider the example of circular proof given in the upper part of Figure 4 page 20. For simplicity, the 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 is decompose into three blocks B_0, B_1 and B_2, and the corresponding call graph is displayed below the proof. 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 the proof is well-founded since every idempotent loop is labelled with a matrix having at least one -1 on its diagonal.

Example 3.14. The circular proof of Example 3.9 page 18 is built of two blocks. The former contains only the conclusion, and the latter contains the rest of the proof. The corresponding call graph (which is equal to its transitive closure) has one edge going from the conclusion block to the upper block, and one edge going from the upper block to itself. As both edges (including the loop) are labelled with the empty matrix, the proof is not well-founded.

To conclude this section, we will give a general result establishing the correctness of circular proof systems, provided that the deduction rules for abstract sequents are correct. First, we need to show that the typing rules involving general abstract sequents are correct.

Lemma 3.15. *The two deduction rules of Figure 2 page 18 are locally correct. In other words, if the immediate premises of such a rule are semantically valid, then so is its conclusion.*

Proof. For the (G) rule, we can assume that all the syntactic ordinals of $\gamma[\overline{\alpha} := \overline{\kappa}], \delta$ are interpreted by non-zero ordinals, since otherwise the conclusion is immediately true. As a consequence, we know that the ordinals of $[\![\overline{\kappa}]\!]$ that are mapped to variables of γ are non-zero. Moreover, the validity of the right premises tells us that the list of ordinal constraints C is satisfied by $[\![\overline{\kappa}]\!]$. We can thus conclude using the validity of the first premise.

For the (I_k) rule, we consider the semantics of the choice operators in the premise. By definition, if the conclusion of the sequent is not valid, then there is a counterexample that the choice operator can use. However, such counterexample cannot exist as this would imply that the premise of the rule is not valid. Hence, the conclusion of the sequent must be valid. □

Note that in the previous lemma, the correctness of the (I_k) rule does not involve the new hypothesis that is introduced (i.e., the bracketed sequent). The justification for such hypotheses is globally handled by our notion of well-founded proof (Definition 3.12 page 19).

Theorem 3.16. *Let us consider a circular deduction system whose deduction rules over abstract sequents are assumed to be 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 correct (by hypothesis and by Lemma 3.15 page 20), the call graph must contain cycles. We will thus unroll the proof to exhibit an infinite branch that will imply the existence of an infinite, decreasing sequence of ordinals.

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 the 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}, labelled with a matrix M_i such that $\overline{o_{i+1}} <_{M_i} \overline{o_i}$. The first element of the sequence $(B_0, \overline{o_0}, \mathcal{M}_0)$ satisfies these conditions. In particular, the conclusion of B_0 is false in the model $\mathcal{M}_0 = \mathcal{M}$, independently of any ordinal. Moreover, the matrix that will label the edge linking B_0 to B_1 will necessarily be empty.

Let us now suppose that the sequence has been constructed up to index i, and define the triple $(B_{i+1}, \overline{o_{i+1}}, \mathcal{M}_{i+1})$. If $i = 0$ then B_i ends with an abstract sequent that is false in the model $\mathcal{M}$, and in that case we take $\mathcal{M}_{i+1} = \mathcal{M}$. If $i \neq 0$ then the conclusion of B_i is a general abstract sequent, which means that it must end with the following rule.

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

By construction, we know that $\forall\overline{\alpha}(\gamma \vdash C(\overline{\alpha}) \Rightarrow J(\overline{\alpha}))$ is false in $\mathcal{M}_i$, and that $\overline{o_i}$ is a counterexample. This means that $[\![\gamma[\overline{\alpha} := \overline{o_i}]]\!]^{\mathcal{M}_i}$ only contains positive ordinals, that C is satisfied by $\overline{o_i}$, and that $[\![J]\!](\overline{o_i}) = 0$. Thus, using Lemma 2.7 page 14, we can define $\mathcal{M}_{i+1}$ to be a model with $[\![\overline{\kappa}]\!]^{\mathcal{M}_{i+1}} = \overline{o_i}$. This establishes that the abstract sequent $\gamma[\overline{\alpha} := \overline{\kappa}] \vdash J(\overline{\kappa})$ is false in the model $\mathcal{M}_{i+1}$.

As all the deduction rules for abstract sequents are supposed correct, at least one premise of B_i must be false in $\mathcal{M}_{i+1}$. The first rule at the top of such a leaf must be G, since it is the only deduction rules having a general abstract sequent as a premise.

$$\frac{\forall\overline{\beta}(\delta \vdash D(\overline{\beta}) \Rightarrow K(\overline{\beta})) \qquad (\delta[\overline{\beta} := \overline{\tau}], \delta' \vdash \tau_i < D(\overline{\tau})_i)_{i \in dom(D)}}{\delta[\overline{\beta} := \overline{\tau}], \delta' \vdash K(\overline{\tau})}\ \mathrm{G}$$

As the conclusion of this rule is false in $\mathcal{M}_{i+1}$, we know that $[\![\delta[\overline{\beta} := \overline{\tau}], \delta']\!]^{\mathcal{M}_{i+1}}$ only contains positive ordinals and that $[\![K]\!]([\![\overline{\tau}]\!]^{\mathcal{M}_{i+1}}) = 0$. According to Lemma 2.11 page 15, the premises of the form $\delta[\overline{\beta} := \overline{\tau}], \delta' \vdash \tau_i < D(\overline{\tau})_i$ must be true. Therefore, $\forall\overline{\beta}(\delta \vdash D(\overline{\beta}) \Rightarrow K(\overline{\beta}))$ must be false in the model $\mathcal{M}_{i+1}$. We can thus take B_{i+1} to be the block corresponding to that premise, and $\overline{o_{i+1}}$ to be $[\![\overline{\tau}]\!]^{\mathcal{M}_{i+1}}$, which is indeed a counterexample to $\forall\overline{\beta}(\delta \vdash D(\overline{\beta}) \Rightarrow K(\overline{\beta}))$.

By definition, the call graph must contain an edge linking B_i to B_{i+1}. It is labelled with a matrix M_i, and we need to show that $\overline{o_{i+1}} <_{M_i} \overline{o_i}$ to complete 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 the value of $(M_i)_{m,n}$. If it is equal to ∞ then there is nothing to prove. If it is equal to -1 (resp. 0) then there is a proof of $\delta[\overline{\beta} := \overline{\tau}], \delta' \vdash \tau_n < \kappa_m$ (resp. $\delta[\overline{\beta} := \overline{\tau}], \delta' \vdash \tau_n \leq \kappa_m$), and hence Lemma 2.11 page 15 gives us $[\![\tau_n]\!]^{\mathcal{M}_{i+1}} < [\![\kappa_m]\!]^{\mathcal{M}_{i+1}}$ (resp. $[\![\tau_n]\!]^{\mathcal{M}_{i+1}} \leq [\![\kappa_m]\!]^{\mathcal{M}_{i+1}}$). We can thus conclude that $o_{i+1,n} < o_{i,m}$ (resp. $o_{i+1,n} \leq o_{i,m}$) 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.

To conclude, we will use the same argument as in [36, Theorem 4], and rely on Ramsey's theorem for pairs.[12] Intuitively, this theorem states that for any colouring on sorted pairs of natural numbers (with finitely many colours), there is an infinite subset of the natural numbers whose sorted pairs all have the same colour. For all $0 \leq i < j$, we define $M_{i,j}$ to be the matrix $M_i M_{i+1} \ldots M_{j-1}$. 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 (Lemma 2.14 page 16) and to the definition of $M_{i,j}$, this implies that $MM = M_{u_0,u_1} M_{u_1,u_2} = M_{u_0,u_2} = M$.

Finally, we can use Lemma 2.16 page 16 to obtain $\overline{o_j} <_M \overline{o_i}$ 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 $(o_{u_i,k})_{i\in\mathbb{N}}$, which is obviously contradictory. □

4 LANGUAGE AND TYPE SYSTEM

In this section, we consider a first, restricted version of our language and type system. It does not provide general recursion, but enjoys strong normalisation (Theorem 6.25 page 38). Surprisingly, recursion is still possible for specific inductive data types, using λ-calculus recursors that are typable thanks to subtyping (examples are given in Section 5 page 27). The language is formed using three syntactic entities: terms, types and syntactic ordinals (defined in Section 2 page 13). 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 are not manipulated by the user directly. However, we will see in Section 7 page 39 that the type system can be naturally extended so that the user may express size invariants using ordinals.

Although the system is Curry-style (or implicitly typed), terms, types and syntactic ordinals are defined mutually inductively due to the presence of choice operators in their syntax.

Definition 4.1. *Let $\mathcal{V}_\Lambda = \{x, y, z, \ldots\}$, $\mathcal{V}_\mathcal{F} = \{X, Y, Z, \ldots\}$ 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*

[12] Ramsey's theorem for pairs. If f is a mapping from $\{(i, j) \in \mathbb{N} \times \mathbb{N} \mid i < j\}$ to a finite set X, then there exists an infinite set $Y \subseteq \mathbb{N}$ such that f is constant on $\{(i, j) \in Y \times Y \mid i < j\}$.

types are defined using the following two BNF grammars, where κ is a syntactic ordinal.

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

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

The syntactic ordinals are built according to Definition 3.2 page 17, using abstract judgments of the form $J(\overline{\alpha})$ = "$t : A$" and $J(\overline{\alpha})$ = "$t \in A \subseteq B$", where the ordinals $\overline{\alpha}$ may appear in the term t, and in the formulas A and B. Furthermore, we ask that terms of the form $\varepsilon_{x\in A}(t \notin B)$ do not contain free λ-variables. This means that only the bound variable x may appear in t, and terms like $\lambda y.\varepsilon_{x\in A}(y\,x \notin B)$ will thus be considered ill-formed.

The term language contains the usual syntax of the λ-calculus, extended with records, projections, constructors and pattern matching (the corresponding notations are explained in Convention 4.2 page 23). A term of the form $\varepsilon_{x\in A}(t \notin B)$ corresponds to a choice operator denoting a closed term u of type A such that $t[x := u]$ does not have type B. The restriction to closed choice is absolutely necessary for their interpretation in the semantics (see Definition 6.14 page 32). Note that the user will only have access to so-called pure terms, which do not contain choice operators (see Definition 6.1 page 30). Every type would otherwise be inhabited (see Remark 4.6 page 26).

Convention 4.2. *In our meta-language, we rely on the notation $\{(l_i = t_i)_{i\in I}\}$, where $I \subseteq \mathbb{N}$ is a finite set of natural numbers, to denote records concisely. 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, (extensible) 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.*

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 either be *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 (Theorem 6.27 page 38). Our inductive and coinductive types $\mu_\kappa X.A$ and $\nu_\kappa X.A$ carry size information in the form of a syntactic ordinals κ. When $\kappa = \infty$ they intuitively correspond to the least and greatest fixed points of the parametric type $X \mapsto A$. However, this is only the case if $X \mapsto A$ satisfies the usual covariance condition. Choice operators $\varepsilon_X(t \in A)$ and $\varepsilon_X(t \notin A)$, binding the propositional variable X in A, are also provided for types. As for the choice operators for terms, they correspond to witnesses of the property they denote, and they will be interpreted as such in the semantics. However, contrary to choice operators for terms, they do not require any syntactic restriction to be given a semantical interpretation.

Convention 4.3. *To lighten the syntax, we will use the following standard syntactic sugars. We will sometimes group binders, and write $\lambda x\,y.t$ for $\lambda x.\lambda y.t$, or $\forall X\,Y.A$ for $\forall X.\forall Y.A$. Moreover, we consider that binders have the lowest priority, which means that $\lambda x.x\,x$ and $\forall X.A \Rightarrow B$ are to be read as $\lambda x.(x\,x)$ and $\forall X.(A \Rightarrow B)$. We will use the notations $\mu X.A$ and $\nu X.A$ for $\mu_\infty X.A$ and $\nu_\infty X.A$, and we will sometimes use the letter F to denote a parametric type $X \mapsto A$ 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 mean $C_k \to \lambda x.t$. Finally, we will use $t.C_k$ as an abbreviation for the term $[t \mid C_k\ x \to x]$.*

We now define the reduction relation of our language, which contains β-reduction as well as rules for pattern matching and record projection. Terms corresponding to runtime errors are also reduced to a diverging term $\Omega = (\lambda x.x\,x)\ (\lambda x.x\,x)$ for termination to subsume type safety.

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

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

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

Fig. 5. Reduction rules of the language (without general recursion).

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

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

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

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

Fig. 6. Typing rules for the system without general recursion.

Definition 4.4. *The reduction relation* $(\succ) \subseteq \Lambda \times \Lambda$ *is defined as the contextual closure of the rules given in Figure 5. Its reflexive, transitive closure is denoted* $(\succ^*)$.

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 4.5. *Our type system manipulates typing judgements of the form* $\vdash t : A$ *meaning "t has type A", and local subtyping judgements of the form* $\gamma \vdash t \in A \subseteq B$ *meaning "if the ordinals of γ are non-zero and t has type A, then t also has type B". Usual subtyping judgements of the form* $\gamma \vdash A \subseteq B$*, meaning "if all the ordinals of γ are non-zero, every element of A is also an element of B", are then encoded as* $\gamma \vdash \varepsilon_{x\in A}(x \notin B) \in A \subseteq B$*. The typing and subtyping rules of the system are given in Figures 6 and 7 page 24.*

The first three typing rules of Figure 6 page 24 include (a variations of) the rules for the simply-typed λ-calculus given in the introduction (page 3). They are in fact slightly more general since $(\to_i)$ and (ε) contain a local subtyping judgment. As a consequence, the constraint on the shape of the type of their conclusion is transformed into a subtyping constraint. A similar technique is also applied to the introduction rules $(\times_i)$ and $(+_i)$ for sum and product types. Finally, the elimination rules $(\times_e)$ and $(+_e)$ for sums and products are not very surprising. It is nonetheless worth noting that the premise of $(\times_e)$ relies on an extensible product type with one field, which is the most general possible choice. Indeed, the term t can be an element of every (extensible) record type containing a field with label l_k of type A, which are all subtypes of $\{l_k : A; \dots\}$.

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

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

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

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

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

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

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

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

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

Fig. 7. Subtyping rules for the system without general recursion.

The local subtyping rules given in Figure 7 page 24 may seem complex, but they are actually rather straightforward. The (=) axiom corresponds to a form of reflexivity rule, which is implicitly compatible with α-equivalence (i.e., the types on the left and on the right of the inclusion need only be α-equivalent). The ordinal context γ that appears in the conclusion of the rules is only extended by the (μ_l) and (ν_r) rules, and it is only used directly by the second premise of the (μ_r) and (ν_l) rules. It is also easy to see that the rules for least and greatest fixed points are somewhat dual to each other, and this is also the case for quantifiers. Another important thing to remark is that the term t that appears in the conclusion of every local subtyping rule is only manipulated in the case of connectives with algorithmic contents (i.e., functions, sums and products). In this case, the term t is extended with an eliminator of the corresponding type. Application is used in the ($\to$) rule, case analysis is used in the (+) rule, and projection is used in the rules ($\times_{ss}$), ($\times_{se}$) and ($\times_{ee}$), which are related to product types. Note that the latter three are almost identical, but they are required for strict and extensible products to interact correctly. Finally, note that all the rules related to least fixed points (and dually, greatest fixed points) only perform an unrolling. The

$$\dfrac{\forall\overline{\alpha}(\gamma \vdash C(\overline{\alpha}) \Rightarrow A \subseteq B) \qquad (\gamma[\overline{\alpha} := \overline{\kappa}], \delta \vdash \kappa_i < C(\overline{\kappa})_i)_{i \in dom(C)}}{\gamma[\overline{\alpha} := \overline{\kappa}], \delta \vdash A[\overline{\alpha} := \overline{\kappa}] \subseteq B[\overline{\alpha} := \overline{\kappa}]}\ \mathrm{G}$$

$$\dfrac{\begin{array}{c}[\forall\overline{\alpha}(\gamma \vdash C(\overline{\alpha}) \Rightarrow A \subseteq B)]_k \\ \vdots \\ \gamma[\overline{\alpha} := \overline{\kappa}] \vdash A[\overline{\alpha} := \overline{\kappa}] \subseteq B[\overline{\alpha} := \overline{\kappa}]\end{array} \qquad \text{where } \overline{\kappa} = \overline{\varepsilon}_{\overline{\alpha} < C(\overline{\alpha})}(A \not\subseteq B)}{\forall\overline{\alpha}(\gamma \vdash C(\overline{\alpha}) \Rightarrow A \subseteq B)}\ \mathrm{I}_k$$

Fig. 8. Specialised induction and generalization rules for subtyping.

(μ_r) and (μ_r^∞) rules unroll a fixed point appearing on the right of the inclusion, and the former additionally requires an ordinal that is smaller than the one in the conclusion. On the other hand, the (μ_l) rule unrolls a fixed point appearing on the left of the inclusion by assuming the positivity of the current ordinal, and introducing a witness denoting a smaller ordinal. Intuitively, we will use circular local subtyping proofs to be able to apply the (μ_l) rule, and dually the (ν_r) rule, a finite but arbitrary number of times, to account for any particular concrete value of the ordinals.

Remark 4.6. *It is important that choice operators of the form $\varepsilon_{x \in A}(t \notin B)$ are not available to the user. Indeed, they can be used to inhabit any type A thanks to the following derivation.*

$$\dfrac{\dfrac{}{\vdash \varepsilon_{x \in A}(t \notin B) \in A \subseteq A}\,=}{\vdash \varepsilon_{x \in A}(t \notin B) : A}\,\varepsilon$$

Note that Theorem 6.24 page 38 will only apply to terms without choice operators.

Our typing and local subtyping judgements formally correspond to abstract sequents, following the terminology of Definition 3.5 page 17. In particulars, the abstract sequents corresponding to typing judgements always carry an empty ordinal context. We now consider the circular deduction system built using the typing and subtyping rules of our system.

Definition 4.7. *The type system of our language is the circular deduction system (see Definition 3.8 page 18) induced by the typing and subtyping rules of Figures 6 and 7 page 24. In fact, we restrict ourselves to general abstract sequents of the form $\forall\overline{\alpha}(\gamma \vdash C(\overline{\alpha}) \Rightarrow A \subseteq B)$. As a consequence, we will only apply the restricted generalisation and induction rules given in Figure 8.*

Note that in the system presented here, only (local) subtyping derivations have a circular structure. In fact, we only apply the (G) rule to usual subtyping judgements of the form $\gamma \vdash A \subseteq B$, which are encoded as $\gamma \vdash \varepsilon_{x \in A}(x \notin B) \in A \subseteq B$. This means that the (G) rule is always composed with the (S) rule (which proves a local subtyping judgement from an usual subtyping judgement). The use of (S) is essential, since the terms of our local subtyping judgements grow from bottom up in our subtyping rules. This means that, without (S), a local subtyping judgement of a given shape may not be encountered more than once in a given branch of a local subtyping derivation.

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.[13]

[13]This particular point will be discussed in Section 9 page 50, in relation with heuristics.

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

where

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

Fig. 9. Derivation of Mitchell's containment axiom.

Example 4.8 (containment axiom). In our system, it is possible to derive Mitchell's containment axiom [41], as well as one of its variations.

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

The (non-circular) derivation of the former is given in Figure 9 page 27. Note that the proof only relies on well-formed choice operators, whose definitions are not cyclic.

Example 4.9 (mixed induction and coinduction). To show that our syste is suitable for handling alternations of inductive and coinductive, we consider the following types S and L.

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

Both of these types represent particular streams of A's and B's. Intuitively, the elementes of S are streams that contain only finitely many A's, and the elements of L are streams that do not contain infinitely many consecutive A's. In our system, it is possible to prove $\mathrm{S} \subseteq \mathrm{L}$ using the circular proof displayed in Figure 10 page 28. Note that the block decomposition of the proof is given in Example 3.13 page 20. We can thus conclude that it is well-founded (and thus valid).

5 RECURSION WITHOUT FIXED POINT 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 encoding and the Scott encoding of natural numbers. Although they have little (if any) practical interest, they demonstrate well the use of polymorphism and fixed points. 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.

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

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 [34]. The latter requires some type annotations for our implementation to guess the correct instantiation of unifications variables. In particular, the type

$$
\begin{prooftree}
\AxiomC{$[\forall\alpha_0,\alpha_1(\vdash S_{\alpha_1} \subseteq \mathrm{G}(L_{\alpha_0}))]_1$}
\RightLabel{G}
\UnaryInfC{$\vdash S_{\kappa_5} \subseteq \mathrm{G}(L_{\kappa_4})$}
\RightLabel{S}
\UnaryInfC{$\vdash x_2.A \in S_{\kappa_5} \subseteq \mathrm{G}(L_{\kappa_4})$}
\AxiomC{$[\forall\alpha_0,\alpha_1(\vdash \mathrm{F}(S_{\alpha_1}) \subseteq \mathrm{G}(L_{\alpha_0}))]_2$}
\RightLabel{G}
\UnaryInfC{$\kappa_4 \vdash \mathrm{F}(S_{\kappa_5}) \subseteq \mathrm{G}(L_{\kappa_6})$}
\RightLabel{S}
\UnaryInfC{$\kappa_4 \vdash x_2.B \in \mathrm{F}(S_{\kappa_5}) \subseteq \mathrm{G}(L_{\kappa_6})$}
\RightLabel{ν_r}
\UnaryInfC{$\vdash x_2.B \in \mathrm{F}(S_{\kappa_5}) \subseteq L_{\kappa_4}$}
\RightLabel{$+$}
\BinaryInfC{$\vdash x_2 \in [\mathrm{A}\ \mathrm{of}\ S_{\kappa_5} \mid \mathrm{B}\ \mathrm{of}\ \mathrm{F}(S_{\kappa_5})] \subseteq [\mathrm{A}\ \mathrm{of}\ \mathrm{G}(L_{\kappa_4}) \mid \mathrm{B}\ \mathrm{of}\ L_{\kappa_4}]$}
\RightLabel{μ_r}
\UnaryInfC{$\vdash x_2 \in [\mathrm{A}\ \mathrm{of}\ S_{\kappa_5} \mid \mathrm{B}\ \mathrm{of}\ \mathrm{F}(S_{\kappa_5})] \subseteq \mathrm{G}(L_{\kappa_4})$}
\RightLabel{ν_l}
\UnaryInfC{$\vdash x_2 \in \mathrm{F}(S_{\kappa_5}) \subseteq \mathrm{G}(L_{\kappa_4})$}
\RightLabel{I_2}
\UnaryInfC{$\forall\alpha_0,\alpha_1(\vdash \mathrm{F}(S_{\alpha_1}) \subseteq \mathrm{G}(L_{\alpha_0}))$}
\RightLabel{G}
\UnaryInfC{$\kappa_2 \vdash \mathrm{F}(S_{\kappa_3}) \subseteq \mathrm{G}(L_{\kappa_1})$}
\RightLabel{S}
\UnaryInfC{$\kappa_2 \vdash x_1 \in \mathrm{F}(S_{\kappa_3}) \subseteq \mathrm{G}(L_{\kappa_1})$}
\RightLabel{μ_l}
\UnaryInfC{$\vdash x_1 \in S_{\kappa_2} \subseteq \mathrm{G}(L_{\kappa_1})$}
\RightLabel{I_1}
\UnaryInfC{$\forall\alpha_0,\alpha_1(\vdash S_{\alpha_1} \subseteq \mathrm{G}(L_{\alpha_0}))$}
\RightLabel{G}
\UnaryInfC{$\infty \vdash \mathrm{S} \subseteq \mathrm{G}(L_{\kappa_0})$}
\RightLabel{S}
\UnaryInfC{$\infty \vdash x_0 \in \mathrm{S} \subseteq \mathrm{G}(L_{\kappa_0})$}
\RightLabel{ν_r}
\UnaryInfC{$\vdash x_0 \in \mathrm{S} \subseteq \mathrm{L}$}
\end{prooftree}
$$

where

$$
\begin{array}{rlrl}
\mathrm{F}(X) &= \nu Y.[\mathrm{A}\ \mathrm{of}\ X \mid \mathrm{B}\ \mathrm{of}\ Y] & \kappa_3 &= \varepsilon_{\alpha<\kappa_2}(x_1 \in \mathrm{F}(S_\alpha)) \\
S_\alpha &= \mu_\alpha X.\mathrm{F}(X) & \kappa_4 &= \overline{\varepsilon}_{\alpha_1,\alpha_2<O,O}(\mathrm{F}(S_{\alpha_2}) \nsubseteq \mathrm{G}(L_{\alpha_1}))_1 \\
\mathrm{G}(Y) &= \mu X.[\mathrm{A}\ \mathrm{of}\ X \mid \mathrm{B}\ \mathrm{of}\ Y] & \kappa_5 &= \overline{\varepsilon}_{\alpha_1,\alpha_2<O,O}(\mathrm{F}(S_{\alpha_2}) \nsubseteq \mathrm{G}(L_{\alpha_1}))_2 \\
L_\beta &= \nu_\beta Y.\mathrm{G}(Y) & \kappa_6 &= \varepsilon_{\alpha<\kappa_4}(x_2.B \notin \mathrm{G}(L_\alpha)) \\
\kappa_0 &= \varepsilon_{\alpha<\infty}(x_0 \notin \mathrm{G}(L_\alpha)) & x_0 &= \varepsilon_{x\in\mathrm{S}}(x \notin \mathrm{L}) \\
\kappa_1 &= \overline{\varepsilon}_{\alpha_1,\alpha_2<O,O}(S_{\alpha_2} \nsubseteq \mathrm{G}(L_{\alpha_1}))_1 & x_1 &= \varepsilon_{x\in S_{\kappa_2}}(x \notin \mathrm{G}(L_{\kappa_1})) \\
\kappa_2 &= \overline{\varepsilon}_{\alpha_1,\alpha_2<O,O}(S_{\alpha_2} \nsubseteq \mathrm{G}(L_{\alpha_1}))_2 & x_2 &= \varepsilon_{x\in\mathrm{F}(S_{\kappa_5})}(x \notin \mathrm{G}(L_{\kappa_4}))
\end{array}
$$

Fig. 10. Example of circular proof involving inductive and coinductive types.

$N_T = (\mathrm{T} \to \mathrm{T}) \to \mathrm{T} \to \mathrm{T}$ with $\mathrm{T} = \mu X.((X \to \mathbb{B}) \to \mathbb{B})$ must be used for the natural numbers. Note that $\mathbb{B}$ corresponds to (some encoding of) the booleans, with constants $T : \mathbb{B}$ and $F : \mathbb{B}$.

$$P_C : \mathbb{N}_C \to \mathbb{N}_C = \lambda n.n\ (\lambda p\, x{:}\mathbb{N}_C\, y{:}\mathbb{N}_C.p\ (S_C\ x)\ x)\ (\lambda x\, y.y)\ 0_C\ 0_C$$

$$R_C : \forall P.(P \to \mathbb{N}_C \to P) \to P \to \mathbb{N}_C \to P = \lambda f\, a\, n.n\ (\lambda x\, p{:}\mathbb{N}_C.f\ (x\ (S_C\ p))\ p)\ (\lambda p.a)\ 0_C$$

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

Scott numerals were initially introduced because they admit a constant time predecessor function P_S, whereas Church numerals do not.

$$P_S : \mathbb{N}_S \to \mathbb{N}_S = \lambda n.n\ (\lambda p.p)\ 0_S$$

Usually, programming using Scott numerals requires a recursor similar to that of Gödel's System T. Such a recursor can be easily programmed using general recursion, but this would require introducing typable terms that are not strongly normalising. In our type system, we can typecheck a strongly normalisable recursor R_S due to Michel Parigot [44]. It is displayed below together with terms and types involved in its definition.

$$\mathrm{U}(P) = \forall Y.Y \to \mathbb{N}_S \to P$$

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

It is easy to check that R_S is indeed a recursor for Scott numerals. This term is similar to a λ-calculus fixed point combinator, but only allows a limited number of unfoldings. As the recursor is typable, Theorem 6.25 page 38 implies that it is strongly normalising. The crucial point for typing the recursor is the subtyping relation $\mathbb{N}_S \subseteq \mathbb{N}'$. It is however not clear what are the terms of type $\mathbb{N}'$ that are not in $\mathbb{N}_S$. In our implementation, $n : \mathbb{N}'$ is the only required type annotation for R_S to be accepted (the types of ζ and δ do not need to be provided).

The recursor for Scott numerals can be adapted to other algebraic data types like lists or trees. More surprisingly, it is also possible to give similar recursors for some coinductive data types. For instance, it is possible to encode streams using the following definitions.[14]

$$
\begin{aligned}
\mathbb{S}(A) &= \nu K.(\exists S.\{\mathrm{head} : S \to A;\ \mathrm{tail} : S \to K;\ \mathrm{state} : S\}) \\
\mathrm{head} &: \forall A.\mathbb{S}(A) \to A = \lambda s.s.head\ s.state \\
\mathrm{tail} &: \forall A.\mathbb{S}(A) \to \mathbb{S}(A) = \lambda s.s.tail\ s.state \\
\mathrm{cons} &: \forall A.A \to \mathbb{S}(A) \to \mathbb{S}(A) = \lambda a\, l.\{\mathrm{head} = \lambda u.a;\ \mathrm{tail} = \lambda u.l;\ \mathrm{state} = \{\}\}
\end{aligned}
$$

Convention 5.1. *In the examples, we may use arbirary names (such as* head *or* tail*) for record fields and constructors. In particular, we do not limit ourselves to the notation C_k and l_k used in the formal definiton of the language. However, it is very easy to see that these representations are equivalent. Moreover, we sometimes use the standard cartesian product notation $A_1 \times \cdots \times A_n$ for the type $\{(l_i : A_i)_{i\in[1..n]}\}$, the tuple notation $(t_1, \ldots, t_n)$ for the term $\{(l_i = t_i)_{i\in[1..n]}\}$, and the notation $t.i$ for the projection $t.l_i$ when the type of t is a cartesian product $A_1 \times \cdots \times A_n$ and $1 \le i \le n$.*

In the definition of $\mathbb{S}(A)$, the existentially quantified type can be seen as the representation of an internal state.[15] It must be provided to compute the head or the tail of the stream, and thus introduces some laziness into the data type. Our stongly normalising coiterator $I_{\mathbb{S}}$ for streams is given below, together with terms and types involved in its definition.

$$
\begin{aligned}
\mathrm{T}(A,P) &= \forall Y.(P \times Y) \to \{\mathrm{head} : (P \times Y) \to A;\ \mathrm{tail} : Y;\ \mathrm{state} : P \times Y\} \\
\mathbb{S}'(A,P) &= \{\mathrm{head} : (P \times \mathrm{T}(A,P)) \to A;\ \mathrm{tail} : \mathrm{T}(A,P);\ \mathrm{state} : P \times \mathrm{T}(A,P)\} \\
\zeta &: \forall A\, P.(P \to A) \to \forall X.(P \times X) \to A = \lambda f\, s.f\ s.1 \\
\delta &: \forall A\, P.(P \to A) \to (P \to P) \to \mathrm{T}(A,P) \\
&= \lambda f\, n\, s.\{\mathrm{head} = \zeta\ f;\ \mathrm{tail} = s.2;\ \mathrm{state} = (n\ s.1, s.2)\} \\
I_{\mathbb{S}} &: \forall A\, P.P \to (P \to A) \to (P \to P) \to \mathbb{S}(A) \\
&= \lambda s\, f\, n.\ \ \mathrm{let}\ A, P\ \mathrm{such\ that}\ f : P \to A\ \mathrm{in} \\
&\qquad\qquad \{\mathrm{head} = \zeta\ f;\ \mathrm{tail} = \delta\ f\ n;\ \mathrm{state} = (s, \delta\ f\ n)\} : \mathbb{S}'(A,P)
\end{aligned}
$$

Note that in the definition of $I_{\mathbb{S}}$, we deliberately use the same names as in the definitions of R_S to highlight their similarities. The minimum type annotation for our implementation to type-check $I_{\mathbb{S}}$ involves the subtyping relation $\mathbb{S}'(\mathrm{A},\mathrm{P}) \subseteq \mathbb{S}(\mathrm{A})$. The let-binding syntax in $I_{\mathbb{S}}$ is used to name

[14] The used notation conventions for records and product types are explained in Convention 5.1.

[15] The order in which the greatest fixed point and the existential type are given is essential for typing cons.

universally quantified types (see Section 9 page 50). It is only used in the implementation, and is not part of the theoretical type system. As for Scott numerals, the types of ζ and δ are not required, and a question arises about the inhabitants of the type $\exists P.\mathbb{S}'(\mathrm{A}, P)$.

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 $\mu N.[\mathrm{Z} \mid \mathrm{S}\ \mathrm{of}\ N]$, and we conjecture that this is impossible. However, if we encode the sum type using a record type, a recursor can be given. The type of the unary natural numbers then becomes $\mathbb{N}'_S = \mu X.(\forall Y.\{\mathrm{z} : Y;\ \mathrm{s} : X \to Y\} \to Y)$, which is similar to the type of Scott numerals $\mathbb{N}_S$.

6 REALIZABILITY SEMANTICS

In this section, we build a realizability model and we prove that it is adequate with respect to our type system. In particular, we interpret every term t with a *pure term* $[\![t]\!]$, and every type A with a set of *strongly normalising pure terms* $[\![A]\!]$. Consequently, we will have $[\![t]\!] \in [\![A]\!]$ whenever $\vdash t : A$ is derivable using a well-founded circular proof.

Definition 6.1. *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 $[\![\Lambda]\!] \subset \Lambda$ the set of all pure terms. A pure term $t \in [\![\Lambda]\!]$ is said to be strongly normalising if there is no infinite sequence of reduction starting from t, using the relation $(\succ)$ of Definition 4.4 page 24. We denote $\mathcal{N} \subset [\![\Lambda]\!]$ the set of strongly normalising pure terms.*

Definition 6.2. *The set $\mathcal{H}$ of head contexts (which correspond to terms with a hole in head position) is generated by the following BNF grammar.*

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

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 λ-variables). We denote $(\succ_H)$ the head reduction *relation defined as the closure under head context of the rules of Figure 5 page 24. We say that a term is in* head normal form *if it cannot be reduced using $(\succ_H)$.*

Definition 6.3. *We say that a set of pure terms $\Phi \subseteq [\![\Lambda]\!]$ is* saturated *whenever it satisfies the following five conditions.*

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

Note that the conditions (1) and (5) in the above definition are unusual, especially the former which requires saturated sets to be closed under head reduction. However, they are necessary for the $(+)$ subtyping rule to be shown adequate in Theorem 6.23 page 36.

Lemma 6.4. *Let $\Phi \subseteq [\![\Lambda]\!]$ be a saturated set of pure terms, and let $t \in \mathcal{N}$ be a strongly normalising term. If we have $t \succ_H u$ for some term $u \in \Phi$, then $t \in \Phi$.*

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

Lemma 6.5. *The set $\mathcal{N}$ is saturated.*

Proof. We need to show that $\mathcal{N}$ satisfies the five conditions of Definition 6.3.

(1) Immediate by definition of $\mathcal{N}$ since $(\succ_H) \subseteq (\succ)$.
(2) 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 $H[(\lambda x.t)\ u] \succ H[t[x := u]] \succ^* H'[t'[x := u']]$.
(3) Let us take $H[t\,u] \in \mathcal{N}$ and suppose, by contradiction, that $H[[C_k\ u \mid C_k \to t]] \notin \mathcal{N}$. There cannot be an infinite reduction of H, u or t. Thus, an infinite reduction of $H[[C_k\ u \mid C_k \to t]]$ must starts with $H[[C_k\ u \mid C_k \to t]] \succ^* H'[[C_k\ u' \mid C_k \to t']] \succ H'[t'\,u']$, where $H \succ^* H'$, $t \succ^* t'$ and $u \succ^* u'$. This can be transformed into $H[[C_k\ u \mid C_k \to t]] \succ H[t\,u] \succ^* H'[t'\,u']$, which contradicts $H[t\,u] \in \mathcal{N}$.
(4) Let us take $H[t] \in \mathcal{N}$, $t_i \in \mathcal{N}$ for all $i \in I$, and assume $H[\{l_k = t; (l_i = t_i)_{i\in I}\}.l_k] \notin \mathcal{N}$ by contradiction. There cannot be an infinite reduction of H, t nor of any of the t_i. As a consequence, an infinite reduction of the term $H[\{l_k = t; (l_i = t_i)_{i\in I}\}.l_k]$ must start with $H[\{l_k = t; (l_i = t_i)_{i\in I}\}.l_k] \succ^* H[\{l_k = t'; (l_i = t_i')_{i\in I}\}.l_k]$, 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 $H[\{l_k = t; (l_i = t_i)_{i\in I}\}.l_k] \succ^* H[t] \succ H'[t']$.
(5) Finally, let us take $H[[t \mid (C_i \to t_i)_{i\in I}]] \in \mathcal{N}$, a finite set $J \subset \mathbb{N}$ 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 for H, t nor any of the t_j. Thus, an infinite reduction must start 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 6.6. *The set of* neutral terms $\mathcal{N}_0 \subset [\![\Lambda]\!]$ *is the smallest set such that:*

(1) for every λ-variable $x \in \mathcal{V}_\Lambda$ 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 set of indices I, terms $(t_i)_{i\in I} \in \mathcal{N}^I$ and $t \in \mathcal{N}_0$ we have $[t \mid (C_i \to t_i)_{i\in I}] \in \mathcal{N}_0$.

As the set of neutral terms $\mathcal{N}_0$ is not saturated, we will consider the least saturated in which it is included. This set will then correspond to the smallest possible interpretation for a type. It will most notably be used to interpret the empty type.

Definition 6.7. *Given $\Phi \subseteq [\![\Lambda]\!]$, we denote $\overline{\Phi} \subseteq [\![\Lambda]\!]$ the smallest saturated set containing Φ.*

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

Proof. We obviously have $\mathcal{N}_0 \subseteq \overline{\mathcal{N}_0}$ and $\mathcal{N}_0 \subseteq \mathcal{N}$. The saturation operation being obviously covariant, the latter yields $\overline{\mathcal{N}_0} \subseteq \overline{\mathcal{N}} = \mathcal{N}$. □

Definition 6.9. *Given two sets $\Phi_1, \Phi_2 \subseteq [\![\Lambda]\!]$ we define $(\Phi_1 \Rightarrow \Phi_2) \subseteq [\![\Lambda]\!]$ as follows.*

$$(\Phi_1 \Rightarrow \Phi_2) = \{t \in [\![\Lambda]\!] \mid \forall u \in \Phi_1,\ t\ u \in \Phi_2\}$$

Lemma 6.10. *Let $\Phi_1, \Phi_2, \Psi_1, \Psi_2 \subseteq [\![\Lambda]\!]$ 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 6.11. *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 6.8 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 6.10. 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 of 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 Definition 2.6 page 14. Of course, the interpretation of syntactic ordinals will involve the interpretation of terms and formulas through abstract judgements. The interpretation of our three syntactic entities is thus defined mutually inductively, as was their syntax.

Definition 6.12. *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 for syntactic ordinals in Definition 2.5 page 14, and it will allow us to work with closed syntactic elements only. Most notably, we will use substitutions with elements of the semantics instead of relying on a semantic map for interpreting free variables.

Definition 6.13. *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, but the definition of $\mathcal{F}^*$ impacts the definition of Λ^* since terms and formulas are defined mutually inductively.*[16] *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. However, λ-variables are allowed as candidates may contain open terms.*

Definition 6.14. *A closed parametric term $t \in \Lambda^*$ (resp. closed parametric formula $A \in \mathcal{F}^*$) is interpreted by a pure term $[\![t]\!] \in [\![\Lambda]\!]$ (resp. a candidate $[\![A]\!] \in [\![\mathcal{F}]\!]$) according to Figure 11 page 32. Syntactic ordinals are interpreted through Definition 2.6 page 14, interpreting abstract judgments in the obvious way according to Definition 3.4 page 17, and taking the ordinal Ω of Definition 2.4 page 13 to be the cardinal $2^{2^{\aleph_0}}$ (seen as an ordinal through the Von Neumann cardinal assignment).*

Note that the axiom of choice is required to interpret choice operators. Moreover, in the interpretation of terms 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 the syntactic restriction given in Definition 4.1 page 22. Indeed, invalid terms like $\lambda y.\varepsilon_{x\in A}(x\,y \notin B)$ cannot be given an interpretation as pure terms. Note also 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 $\mathcal{N}$). If it was omitted, we would have $[\![\mu_0 F]\!] = \emptyset \notin [\![\mathcal{F}]\!]$ (resp. $[\![\nu_0 F]\!] = \Lambda \notin [\![\mathcal{F}]\!]$), which is not a valid type interpretation.

Lemma 6.15. *The semantic interpretation of terms, formulas and syntactic ordinals commutes with the substitution of the three kinds of variables. For instance, we have $[\![t[x := u]]\!] = [\![t[x := [\![u]\!]]]\!]$, $[\![A[X := B]]\!] = [\![A[X := [\![B]\!]]]\!]$ and $[\![A[\alpha := \kappa]]\!] = [\![A[\alpha := [\![\kappa]\!]]]\!]$.*

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

[16] The set of parametric syntactic ordinals $\mathcal{O}^*$ of Definition 2.5 page 14 is also impacted through abstract judgments.

$$\begin{aligned}
[\![x]\!] &= x \\
[\![t\,u]\!] &= [\![t]\!]\;[\![u]\!] \\
[\![\lambda x.t]\!] &= \lambda x.[\![t]\!] \\
[\![C\,u]\!] &= C\;[\![u]\!] \\
[\![[u \mid (C_i \to t_i)_{i\in I}]]\!] &= [[\![u]\!] \mid (C_i \to [\![t_i]\!])_{i\in I}] \\
[\![\{(l_i = t_i)_{i\in I}\}]\!] &= \{(l_i = [\![t_i]\!])_{i\in I}\} \\
[\![\varepsilon_{x\in A}(t \notin B)]\!] &= \begin{cases} u \in [\![A]\!] \text{ such that } [\![t[x := u]]\!] \notin [\![B]\!] \text{ if it exists,} \\ \text{any } t \in \mathcal{N}_0 \text{ otherwise.} \end{cases}
\end{aligned}$$

$$\begin{aligned}
[\![\Phi]\!] &= \Phi \;\text{ if } \Phi \in [\![\mathcal{F}]\!] \\
[\![A \to B]\!] &= [\![A]\!] \Rightarrow [\![B]\!] \\
[\![\{(l_i : A_i)_{i\in I}; \dots\}]\!] &= \{t \in \mathcal{N} \mid \forall i \in I, t.l_i \in [\![A_i]\!]\} \\
[\![\{(l_i : A_i)_{i\in I\neq\emptyset}\}]\!] &= \{t \in \mathcal{N} \mid \forall i \in I, t.l_i \in [\![A_i]\!] \text{ and } \forall i \notin I, t.l_i \succ_H^* \Omega\} \\
[\![\{\}]\!] &= \overline{\mathcal{N}_0 \cup \{\{\}\}} \\
[\![[(C_i : A_i)_{i\in I}]]\!] &= \{t \in \mathcal{N} \mid \forall \Phi \in [\![\mathcal{F}]\!], \forall (t_i \in ([\![A_i]\!] \Rightarrow \Phi))_{i\in I}, [t \mid (C_i \to t_i)_{i\in I}] \in \Phi\} \\
[\![\forall X.A]\!] &= \cap_{\Phi\in[\![\mathcal{F}]\!]} [\![A[X := \Phi]]\!] \\
[\![\exists X.A]\!] &= \cup_{\Phi\in[\![\mathcal{F}]\!]} [\![A[X := \Phi]]\!] \\
[\![\mu_\kappa X.A]\!] &= \left(\cup_{o<[\![\kappa]\!]} [\![A[X := \mu_o X A]]\!]\right) \cup \overline{\mathcal{N}}_0 \\
[\![\nu_\kappa X.A]\!] &= \left(\cap_{o<[\![\kappa]\!]} [\![A[X := \nu_o X A]]\!]\right) \cap \mathcal{N} \\
[\![\varepsilon_X(t \in A)]\!] &= \begin{cases} \Phi \in [\![\mathcal{F}]\!] \text{ such that } [\![t]\!] \in [\![A[X := \Phi]]\!] \text{ if it exists,} \\ \mathcal{N} \text{ otherwise.} \end{cases} \\
[\![\varepsilon_X(t \notin A)]\!] &= \begin{cases} \Phi \in [\![\mathcal{F}]\!] \text{ such that } [\![t]\!] \notin [\![A[X := \Phi]]\!] \text{ if it exists,} \\ \mathcal{N} \text{ otherwise.} \end{cases}
\end{aligned}$$

Fig. 11. Semantical interpretation of closed parametric terms and types.

Lemma 6.16. *For all candidates* $\Phi, \Psi \in [\![\mathcal{F}]\!]$, *we have* $(\Phi \Rightarrow \Psi) \in [\![\mathcal{F}]\!]$.

Proof. Since Φ and Ψ are candidates, we have $\overline{\mathcal{N}_0} \subseteq \Phi \subseteq \mathcal{N}$ and $\overline{\mathcal{N}_0} \subseteq \Psi \subseteq \mathcal{N}$. As a consequence, we can use Lemma 6.10 page 31 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 $\overline{\mathcal{N}_0} \subseteq (\Phi \Rightarrow \Psi) \subseteq \mathcal{N}$ thanks to Lemma 6.11 page 31. It remains to show that $\Phi \Rightarrow \Psi$ is saturated by proving the five conditions of Definition 6.3 page 30.

(1) 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 $t\ u \succ_H t'\ u$, it is enough to show $t\ u \in \Psi$ according to the saturation condition (1) on Ψ. We can thus conclude by definition of $t \in (\Phi \Rightarrow \Psi)$.
(2) 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 (2) on Ψ with the context $H\ v$.

(3) 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 we can hence use the saturation condition (3) on Ψ with $H\ v$.

(4) 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 $H[\{l_k = t; (l_i = t_i)_{i\in I}\}.l_k] \in (\Phi \Rightarrow \Psi)$ so we take $v \in \Phi \subseteq \mathcal{N}$ and we prove $H[\{l_k = t; (l_i = t_i)_{i\in I}\}.l_k]\ v \in \Psi$. As $H[t] \in (\Phi \Rightarrow \Psi)$ we know that $H[t]\ v \in \Psi$ and we can thus conclude with the saturation condition (4) on Ψ with $H\ v$.

(5) 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 that $H[t \mid (C_i \to t_i)_{i\in J}] \in (\Phi \Rightarrow \Psi)$, 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 $H[t \mid (C_i \to t_i)_{i\in I}]\ v \in \Psi$. We can hence conclude using the saturation condition (5) on Ψ with the context $H\ v$. □

Lemma 6.17. *If we have $\Phi_i \in [\![\mathcal{F}]\!]$ for all $i \in I$, then $[\![[(C_i : \Phi_i)_{i\in I}]]\!] \in [\![\mathcal{F}]\!]$.*

Proof. By definition, it is immediate that $[\![[(C_i : \Phi_i)_{i\in I}]]\!] \subseteq \mathcal{N}$. Let us take $t \in \overline{\mathcal{N}}_0$ and show that $t \in [\![[(C_i : \Phi_i)_{i\in I}]]\!]$. 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 since $[t \mid (C_i \to t_i)_{i\in I}] \in \overline{\mathcal{N}}_0$ and $\overline{\mathcal{N}}_0 \subseteq \Phi$. It remains to show that $[\![[(C_i : \Phi_i)_{i\in I}]]\!]$ satisfies the five conditions of Definition 6.3 page 30.

(1) Let us take $t \in [\![[(C_i : \Phi_i)_{i\in I}]]\!]$ such that $t \succ_H t'$, and show that $t' \in [\![[(C_i : \Phi_i)_{i\in I}]]\!]$. We take $\Phi \in [\![\mathcal{F}]\!]$ and $t_i \in (\Phi_i \Rightarrow \Phi)$ for all $i \in I$, and we show $[t' \mid (C_i \to t_i)_{i\in I}] \in \Phi$. Since $t \in [\![[(C_i : \Phi_i)_{i\in I}]]\!]$ we know that $[t \mid (C_i \to t_i)_{i\in I}] \in \Phi$. We can thus conclude using the saturation condition (1) on Φ since $[t \mid (C_i \to t_i)_{i\in I}] \succ_H [t' \mid (C_i \to t_i)_{i\in I}]$.

(2) Let us suppose that we have $H[t[x := u]] \in [\![[(C_i : \Phi_i)_{i\in I}]]\!]$ and $u \in \mathcal{N}$, and show that $H[(\lambda x.t)\ u] \in [\![[(C_i : \Phi_i)_{i\in I}]]\!]$. 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 $H[t[x := u]] \in [\![[(C_i : \Phi_i)_{i\in I}]]\!]$ 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 (2) on Φ with the context $[H \mid (C_i \to t_i)_{i\in I}]$.

(3) We suppose $H[t\ u] \in [\![[(C_i : \Phi_i)_{i\in I}]]\!]$ and show $H[[D\ u \mid D \to t]] \in [\![[(C_i : \Phi_i)_{i\in I}]]\!]$. We take $\Phi \in [\![\mathcal{F}]\!]$ and $t_i \in (\Phi_i \Rightarrow \Phi)$ for all $i \in I$, and show $[H[[D\ u \mid D \to t]] \mid (C_i \to t_i)_{i\in I}] \in \Phi$. As $H[t\ u] \in [\![[(C_i : \Phi_i)_{i\in I}]]\!]$ we know that $[H[t\ u] \mid (C_i \to t_i)_{i\in I}] \in \Phi$. We can thus conclude using the saturation condition (3) of Φ with the context $[H \mid (C_i \to t_i)_{i\in I}]$.

(4) Let us now suppose that $H[t] \in [\![[(C_i : \Phi_i)_{i\in I}]]\!]$ and that $t_i \in \mathcal{N}$ for all $i \in I$. We need to show that $H[\{l_k = t; (l_i = t_i)_{i\in I}\}.l_k] \in [\![[(C_i : \Phi_i)_{i\in I}]]\!]$ so we take $\Phi \in [\![\mathcal{F}]\!]$ and $t_i \in (\Phi_i \Rightarrow \Phi)$ for all $i \in I$, and show $[H[\{l_k = t; (l_i = t_i)_{i\in I}\}.l_k] \mid (C_i \to t_i)_{i\in I}] \in \Phi$. As $H[t] \in [\![[(C_i : \Phi_i)_{i\in I}]]\!]$ we know that $[H[t] \mid (C_i \to t_i)_{i\in I}] \in \Phi$. We can thus conclude with the saturation condition (4) on Φ with the context $[H \mid (C_i \to t_i)_{i\in I}]$.

(5) We now suppose $H[t \mid (C_i \to t_i)_{i\in I}] \in [\![[(C_i : \Phi_i)_{i\in I}]]\!]$ and $I \subset J$ with $t_j \in \mathcal{N}$ for all $j \in J \setminus I$. We need to show $H[t \mid (C_i \to t_i)_{i\in J}] \in [\![[(C_i : \Phi_i)_{i\in I}]]\!]$ 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 $H[t \mid (C_i \to t_i)_{i\in I}] \in [\![[(C_i : \Phi_i)_{i\in I}]]\!]$ we have $[H[t \mid (C_i \to t_i)_{i\in I}] \mid (C_i \to t_i)_{i\in I}] \in \Phi$. We thus conclude by saturation condition (5) on Φ with the context $[H \mid (C_i \to t_i)_{i\in I}]$. □

Lemma 6.18. *If we have $\Phi_i \in [\![\mathcal{F}]\!]$ for all $i \in I$, then $[\![\{(l_i : \Phi_i)_{i\in I}; \dots\}]\!] \in [\![\mathcal{F}]\!]$.*

Proof. Similar to the proof of Lemma 6.17. □

Lemma 6.19. *If we have $\Phi_i \in [\![\mathcal{F}]\!]$ for all $i \in I$, then $[\![\{(l_i : \Phi_i)_{i\in I}\}]\!] \in [\![\mathcal{F}]\!]$.*

Proof. Immediate if $I = \emptyset$, and similar to the proof of Lemma 6.17 otherwise. □

Theorem 6.20. *For every closed parametric term $t \in \Lambda^*$ (resp. ordinal $\kappa \in O^*$, resp. type $A \in \mathcal{F}^*$) we have $[\![t]\!] \in [\![\Lambda]\!]$ (resp. $[\![\kappa]\!] \in [\![O]\!]$, 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 follows from Definition 2.6 page 14, using the induction hypothesis to interpret the predicates in choice operators according to Definition 3.4 page 17. 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 $A \Rightarrow B$, $[(C_i : A_i)_{i \in I}]$, $\{(l_i : A_i)_{i \in I}; \ldots\}$ or $\{(l_i : A_i)_{i \in I}\}$, we respectively use Lemma 6.16 page 33, Lemma 6.17, Lemma 6.18 and Lemma 6.19, together with the induction hypotheses and Lemma 6.15 page 32. The remaining four cases are treated below.

- 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 since candidates are stable by intersection.
- Similarly, for types of the form $\exists X.A$, the induction hypothesis gives $[\![A[X := \Phi]]\!] \in [\![\mathcal{F}]\!]$ for all $\Phi \in [\![\mathcal{F}]\!]$. We can then conclude since candidates are stable by union.
- For types of the form $\mu_\kappa X.A$, we need to show $[\![\mu_\kappa X.A]\!] \in [\![\mathcal{F}]\!]$. Since $[\![\mu_\kappa X.A]\!] = [\![\mu_{[\![\kappa]\!]} X.A]\!]$ by Lemma 6.15 page 32, we will show $[\![\mu_o X.A]\!] \in [\![\mathcal{F}]\!]$ for all $o \leq [\![\kappa]\!]$ by ordinal induction. If $o = 0$ then we have $[\![\mu_0 X.A]\!] = \overline{\mathcal{N}}_0$ by definition, and the proof is immediate. Otherwise, we have $[\![\mu_o X.A]\!] = \cup_{o' < o} [\![A[X := \mu_{o'} X.A]]\!]$ and the local induction hypothesis gives $[\![\mu_{o'} X.A]\!] \in [\![\mathcal{F}]\!]$ for all $o' < o$. We then obtain $[\![A[X := \mu_{o'} X.A]]\!] = [\![A[X := [\![\mu_{o'} X.A]\!]]]\!]$ for all $o' < o$ using Lemma 6.15 page 32. The global induction hypothesis then given $[\![A[X := [\![\mu_{o'} X.A]\!]]]\!] \in [\![\mathcal{F}]\!]$ for all $o' < o$, and we can then conclude using the fact that candidates are stable by union.
- For types of the form $\nu_\kappa X.A$, we proceed in a similar way as in the previous case, using the fact that candidates are stable by intersection. 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 should either reduce to a neutral term (an element of $\overline{\mathcal{N}}_0$) or to a constructor. Although the semantics of our sum types involve arrows, we still obtain this result thanks to parametricity. Indeed, the common codomain of the arrows is quantified over universally in the interpretation of sum types.

Lemma 6.21. *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 6.22. *If we have $[\![A_i]\!] \in [\![\mathcal{F}]\!]$ for all $i \in I$, then $t \in [\![[(C_i : A_i)_{i \in I}]]\!]$ 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 $t \in [\![[(C_i : A_i)_{i \in I}]]\!]$. By definition, we immediately have $t \in \mathcal{N}$, so according to Lemma 6.21 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 $[\![[(C_i : A_i)_{i \in I}]]\!]$ 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 $\lambda x.x \in (\llbracket A_k \rrbracket \Rightarrow \llbracket A_k \rrbracket)$ and $\lambda x.y \in (\llbracket A_i \rrbracket \Rightarrow \mathcal{N}_0) \subseteq (\llbracket A_i \rrbracket \Rightarrow \llbracket A_k \rrbracket)$ for all $i \in I \setminus \{k\}$. Therefore, we can use the definition of $\llbracket [(C_i : A_i)_{i \in I}] \rrbracket$ to obtain $u_k \in \llbracket A_k \rrbracket$. We can then conclude that $u \in \llbracket A_k \rrbracket$ as $\llbracket A_k \rrbracket$ is saturated and $u_k \succ_H (\lambda x.x)u \succ_H u$.

($\Leftarrow$) 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 $u \in \llbracket A_k \rrbracket$. We need to show $t \in \llbracket [(C_i : A_i)_{i \in I}] \rrbracket$, so we take a set $\Phi \in \llbracket \mathcal{F} \rrbracket$, terms $t_i \in (\llbracket A_i \rrbracket \Rightarrow \Phi)$ 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 6.4 page 30. 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 $u \in \llbracket A_k \rrbracket$, then we need to show $t_k\ u \in \llbracket A_k \rrbracket$, which follows from $t_k \in (\llbracket A_k \rrbracket \Rightarrow \Phi)$. □

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 the adequacy lemma.

Theorem 6.23. *Let γ be an ordinal context such that $\llbracket \tau \rrbracket > 0$ for all $\tau \in \gamma$.*

(1) If $\gamma \vdash t \in A \subseteq B$ is derivable by a well-founded proof and $\llbracket t \rrbracket \in \llbracket A \rrbracket$ then $\llbracket t \rrbracket \in \llbracket B \rrbracket$.
(2) If $\vdash t : A$ is derivable by a well-founded proof then $\llbracket t \rrbracket \in \llbracket A \rrbracket$.

Proof. According to Theorem 3.16 page 21 we only have to prove that our typing and subtyping rules are correct. We thus consider all the rules of Figures 6 and 7 page 24.

($\to_i$) We need to show $\llbracket \lambda x.t \rrbracket \in \llbracket C \rrbracket$. However, according to the second induction hypothesis, it is enough to show $\llbracket \lambda x.t \rrbracket \in \llbracket A \to B \rrbracket = (\llbracket A \rrbracket \Rightarrow \llbracket B \rrbracket)$. Using the second induction hypothesis we have $\llbracket t[x := \varepsilon_{x \in A}(t \notin B)] \rrbracket \in \llbracket B \rrbracket$. By definition of the choice operator, this means that we have $\llbracket t[x := u] \rrbracket = \llbracket t \rrbracket[x := u] \in \llbracket B \rrbracket$ for all $u \in \llbracket A \rrbracket$. By Theorem 6.20 we know that $\llbracket B \rrbracket$ is saturated and that $\llbracket A \rrbracket \subseteq \mathcal{N}$. We then use the saturation condition (2) on $\llbracket B \rrbracket$ to obtain $\llbracket \lambda x.t \rrbracket\ u \in \llbracket B \rrbracket$ for all $u \in \llbracket A \rrbracket$.

($\to_e$) We need to show $\llbracket t \rrbracket\ \llbracket u \rrbracket \in \llbracket B \rrbracket$. By induction hypothesis we know that $\llbracket t \rrbracket \in \llbracket A \to B \rrbracket$ and that $\llbracket u \rrbracket \in \llbracket A \rrbracket$, so we can conclude by definition of $\llbracket A \to B \rrbracket = (\llbracket A \rrbracket \Rightarrow \llbracket B \rrbracket)$.

(ε) We need to show $\llbracket \varepsilon_{x \in A}(t \notin B) \rrbracket \in \llbracket C \rrbracket$. However, according to the induction hypothesis, it is enough to show $\llbracket \varepsilon_{x \in A}(t \notin B) \rrbracket \in \llbracket A \rrbracket$. This follows immediately from the definition of $\llbracket \varepsilon_{x \in A}(t \notin B) \rrbracket$. In particular, $\overline{\mathcal{N}}_0 \subseteq \llbracket A \rrbracket$ according to Theorem 6.20 page 35.

($\times_i$) We need to show that $\llbracket \{(l_i = t_i)_{i \in I}\} \rrbracket \in \llbracket B \rrbracket$. According to the first induction hypothesis, it is enough to show $\llbracket \{(l_i = t_i)_{i \in I}\} \rrbracket \in \llbracket \{(l_i : A_i)_{i \in I}\} \rrbracket$. By definition, we need to take $k \in I$ and show $\{(l_i = \llbracket t_i \rrbracket)_{i \in I}\}.l_k \in \llbracket A_k \rrbracket$. By induction hypothesis we know that $\llbracket t_k \rrbracket \in \llbracket A_k \rrbracket$, hence we can use the saturation condition (4) on $\llbracket A_k \rrbracket$ (it is saturated according to Theorem 6.20 page 35). Note that if $k \notin I$ then we immediately have $\{(l_i = \llbracket t_i \rrbracket)_{i \in I}\}.l_k \succ_H^* \Omega$.

($\times_e$) We need to show $\llbracket t.l_k \rrbracket = \llbracket t \rrbracket.l_k \in \llbracket A \rrbracket$. Since we have $\llbracket t \rrbracket \in \llbracket \{l_k : A; \dots\} \rrbracket$ according to our induction hypothesis, we can conclude by definition of $\llbracket \{l_k : A; \dots\} \rrbracket$.

($+_i$) We need to show $\llbracket C_k\, t \rrbracket \in \llbracket B \rrbracket$. According to the first induction hypothesis, it is enough to show $\llbracket C_k\, t \rrbracket = C_k\, \llbracket t \rrbracket \in \llbracket [C_k : A] \rrbracket$. By definition, we need to take $\Phi \in \llbracket \mathcal{F} \rrbracket$, $t_k : (\llbracket A \rrbracket \Rightarrow \Phi)$ and show $[C_k\, \llbracket t \rrbracket \mid C_k \to t_k] \in \Phi$. Using the saturation condition (3) on Φ, it is enough to show $t_k\ \llbracket t \rrbracket \in \Phi$. This follows by definition of $(\llbracket A \rrbracket \Rightarrow \Phi)$ since $\llbracket t \rrbracket \in \llbracket A \rrbracket$ according to the second induction hypothesis.

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

($\rightarrow$) Let us suppose that $[\![t]\!] \in [\![A_1 \rightarrow B_1]\!]$, and assume that $[\![t]\!] \notin [\![A_2 \rightarrow B_2]\!]$ by contradiction. By definition of $[\![A_2 \rightarrow B_2]\!] = ([\![A_2]\!] \Rightarrow [\![B_2]\!])$ 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 $t \in [\![A_1 \rightarrow B_1]\!]$. Using the second induction hypothesis this gives $[\![t]\!]\, v \in [\![B_2]\!]$, which is a contradiction.

($=$) This is a trivial implication.

(S) Let us assume that $[\![t]\!] \in [\![A]\!]$ and that $[\![t]\!] \notin [\![B]\!]$. By definition of $u = [\![\varepsilon_{x \in A}(x \notin B)]\!]$ we must have $u \in [\![A]\!]$ and $u \notin [\![B]\!]$, but this contradicts the induction hypothesis.

($\forall_l$) Let us assume $[\![t]\!] \in [\![\forall X.A]\!]$ and show $[\![t]\!] \in [\![B]\!]$. Using the induction hypothesis, it is enough to show $[\![t]\!] \in [\![A[X := U]]\!]$, which is equivalent to $[\![t]\!] \in [\![A[X := [\![U]\!]]]\!]$ according to Lemma 6.15 page 32. By definition of $[\![\forall X.A]\!]$, we have $[\![t]\!] \in [\![A[X := \Phi]]\!]$ for all $\Phi \in [\![\mathcal{F}]\!]$. We can thus conclude since $[\![U]\!] \in [\![\mathcal{F}]\!]$ by Theorem 6.20 page 35.

($\forall_r$) Let us assume $[\![t]\!] \in [\![A]\!]$ and 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.

($\times_{ss}$) 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 by induction hypothesis since $[\![t]\!].l_k \in [\![A_k]\!]$ according to the definition of $[\![\{(l_i : A_i)_{i \in I}\}]\!]$.

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

($\times_{ee}$) Also similar to the ($\times_{ss}$) case.

($+$) Let us assume $[\![t]\!] \in [\![[(C_i : A_i)_{i \in I_1}]]\!]$ and show $[\![t]\!] \in [\![[(C_i : B_i)_{i \in I_2}]]\!]$. According to the ($\Rightarrow$) direction of Lemma 6.22 page 35, we know that $t \succ_H^* v$ with only two possibilities for v. In the case where $v \in \overline{\mathcal{N}}_0$ we can conclude directly using the ($\Leftarrow$) direction of Lemma 6.22. 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 $t.C_k \succ_H^* [C_k\, u \mid C_k \rightarrow \lambda x.x] \succ_H u$. Since $t \in \mathcal{N}$, we can use Lemma 6.4 page 30 to deduce that $t.C_k \in [\![A_k]\!]$. Hence, we obtain $t.C_k \in [\![B_k]\!]$ by induction hypothesis. As $[\![B_k]\!]$ is saturated by Theorem 6.20 page 35, we can use the condition (1) to get $u \in [\![B_k]\!]$. We can then conclude using the ($\Leftarrow$) direction of Lemma 6.22 again.

(μ_r) Let us assume $[\![t]\!] \in [\![A]\!]$ and show $[\![t]\!] \in [\![\mu_\kappa F]\!]$. According to the first induction hypothesis we have $[\![t]\!] \in [\![F(\mu_\tau F)]\!]$, so we only need to show $[\![F(\mu_\tau F)]\!] \subseteq [\![\mu_\kappa F]\!]$. Using the second induction hypothesis (and Lemma 2.11 page 15) we obtain $[\![\tau]\!] < [\![\kappa]\!]$. We thus have $[\![F(\mu_{[\![\tau]\!]} F)]\!] \subseteq [\![\mu_{[\![\kappa]\!]} F]\!]$ 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 6.15 page 32 twice.

(ν_l) Similar to the (μ_r) case.

(μ_r^∞) Let us assume $[\![t]\!] \in [\![A]\!]$ and show $[\![t]\!] \in [\![\mu_\infty F]\!]$. According to the induction hypothesis we have $[\![t]\!] \in [\![F(\mu_\infty F)]\!]$. Hence, we only need to show $[\![F(\mu_\infty F)]\!] \subseteq [\![\mu_\infty F]\!]$. Since the cardinal of the ordinal $[\![\infty]\!]$ is $2^{2^{\aleph_0}}$, it is larger than the cardinal of $[\![\mathcal{F}]\!]$ which is $2^{\aleph_0}$. Therefore, the inductive definition of $[\![\mu_{[\![\infty]\!]} F]\!]$ must reach its stationary point strictly before $[\![\infty]\!]$.[17] As a consequence, we have $[\![\mu_{[\![\infty]\!]} F]\!] = [\![\mu_{[\![\infty]\!]+1} F]\!] \supseteq [\![F(\mu_{[\![\infty]\!]} F)]\!]$ by definition, and we can thus conclude using Lemma 6.15 page 32 on both sides.

(ν_l^∞) Similar to the (μ_r^∞) case.

[17]This stationary point is not a fixed point when F is not covariant, but we do not need this information.

(μ_l) Let us assume $[\![t]\!] \in [\![\mu_\kappa F]\!]$ and show $[\![t]\!] \in [\![B]\!]$. 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 6.20 page 35. 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 6.24. *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 6.23 page 36), it must be that $[\![t]\!] \in \overline{\mathcal{N}_0}$ since $[\![\forall X.X]\!] = [\![[\,]]\!] = \overline{\mathcal{N}_0}$. This is a contradiction since $\overline{\mathcal{N}_0}$ only contains open terms. □

Theorem 6.25. *Let $t \in [\![\Lambda]\!]$ be a closed, pure term, and $A \in \mathcal{F}$ be a closed type. If $\vdash t : A$ is derivable, then t is strongly normalising (i.e., we have $t \in \mathcal{N}$).*

Proof. Using the adequacy lemma (Theorem 6.23 page 36), we know that $[\![t]\!] \in [\![A]\!]$. However, since t is pure, it is easy to see that $[\![t]\!] = t$. Moreover, we have $[\![A]\!] \subseteq \mathcal{N}$ according to Theorem 6.20 page 36, and hence we have $t \in \mathcal{N}$. □

Note that, as a direct consequence of strong normalisation, we know that a well-typed term cannot produce runtime errors such as the projection of a record field on a λ-abstraction. Indeed, the reduction relation ($\succ$) of Definition 4.4 page 24 introduces non-termination when such errors are encountered. We will now consider a stronger safety result that will be limited to so-called *simple* types, which include lists and binary trees for example.

Definition 6.26. *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 fixed points carrying the ∞ ordinal. Moreover, we will assume that a simple type A does not have two consecutive least fixed points, and that the body of fixed points is not limited to a variable like in $\mu X.Y$ or $\mu X.X$.*

Theorem 6.27. *Let t be a closed, pure term, and A be a simple type. If $\vdash t : A$ is derivable, then there is a term u in normal form such that $t \succ^* u$ and $\vdash u : A$ is derivable.*

Proof. According to Theorem 6.25 page 38, 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. If $A = \mu X.B$ then 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 fixed points are still true in A' since B cannot be equal to X by hypothesis. Moreover, pure types cannot contain two consecutive fixed points. Hence A' is either a sum type or a strict product type.

If $A' = [(C_i : A_i)_{i \in I}]$ then by Lemma 6.22 page 35 we know that $u = C_k\,v$ with $k \in I$ and $v \in [\![A_k]\!]$. As u is in normal form (and hence in head normal form) and it is close, it must be that $u \notin \overline{\mathcal{N}_0}$. Moreover, we know that v is also in normal form (it is a subterm of u). Hence, the induction hypothesis gives us a derivation of $\vdash v : A_k$. In the case where $A' = A = [(C_i : A_i)_{i \in I}]$, we can thus conclude using the following derivation.

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

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 should rather be of the form $A_k[X := A]$, so the induction hypothesis gives a proof of $\vdash v : A_k[X := A]$.

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

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 $u = \{(l_i = v_i)_{i \in I}\}$ with $v_i \in [\![A_i]\!]$ for all $i \in I$. Indeed, all the other possible forms of normal forms can be ruled out using similar techniques as in the proof of Lemma 6.22 page 35. The induction hypotheses thus give proofs of $\vdash v_i : A_i$ for all $i \in I$, and we can then reconstruct as we did for 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 6.27, the main aim of this condition is to allow for the correctness of the subtyping rule for sums recalled below.

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

Indeed, closure under head reduction is necessary to accommodate the term of the form $t.C_i$ that is used in the rule. It would be possible to use more complex term, similar to the one that would be introduced using the following encoding of sums as products.

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

However, there is a fundamental problem with this encoding as the term would contain all the types A_i and B_i due to the subtyping rule on the arrow types. As a consequence, such terms would prevents the derivation of subtyping relations like $\forall X[C : A] \subseteq [C : \forall XA]$ or $[C : \exists XA] \subseteq \exists X[C : A]$. On the contrary, terms of the form $t.C_i$ do not contain any of the A_i or B_i type, and does not suffer from this limitation.

7 FIXED POINT AND TERMINATION

We will now extend the system with general recursion using a fixed point 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 fixed point. Nonetheless, we will prove normalisation for all the weak reduction strategies: those that do not reduce under λ-abstractions (and hence under the right members of case analyses).

To establish the termination of certain programs, we will sometimes need to show that some functions are size-preserving. For example, proving the termination of the usual quicksort function we will require the partitioning function to produce lists that are no bigger than the input list. To this aim, we extend the syntax of types with quantification over ordinals. It can be used to express size invariants like in the following type, where $\mathrm{List}(A, \alpha) = \mu_\alpha L.[\mathrm{Nil} \mid \mathrm{Cons~of~} A \times L]$ is the type of lists of size at most α with elements in A.

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

Finally, proving the termination of recursive programs will often require our typing judgements to carry an ordinal contexts. We will then be able to assume that certain ordinals are positive while building typing proofs. For example, if we know that $l : \mathrm{List}(A, \alpha)$ and want to type the case analysis $[l \mid \mathrm{Nil} \to u \mid \mathrm{Cons} \to v]$, then we may safely assume $\alpha > 0$ when typing u and v. Indeed, if $\alpha = 0$ then l can only be a neutral term, and hence $[l \mid \mathrm{Nil} \to u \mid \mathrm{Cons} \to v] : C$ is trivially valid in the semantics. Without this technique, we would for example not be able to use the type given above for the partitioning function. To transfer positivity hypotheses from local subtyping judgements to typing judgements, we will rely on new connectives $A \land \kappa$ and $\kappa \hookrightarrow A$, where κ is a syntactic ordinal. The former will be interpreted as A if κ is non-zero and as $\forall X.X$ otherwise, and the latter will be interpreted as A if κ is non-zero and as $\exists X.X$ otherwise. They will be used to strengthen our typing rules, and will be handled using specific local subtyping rules.

Definition 7.1. *We extend the syntax of terms and types given in Definition 4.1 page 22 with a fixed point for general recursion, ordinal quantifiers, and ordinal conjunction and implication.*

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

$$A, B ::= \cdots \mid \forall\alpha.A \mid \exists\alpha.A \mid A \land \kappa \mid \kappa \hookrightarrow A$$

Note that this new definition also impacts that of abstract judgements and syntactic ordinals, although they still have the same form.

Convention 7.2. *We will use the abbreviations $A \land \gamma$ and $\gamma \hookrightarrow A$, where $\gamma = \kappa_1, \ldots, \kappa_n$ is an ordinal context, to denote $((A \land \kappa_1) \ldots) \land \kappa_n$ and $\kappa_1 \hookrightarrow (\ldots(\kappa_n \hookrightarrow A))$ respectively. In particular, if γ is empty then $A \land \gamma = \gamma \hookrightarrow A = A$. We will write γ, δ for the union of ordinal contexts γ and δ.*

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

Definition 7.4. *Our typing judgements are now of the form $\gamma \vdash t : A$, as they carry an ordinal context. The typing rules of the system are replaced with those of Figure 12. The local subtyping rules of the system are still those of Figure 7 page 25, but they are extended with the rules of Figure 13 page 41 to handle the new connectives.*

The typing rules of the system need to be changed completely to account for ordinal contexts. Note that they are strongly linked to the new connectives $A \land \kappa$ and $\kappa \hookrightarrow A$ in types. Moreover, the $(\times_i)$ and $(+_i)$ rules may propagate positive ordinals under some weak normality conditions.

Definition 7.5. *The type system of our language is now circular deduction system (see Definition 3.8 page 18) induced by the typing and subtyping rules of Figure 7 page 25, Figure 12 page 41, and Figure 13. As in Section 4 page 22, we only consider limited forms of general abstract sequents. They will be of the form $\forall\overline{\alpha}(\gamma \vdash C(\overline{\alpha}) \Rightarrow A \subseteq B)$, or of the form $\forall\overline{\alpha}(\gamma \vdash C(\overline{\alpha}) \Rightarrow Yx.t : A)$. As a consequence, we will only apply the restricted generalisation and induction rules of Figure 8 page 26, and of Figure 14.*

The system considered here still uses circular subtyping proofs to handle inductive and coinductive types, but it also relies on circular typing proofs for general recursion. Note that the typing rule for the fixed point $Yx.t$ is extremely simple, as it only performs an unfolding. We may limit the application of the (G) rule in typing derivations because only the (Y) rule introduces circularity. Note that in the I_k rule of Figure 14 we use the notation $\varepsilon_{\overline{\alpha}<C(\overline{\alpha})}(Yx.t \notin A)_i$ for the syntactic ordinal $\varepsilon_{\overline{\alpha}<C(\overline{\alpha})}\neg(Yx.t : A)_i$ (see Definition 3.2 page 17). Similar notations are used in the $(\forall^o_r)$ and $(\exists^o_l)$ rules of Figure 13. In fact, the ordinal choice operator $\varepsilon_{\alpha<O}(t \in B)$ of the $(\exists^o_l)$ formally requires the introduction of a specific form of abstract judgement (with no deduction rule).

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

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

$$\frac{\gamma \vdash \{(l_i = t_i)_{i \in I}\} \in \delta \hookrightarrow \{(l_i : A_i)_{i \in I}\} \subseteq B \qquad (\gamma, \delta \vdash t_i : A_i)_{i \in I} \qquad \delta = \varnothing \text{ or } (t_i{\downarrow})_{i \in I}}{\gamma \vdash \{(l_i = t_i)_{i \in I}\} : B} \times_i$$

$$\frac{\gamma \vdash C\,t \in \delta \hookrightarrow [C \text{ of } A] \subseteq B \qquad \gamma, \delta \vdash t : A \qquad \delta = \varnothing \text{ or } t{\downarrow}}{\gamma \vdash C\,t : B} +_i$$

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

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

Fig. 12. Modified typing rules for the extended system (additions are highlighted in blue).

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

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

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

$$\frac{\gamma, \kappa \vdash t \in A \subseteq B}{\gamma \vdash t \in A \subseteq \kappa \hookrightarrow B} \hookrightarrow_r \qquad \frac{\gamma \vdash t \in A \subseteq B \qquad \kappa \in \gamma}{\gamma \vdash t \in \kappa \hookrightarrow A \subseteq B} \hookrightarrow_l$$

Fig. 13. Additional subtyping rules for the extended system.

Example 7.6. The typing derivation of the recursive identity function on unary natural numbers is given in Figure 15 page 43. Following the terminology of Section 3 page 17, the proof has two blocks B_0 and B_1. The former starts at the root of the proof and only contains one typing rule, and the latter spans the rest of the proof. The call graph has one edge going from B_0 to B_1 with the empty matrix, and a loop on B_1 with the 1×1 matrix (-1) since we can prove $\kappa_0 \vdash \kappa_1 < \kappa_0$. Note that we must have κ_0 in the ordinal context to get $\kappa_1 < \kappa_0$. It is thus essential to use the type $[\mathrm{Z} \mid \mathrm{S} \text{ of } \mathbb{N}_{\kappa_1}] \wedge \kappa_0$ and not just $[\mathrm{Z} \mid \mathrm{S} \text{ of } \mathbb{N}_{\kappa_1}]$ in the first premise of the $(+_e)$ rule.

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

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

Fig. 14. Specialised induction and generalization rules for general recursion.

We will now modify the semantics that was given in Section 6 page 30, to account for the fixed point combinator and for the new connectives. We will not be able to interpret types as subsets of $\mathcal{N}$ anymore, since the reduction rule of the fixed point breaks strong normalisation. We will however preserve a normalisation property for all the *weak* reduction strategies (i.e., those that do not reduce under λ-abstractions, and thus case analyses).

Definition 7.7. *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 5 page 24 together with*

$$Yx.t \succ_w t[x := Yx.t]$$

and that is contextually closed for weak contexts (i.e., contexts whose hole is not placed in the body of a λ-abstraction). Its reflexive, transitive closure is denoted $(\succ_w^*)$.

Definition 7.8. *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 $(\succ_w)$.*

As in Section 6 page 30, we define a notion of (weakly) saturated sets as well as a set of (weak) neutral terms $\mathcal{W}_0$. The semantics is then defined accordingly, using the same techniques. In particular, the first five conditions of Definition 7.9 below exactly correspond to those of Definition 6.3 page 30, where $\mathcal{N}$ has been replaced by $\mathcal{W}$.

Definition 7.9. *A set of pure terms $\Phi \subseteq [\![\Lambda]\!]$ is said to be* weakly saturated *if it satisfies the following six conditions.*

(1) If $t \in \Phi$ and $t \succ_H u$, then $u \in \Phi$.
(2) If $H[t[x := u]] \in \Phi$ and $u \in \mathcal{W}$, then $H[(\lambda x.t)\ u] \in \Phi$.
(3) If $H[t\ u] \in \Phi$, then $H[[C_k\ u \mid C_k \to t]] \in \Phi$.
(4) If $H[t] \in \Phi$, $k \notin I$ and $t_i \in \mathcal{W}$ for all $i \in I$, then $H[\{l_k = t; (l_i = t_i)_{i \in I}\}.l_k] \in \Phi$.
(5) If $H[t \mid (C_i \to t_i)_{i \in I}] \in \Phi$, $I \subset J$ and $t_j \in \mathcal{W}$ for all $j \in J \setminus I$, then $H[t \mid (C_i \to t_i)_{i \in J}] \in \Phi$.
(6) If $H[t[x := Yx.t]] \in \Phi$, then $H[Yx.t] \in \Phi$.

Similarly, Definition 7.10 below corresponds to Definition 6.6 page 31, where $\mathcal{N}$ has been replaced by $\mathcal{W}$, and $\mathcal{N}_0$ by $\mathcal{W}_0$.

Definition 7.10. *The set of* weak neutral terms *$\mathcal{W}_0 \subset [\![\Lambda]\!]$ is the smallest set such that:*

(1) for every λ-variable $x \in \mathcal{V}_\Lambda$ we have $x \in \mathcal{W}_0$,
(2) for every $u \in \mathcal{W}$ and $t \in \mathcal{W}_0$ we have $t\ u \in \mathcal{W}_0$,
(3) for every $i \in \mathbb{N}$ and $t \in \mathcal{W}_0$ we have $t.l_i \in \mathcal{W}_0$,
(4) for every set of indices I, terms $(t_i)_{i \in I} \in \mathcal{W}^I$ and $t \in \mathcal{W}_0$ we have $[t \mid (C_i \to t_i)_{i \in I}] \in \mathcal{W}_0$.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\kappa_0 \vdash n_0 \in [\mathrm{Z} \mid \mathrm{S} \text{ of } \mathbb{N}_{\kappa_1}] \subseteq [\mathrm{Z} \mid \mathrm{S} \text{ of } \mathbb{N}_{\kappa_1}]}\,=}{\kappa_0 \vdash n_0 \in [\mathrm{Z} \mid \mathrm{S} \text{ of } \mathbb{N}_{\kappa_1}] \subseteq [\mathrm{Z} \mid \mathrm{S} \text{ of } \mathbb{N}_{\kappa_1}] \wedge \kappa_0}\,\wedge_r}{\vdash n_0 \in \mathbb{N}_{\kappa_0} \subseteq [\mathrm{Z} \mid \mathrm{S} \text{ of } \mathbb{N}_{\kappa_1}] \wedge \kappa_0}\,\mu_l}{\vdash n_0 : [\mathrm{Z} \mid \mathrm{S} \text{ of } \mathbb{N}_{\kappa_1}] \wedge \kappa_0}\,\varepsilon \qquad \dfrac{}{\kappa_0 \vdash \mathrm{z} : \mathbb{N}} \qquad \dfrac{\dfrac{\dfrac{}{\kappa_0 \vdash \mathrm{s} : \mathbb{N} \to \mathbb{N}} \quad \dfrac{\dfrac{[\forall\alpha_0(\vdash \mathrm{id} : \mathbb{N}_{\alpha_0} \to \mathbb{N})]_0}{\kappa_0 \vdash \mathrm{id} : \mathbb{N}_{\kappa_1} \to \mathbb{N}}\,\mathrm{G} \quad \dfrac{}{\kappa_0 \vdash p_0 : \mathbb{N}_{\kappa_1}}\,\varepsilon}{\kappa_0 \vdash \mathrm{id}\ p_0 : \mathbb{N}}\,\to_e}{\kappa_0 \vdash \mathrm{s}\ (\mathrm{id}\ p_0) : \mathbb{N}}\,\to_e}{\kappa_0 \vdash \lambda p.\mathrm{s}\ (\mathrm{id}\ p) : \mathbb{N}_{\kappa_1} \to \mathbb{N}}\,\to_i}{\vdash [n_0 \mid Z \to \mathrm{z} \mid Sp \to \mathrm{s}\ (\mathrm{id}\ p)] : \mathbb{N}}\,+_e}{\vdash \lambda n.[n \mid Z \to \mathrm{z} \mid Sp \to \mathrm{s}\ (\mathrm{id}\ p)] : \mathbb{N}_{\kappa_0} \to \mathbb{N}}\,\to_i}{\vdash \mathrm{id} : \mathbb{N}_{\kappa_0} \to \mathbb{N}}\,Y}{\forall\alpha_0(\vdash \mathrm{id} : \mathbb{N}_{\alpha_0} \to \mathbb{N})}\,\mathrm{I}_0}{\vdash \mathrm{id} : \mathbb{N} \to \mathbb{N}}\,\mathrm{G}
$$

where

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

Fig. 15. Circular typing derivation for the recursive identity function on unary natural numbers.

We denote $\overline{\mathcal{W}_0}$ the least weakly saturated set containing $\mathcal{W}_0$.

With the above definitions, we can obtain similar properties as in Section 6 page 30. This is mainly due to the fact that the proofs of theses lemmas do not considers reductions which are allowed for $(\succ)$ but not for $(\succ_w)$. We will start by showing that $\mathcal{W}$ is weakly saturated, which requires the following lemma (it was immediate in Section 6).

Lemma 7.11. *Let $t, u, u' \in \mathcal{W}$ be terms such that $u \succ_w^* u'$. If $t[x := u']$ admits an infinite sequence of reductions using $(\succ_w)$, then so does $t[x := u]$.*

Proof. We reason coinductively. We will split the occurrences of the free variables x of t into two groups. Those that appear under a λ-abstraction are denoted x_0 and the others x_1. We thus have $t[x := u] = (t[x_1 := u])[x_0 := u]$ and $t[x := u'] = (t[x_1 := u'])[x_0 := u']$. Let us now consider the first step of an infinite reduction of $t[x := u']$. It must start with $t[x := u'] \succ_w t'[x_0 := u']$, where $t[x_1 := u'] \succ_w t'$, since there cannot be any weak reduction step in the occurrences of u' substituting 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 7.12. *The set $\mathcal{W}$ is weakly saturated.*

Proof. For condition (1) and conditions (3) to (5), the proof is exactly the same as in Lemma 6.5 page 30, replacing $\mathcal{N}$ by $\mathcal{W}$. The remaining two conditions are proved below.

For condition (2), 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 suppose, by contradiction, that $H[(\lambda x.t)\, u]$ has an infinite reduction using $(\succ_w)$. It 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 $u \succ_w^* u'$. Hence, it can be transformed into $H[t[x := u]] \succ_w^* H'[t[x := u]]$, and we can use Lemma 7.11 page 44 to obtain an infinite reduction of $H'[t[x := u]]$ from the infinite reduction of $H'[t[x := u']]$. This gives a contradiction with $H[t[x := u]] \in \mathcal{W}$.

For condition (6), we need to prove that if $H[t[x := Yx.t]] \in \mathcal{W}$, then $H[Yx.t] \in \mathcal{W}$. We suppose, by contradiction, that $H[Yx.t] \in \mathcal{W}$ has an infinite reduction using $(\succ_w)$. It must start with $H[Yx.t] \succ_w^* H'[Yx.t] \succ_w^* H'[t[x := Yx.t]]$ where $H \succ_w^* H'$. Hence, it can be transformed into $H[Yx.t] \succ_w^* H[t[x := Yx.t]] \succ_w^* H'[t[x := Yx.t]]$, which contradicts $H[t[x := Yx.t]] \in \mathcal{W}$. □

We will now consider the interpretation of terms, types and syntactic ordinals to handle the fixed point and the new connectives.

Definition 7.13. *The set of every type interpretation $[\![\mathcal{F}]\!]$ is now defined as follows.*

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

Definition 7.14. *We modify the definition of the interpretation of terms and formulas given in Figure 11 page 33, replacing every occurrence of $\mathcal{N}$ and $\overline{\mathcal{N}_0}$ by $\mathcal{W}$ and $\overline{\mathcal{W}_0}$ respectively. The new syntactic elements are interpreted as follows.*

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

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

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

Theorem 7.15. *For every closed parametric term $t \in \Lambda^*$ (resp. ordinal $\kappa \in O^*$, resp. type $A \in \mathcal{F}^*$) we have $[\![t]\!] \in [\![\Lambda]\!]$ (resp. $[\![\kappa]\!] \in [\![O]\!]$, resp. $[\![A]\!] \in [\![\mathcal{F}]\!]$).*

Proof. The proof is similar to that of Theorem 6.20 page 35, the cases for the new term and type constructors being immediate by induction hypothesis. □

The adequacy lemma of the new system will be similar to that given in Section 6 page 30. For this reason, we will not state all the required lemmas like the one corresponding to Theorem 6.15 page 32, as their proofs do not need to be changed much. We however need a small lemma for handling the weak normality condition in some of our new typing rules.

Lemma 7.16. *If $t \in \Lambda$ is weakly normal (written $t\downarrow$), then $[\![t]\!] \in \mathcal{W}$.*

Proof. Immediate by induction, using Theorem 7.15 in the case where $t = \varepsilon_{x\in A}(t \notin B)$. □

Theorem 7.17. *Let γ be an ordinal context such that $[\![\tau]\!] > 0$ for all $\tau \in \gamma$.*

(1) If $\gamma \vdash t \in A \subseteq B$ 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 6.23 page 36. For the local subtyping rules of Figure 7, 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 page 41, and the local subtyping rules of Figure 13 page 41.

- $(\to_i)$ We need to show $[\![\lambda x.t]\!] \in [\![C]\!]$. According to the first induction hypothesis, it is enough to show $[\![\lambda x.t]\!] \in [\![\delta \hookrightarrow (A \to B)]\!]$. If there is $\kappa \in \delta$ such that $[\![\kappa]\!] = 0$ then $[\![\delta \hookrightarrow (A \to B)]\!] = \mathcal{W}$ and we can conclude immediately by Lemma 7.16 page 45. We can thus assume that $[\![\kappa]\!] \neq 0$ for all $\kappa \in \delta$. Therefore, the positivity context of the second induction hypothesis is valid and we 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 have $[\![\lambda x.t]\!] \in [\![A \to B]\!] = [\![\delta \hookrightarrow (A \to B)]\!]$ 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 we have $[\![t]\!] \in [\![(A \to B) \wedge \delta]\!]$. If $[\![\kappa]\!] = 0$ for some $\kappa \in \delta$ then $[\![(A \to B) \wedge \delta]\!] = \overline{\mathcal{W}_0}$, and thus we have $[\![t]\!] \in \overline{\mathcal{W}_0}$ which implies $[\![t\ u]\!] \in \overline{\mathcal{W}_0} \subseteq [\![B]\!]$. Otherwise we have $[\![t]\!] \in [\![A \to B]\!]$, and we can thus conclude using the definition of $[\![A \to B]\!]$ with the second induction hypothesis.
- $(\times_i)$ We only need to show $[\![\{(l_i : A_i)_{i\in I}\}]\!] \in [\![\delta \hookrightarrow \{(l_i : A_i)_{i\in I}\}]\!]$ thanks to the first induction hypothesis. If $[\![\kappa]\!] = 0$ for some $\kappa \in \delta$ and if $t_i\downarrow$ for all $i \in I$, then we can conclude immediately by Lemma 7.16 page 45 as $[\![\delta \hookrightarrow \{(l_i : A_i)_{i\in I}\}]\!] = \mathcal{W}$. Otherwise, the remaining induction hypotheses give $[\![t_i]\!] \in [\![A_i]\!]$ for all $i \in I$, which implies $[\![\{(l_i = t_i)_{i\in I}\}]\!] \in [\![\{(l_i : A_i)_{i\in I}\}]\!]$ by weak saturation. We can then conclude since $[\![\delta \hookrightarrow \{(l_i : A_i)_{i\in I}\}]\!] = [\![\{(l_i : A_i)_{i\in I}\}]\!]$.
- $(+_i)$ According to the first induction hypothesis, we only need to prove $[\![C\ t]\!] \in [\![\delta \hookrightarrow [C \text{ of } A]]\!]$. If $[\![\kappa]\!] = 0$ for some $\kappa \in \delta$ and if t is weakly normal, then we can conclude immediately using Lemma 7.16 page 45. Otherwise, $[\![\kappa]\!] \neq 0$ for all $\kappa \in \delta$. We can thus use the second induction hypothesis to get $[\![t]\!] \in [\![A]\!]$, from which 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 $[\![t]\!] \in [\![[(C_i : A_i)_{i\in I}] \wedge \delta]\!]$. If $[\![\kappa]\!] = 0$ for some $\kappa \in \delta$ then we have $[\![t]\!] \in \overline{\mathcal{W}_0}$, and thus we obtain $[\![[t \mid (C_i \to t_i)_{i\in I}]]\!] \in \overline{\mathcal{W}_0} \subseteq [\![B]\!]$. Otherwise, the result follows from the remaining 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 (6) on $[\![A]\!]$.
- $(\forall_l^o)$ If $[\![t]\!] \in [\![\forall\alpha.A]\!]$ then we have $[\![t]\!] \in [\![A[\alpha := [\![\kappa]\!]]]\!] = [\![A[\alpha := \kappa]]\!]$ 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 $o \in [\![O]\!]$ such that $[\![t]\!] \notin [\![B[x := o]]\!]$. By definition of the choice operator, this means means that $[\![t]\!] \notin [\![B[x := \varepsilon_{\alpha<O}(t \notin B)]]\!]$. 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.

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

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

Theorem 7.18. *As for the initial system, we get consistency, termination (typed terms are normalising for every weak reduction strategy), and type safety for simple types.*

Proof. The proofs are similar to those of Theorem 6.24 page 38, and Theorems 6.25 and 6.27 page 38 respectively, using the results given in the current section. □

8 TERMINATING EXAMPLES

We now consider several examples of functions that are typable in our system, and accepted by our implementation. We 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 page 43.

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 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 with 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 use the type $\forall A\,B\,C.\forall\alpha\,\beta, (A \to B \to C) \to \mathbb{L}_\alpha(A) \to \mathbb{L}_\beta(B) \to \mathbb{L}_{\min(\alpha,\beta)}(C)$. Nonetheless, it is important to note that the types of map and map$_2$ are subtypes of their usual types (with no size information). For example, we can derive

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

in our system. As a consequence, the map and map$_2$ functions of Figure 16 page 47 are suitable for all applications. In particular, we do not need to provide two different versions (with and without size information).

We will now consider the flatten function, which is also given in Figure 16 page 47. On this particular example, proving termination requires unrolling the fixed point twice. 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 some explanations about the construction of the call graph of the function, which contains four edges.

- 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.

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

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

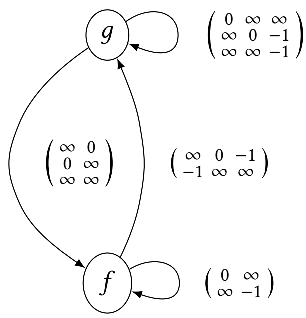
- The edge from f to g represents the definition of g inside f, which must be seen as f calling g. In this call, the first column of the matrix is justified by $\beta_0 < \alpha_1$ because β_0 is the size of the tail of the outer list. The second column is justified because β_1, the size of the inner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

To conclude with this example, we give below the idempotent loops contained in the transitive closure of the call graph, which all contains a -1 on their diagonal. There are only three of them (one on f and two on g), and two are already present in the call graph.

$$
\begin{pmatrix} 0 & \infty \\ \infty & -1 \end{pmatrix}
\qquad
\begin{pmatrix} 0 & \infty & \infty \\ \infty & 0 & -1 \\ \infty & \infty & -1 \end{pmatrix}
\qquad
\begin{pmatrix} -1 & \infty & \infty \\ \infty & 0 & -1 \\ \infty & \infty & \infty \end{pmatrix} = \begin{pmatrix} \infty & 0 \\ 0 & \infty \\ \infty & \infty \end{pmatrix} \begin{pmatrix} \infty & 0 & -1 \\ -1 & \infty & \infty \end{pmatrix}
$$

The last example given in Figure 16 page 47 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 (see Figure 18) and merge sort, but in both cases we are unable to obtain

$$Yf.\lambda l_f.\left[l_f \;\middle|\; \begin{array}{l} [] \to [] \\ l_0 :: l_1 \to \left[l_0 \;\middle|\; \begin{array}{l} [] \to f\ l_1 \\ x_1 :: l_2 \to x_1 :: Yg.\lambda l_g.\left[l_g \;\middle|\; \begin{array}{l} [] \to [] \\ l_3 :: l_4 \to \left[l_3 \;\middle|\; \begin{array}{l} [] \to f\ l_4 \\ x_2 :: l_5 \to x_2 :: g\ (l_5 :: l_4) \end{array} \right] \end{array} \right] (l_2 :: l_1) \end{array} \right] \end{array} \right]$$



$$\forall \alpha_0, \alpha_1 (\vdash \mathrm{f} : \forall A.\mathbb{L}_{\alpha_1}(\mathbb{L}_{\alpha_0}(A)) \to \mathbb{L}(A))$$
$$\forall \beta_0, \beta_1, \beta_2 (\beta_1 \vdash \beta_2 < \beta_1 \Rightarrow \mathrm{g} : [\mathrm{Cons\ of}\ \{\mathrm{hd} : \mathbb{L}_{\beta_2}(A_0);\ \mathrm{tl} : \mathbb{L}_{\beta_0}(\mathbb{L}_{\beta_1}(A_0))\}] \to \mathbb{L}(A_0))$$

Fig. 17. Unrolling of flatten and the corresponding call graph.

size preservation. It may however be possible to obtain size preservation on such a program by enriching our language of syntactic ordinals with an addition symbol for example. This would allow us to give a more precise type to the partition function used by quicksort.

Note that the quicksort example of Figure 18 uses the usual "let x = t in u" syntax (to be read as $(\lambda x.t)\ u$), and an unusual type annotation syntax "let ... such that ... in" which is explained in Section 10 page 54. It must be used because our prototype implementation is unable to guess the most general type for the recursive call in the partition function. The qsort function is however accepted without any type or size annotation.

To illustrate the use of subtyping, a simple example implementing *append lists* is provided in Figure 19. Roughly, an append list is formed like a list, but an additional constructor App 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. Of course, a list can always be used as an append list, without the need of applying fromList. A recursive function toList is however required in the other direction, to effectively concatenate the 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 page 50. 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 compose function, composing two filters.

$$
\begin{aligned}
\text{append} &: \forall A.\mathbb{L}(A) \to \mathbb{L}(A) \to \mathbb{L}(A) \\
&= Y\text{append}.\lambda l1\, l2. \left[l1 \left| \begin{array}{l} [] \to l2 \\ x :: l \to x :: \text{append } l\ l2 \end{array} \right. \right] \\
\text{partition} &: \forall A.\forall \alpha.(A \to \text{Bool}) \to \mathbb{L}_\alpha(A) \to \mathbb{L}_\alpha(A) \times \mathbb{L}_\alpha(A) \\
&= Y\text{partition}.\lambda test\, l. \left[l \left| \begin{array}{l} [] \to ([],[]) \\ x :: l \to \quad \text{let } \alpha, A \text{ such that } l : \mathbb{L}_\alpha(A) \text{ in} \\ \qquad\qquad \text{let } c : \mathbb{L}_\alpha(A) \times \mathbb{L}_\alpha(A) = \text{partition } test\ l \text{ in} \\ \qquad\qquad \left[test\ x \left| \begin{array}{l} Tru \to (x :: c.1, c.2) \\ Fls \to (c.2, x :: c.1) \end{array} \right. \right] \end{array} \right. \right] \\
\text{qsort} &: \forall A.(A \to A \to \text{Bool}) \to \mathbb{L}(A) \to \mathbb{L}(A) \\
&= Y\text{qsort}.\lambda cmp\, l. \left[l \left| \begin{array}{l} [] \to [] \\ x :: l \to \quad \text{let } test = cmp\ x \text{ in} \\ \qquad\qquad \text{let } c = \text{partition } test\ l \text{ in} \\ \qquad\qquad \text{let } below = \text{qsort } cmp\ c.1 \text{ in} \\ \qquad\qquad \text{let } above = \text{qsort } cmp\ c.2 \text{ in} \\ \qquad\qquad \text{append } below\ (x :: above) \end{array} \right. \right]
\end{aligned}
$$

Fig. 18. Example of quicksort.

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

Fig. 19. Append lists as a supertype of lists.

As in the example of the flatten function on lists, both filter and compose require some unrolling. To avoid this, we may replace the type $\mathbb{F}$ with the following type.

$$\mathbb{F}' \quad = \mu Y.(\{\} \to [\text{R of } Y \mid \text{K of } \mathbb{F}])$$

Although $\mathbb{F} \subseteq \mathbb{F}'$ and $\mathbb{F}' \subseteq \mathbb{F}$ are both derivable, $\mathbb{F}'$ carries an ordinal representing the initial number of R constructors in the type. The call graph for compose, given in Figure 21 page 51, is 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 [37]. They contain, for example, the GCD function for binary natural numbers, and the basic operations for exact real arithmetic using the signed digits representation. Of course, these examples are proved terminating by our implementation.

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

Fig. 20. Examples with streams and filters on streams.

9 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. Indeed, only one typing rule applies for each term constructor, and at most two subtyping rules apply for each pair of type constructors.[18] When two different subtyping rules may apply, it is easy to see that they commute. For example, this is the case for the left and right quantifier rules.

$$
\dfrac{\dfrac{\gamma \vdash t \in A[X := C] \subseteq B[Y := \varepsilon_Y(t \notin B)]}{\gamma \vdash t \in A[X := C] \subseteq \forall Y.B}\ \forall_r}{\gamma \vdash t \in \forall X.A \subseteq \forall Y.B}\ \forall_l
\qquad
\dfrac{\dfrac{\gamma \vdash t \in A[X := C] \subseteq B[Y := \varepsilon_Y(t \notin B)]}{\gamma \vdash t \in \forall X.A \subseteq B[Y := \varepsilon_Y(t \notin B)]}\ \forall_l}{\gamma \vdash t \in \forall X.A \subseteq \forall Y.B}\ \forall_r
$$

This is due to the fact that these rules do not modify the term that is carried by the judgement, and that the definition of the involved choice operators only relies on this term and on the type that is transformed by the rule.

Another important remark about the system is that, if we limit the unrolling depth for fixed points in typing rules, then the only possible place where an implementation may loop is in the subtyping function. Indeed, every typing rule (except fixed point unrolling) decreases the size of the term (choice operators having a size of zero).

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 [37], 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

[18] Note that we ignore the (S) rule, which is closely related to the construction of circular proofs.

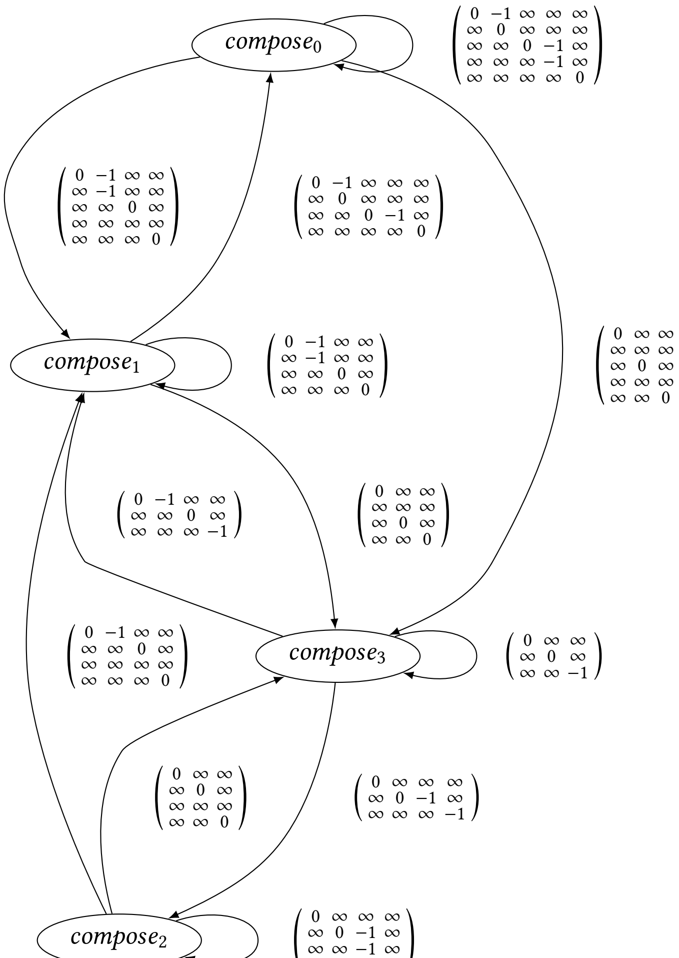


Fig. 21. Call-graph of the compose function.

management of the rules introducing unknown types or ordinals (namely $(\rightarrow_e)$, $(\forall_l)$, $(\exists_r)$, $(\forall_l^o)$, $(\exists_r^o)$, (μ_r) and (ν_l)), in the management of the ordinal contexts with the $A \wedge \gamma$ and $\gamma \hookrightarrow A$ 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 [37], unification variables are handled as follows.

- If we encounter $\gamma \vdash t \in U \subseteq U$ then we use reflexivity.
- If we encounter $\gamma \vdash t \in U \subseteq A$ or $\gamma \vdash t \in A \subseteq U$, then we set $U := A$, provided that U does not occur in A.

Note that in the latter case, it is essential to check for the occurrence of the unification variable inside choice operators, for them to be well-defined (i.e., not cyclic). This occur-check replaces that usual eigenvariable constraints, and it is essential for correction. 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 infer recursive types.

In fact, this approach is too naive in the case where we have a projection $t.l_k$ and the type of t is a unification variable U. Indeed, this will lead to proving a judgement $\vdash t \in U \subseteq \{l_k : V\}$ and 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 variant construction. To solve these issues, each unification variable keeps track of projected fields or constructed variants. This information is initialised or updated when we encounter $\gamma \vdash t \in U \subseteq \{l_1 : A_1, \dots, l_n : A_n, \dots\}$ or $\gamma \vdash t \in [C_1 \text{ of } A_1, \dots, C_n \text{ of } A_n] \subseteq U$. These can be seen as a subtyping constraints (upper bound for record types, lower bound for variant types) which are 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 unification variables. As for types, an ordinal unification variable O may carry constraints like $\tau_1 \leq O < \tau_2$ to delay instantiation until we have a constraint $O \leq \kappa$. Then, we try to set O to either κ or τ_1 to satisfy the constraints. Moreover, when we need to prove $\gamma \vdash t \in A \subseteq \mu_O F$ or $\gamma \vdash t \in \nu_O F \subseteq B$ and O is a unification variable, we must find a positive value for O to be able to apply the μ_r or ν_l rules. In these cases, we define O to be the first ordinal in γ satisfying the constraints on O. If there is none, then we try to instantiate O with ∞, and if the constraints are not satisfied, then we instantiate O with the successor of a fresh unification variable. Instantiating with a successor must be tried as a last resort, as is would often lead to non-termination due to the construction of decreasing chains of unification variables.

Circular subtyping proofs.

To build circular subtyping proofs, we apply the (S) rule followed by the (generalisation) rule (G). The (S) rule allows us to transform a subtyping judgement $\gamma \vdash t \in A \subseteq B$ into $\gamma \vdash \varepsilon_{x \in A}(x \notin B) \in A \subseteq B$, which only depends on the types A and B, making possible to encounter an abstract judgment of the same shape later in the proof. If the term t was kept, then we would never be able to use the induction hypotheses introduced by the (I_k) rule. Indeed, the term t always grows along a subtyping proof not using the S rule. Recall that "same shape" means same judgement up to the ordinals appearing in the judgement.

However, we still need to decide when to start the construction of a circular proof. First, as the (S) rule a creates choice operator using both types, it is better to first apply any rule using the term in the local subtyping judgement, that is the $(\forall_r)$, $(\exists_l)$, $(\forall_r^o)$, $(\exists_l^o)$, (μ_r) and (ν_l) rules. This way, the created choice operators may use less unification variables, allowing more solution for subtyping constraints. Second, we only apply the (S) and (G) rules when there is a type of the form $\mu_\tau X.A$ on the left, or a type of the form $\nu_\kappa X.B$ on the right, as circular proof are only needed for (co)-inductive 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. Finally, all the cycles are used to build the size change termination graph to check that the proof is well-founded.

Note that when there are no quantifiers, only a finite number of distinct general abstract sequents can be produced (up to ordinals). This implies the termination of our algorithm. Indeed, when proving a subtyping judgement $\gamma \vdash t \in A \subseteq B$, 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 appearing in the formula. 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 $\gamma \vdash t \in A \subseteq B$ 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 [33]. 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, although it is not hard to forge an example producing this kind of behaviour.

Circular typing proofs.

The construction of circular typing proofs follows the same principle as for circular subtyping proofs. We create a general abstract sequent using the (G) rule each time we encounter a fixed point $Yx.t$. We then check whether it was already encountered before, in which case we may conclude the proof. If it was not, then we apply the I_k rule, thus registering a new hypothesis. To prove termination of programs whose types are not annotated with sizes, the generalisation we perform 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 $\vdash Yx.t : \mu X.A \to \nu Y.\mu Z.B$ is generalised into $\forall\alpha.\forall\beta.(\vdash Yx.t : \mu_\alpha X.A \to \nu_\beta Y\mu Z.B)$. However, if the type uses ordinal quantifiers then 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 ordinal.

Breadth-first search for typing fixed points.

As explained in the previous section, unrolling a fixed point more than once is often necessary for building typing proofs. When mixed with unification, a breadth-first proof search strategy seems to succeed for more examples. This means that when typing $Yx.t$, we first finish all the other branches of the proof, collecting as much information as possible about the type of $Yx.t$. Our experimentations showed that doing so increases the chances of instantiating unification variable in the expected way.

To implement the breadth-first strategy we first apply all the typing rules on the considered term, delaying all the applications of the (Y) rule. In other words, we 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, starting with the I_k rule.

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

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

Fig. 22. Example involving dot projection (dual category).

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 whenever possible, and using imitation (i.e. constant value) otherwise. 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 judgement 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 tweaked for debugging the prototype itself, rather than for debugging programs written using it. 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, the fact that our prototype proves program termination requires an extra effort for advanced examples.

10 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 almost surely 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 coercion like $t : A$ can be easily added to the system using the following rule.

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

However, such type annotations are often required to reference bound type variables, and a type abstraction constructor $\Lambda X.t$ in terms 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.

$$\text{let } X_1, \ldots, X_n \text{ such that } x : A(X_1, \ldots, X_n) \text{ in } t$$

They allow the user to name a type (usually 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 solve the subtyping constraint $\vdash x \in T \subseteq A(U_1, \ldots, U_n)$ to obtain the value of the variables of $X_1, \ldots, X_n$ from the instantiation of the unification variables $U_1, \ldots, U_n$. For example, a fully annotated identity function can be written as follows.

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

Moreover, this kind of annotations may be used to define dot notation on existential 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.

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

As we use local subtyping when matching type, the implementation can easily search X_0 and Y_0 such that $\gamma \vdash x \in A(X_0, Y_0) \subseteq \exists X \exists Y A(X, Y)$. This will leads to $X_0 = \varepsilon_X x : \exists Y A(X, Y)$ and $Y_0 = \varepsilon_Y x : A(X_0, Y)$. 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 22 page 54).

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.

11 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 improve our system, explore its extension with several common programming features and transform it into a real programming language.

Better and simpler heuristics.

This work describes the type system of PML_2 [50] without classical logic, dependent types and specification and proof of programs. However, PML_2 has now progressed a lot and in the course of its development we have found heuristics to manage unification variables that seems to cover a comparable set of examples while being much simpler. Finding simpler and better heuristics is an important objective as it leads to a more predictable behaviour for the user of the system.

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 realizability models defined in a previous work by Rodolphe Lepigre [38]. To achieve this goal, the system is 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 are already present in PML_2 [50] and allow program proving.

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 on whether we want to have $A \subseteq ([] \to B)$ 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 or rank 2 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] Martín Abadi, Georges Gonthier, and Benjamin Werner. Choice in dynamic linking. In *IN FOSSACS'04 - FOUNDATIONS OF SOFTWARE SCIENCE AND COMPUTATION STRUCTURES 2004, LECTURE NOTES IN COMPUTER SCIENCE*, pages 12–26. Springer, 2004.

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

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

[5] 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.

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

[7] David Baelde, Amina Doumane, and Alexis Saurin. Least and greatest fixed points in ludics. In *CSL*, volume 41 of *LIPIcs*, pages 549–566. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[8] David Baelde, Amina Doumane, and Alexis Saurin. Infinitary proof theory: the multiplicative additive case. In *CSL*, volume 62 of *LIPIcs*, pages 42:1–42:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[9] John L. Bell. Hilbert's ε-operator in intuitionistic type theories. *Mathematical Logic Quarterly*, 39(1):323–337, 1993.

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

[11] 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.

[12] James Brotherston. Cyclic proofs for first-order logic with inductive definitions. In *In TABLEAUX'05, volume 3702 of LNCS*, pages 78–92. Springer-Verlag, 2005.

[13] 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.

[14] Luca Cardelli and Xavier Leroy. Abstract types and the dot notation. In *Proceedings IFIP TC2 working conference on programming concepts and methods*, pages 479–504. North-Holland, 1990.

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

[16] Julien Cretin and Didier Rémy. System F with coercion constraints. In Thomas A. Henzinger and Dale Miller, editors, *CSL-LICS Proceedings*. ACM, 2014.

[17] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *In Proceedings 9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[18] Martin Davis and Ronald Fechter. A free variable version of the first-order predicate calculus. *Journal of Logic and Computation*, 1(4):431–451, 1991.

[19] 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.

[20] Amina Doumane. *On the infinitary proof theory of logics with fixed points. (Théorie de la démonstration infinitaire pour les logiques à points fixes)*. PhD thesis, Paris Diderot University, France, 2017.

[21] Joshua Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 429–442, New York, NY, USA, 2013. ACM.

[22] Jérôme Fortier and Luigi Santocanale. Cuts for circular proofs. In *TACL*, volume 25 of *EPiC Series in Computing*, pages 72–75. EasyChair, 2013.

[23] Jérôme Fortier and Luigi Santocanale. Cuts for circular proofs: semantics and cut-elimination. In *CSL*, volume 23 of *LIPIcs*, pages 248–262. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.

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

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

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

[27] 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.

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

[29] 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.

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

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

[32] 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.

[33] 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.

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

[35] Didier Le Botlan and Didier Rémy. Mlf: Raising ML to the power of system F. *SIGPLAN Not.*, 38(9):27–38, August 2003.

[36] 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.

[37] R. Lepigre and C. Raffalli. SubML implementation, 2015. https://github.com/rlepigre/subml/.

[38] 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.

[39] Rodolphe Lepigre. *Semantics and Implementation of an Extension of ML for Proving Programs. (Sémantique et Implantation d'une Extension de ML pour la Preuve de Programmes).* PhD thesis, Université Grenoble Alpes, France, 2017.

[40] The Coq development team. *The Coq proof assistant reference manual.* LogiCal Project, 2004.

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

[42] 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.

[43] Ulf Norell. Dependently typed programming in agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, 2009.

[44] Miche Parigot. Un récurseur fortement normalisable et typable pour les entiers de Scott. Private communication, 1992.

[45] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *J. Funct. Program.*, 17(1):1–82, January 2007.

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

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

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

[49] C. Raffalli. The PhoX proof assistant, 2008. https://lama.univ-savoie.fr/~raffalli/phox.html.

[50] C. Raffalli. *The PML programming language*, 2012. https://lama.univ-savoie.fr/tracpml.

[51] Didier Rémy. Simple, partial type-inference for system F based on type-containment. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, pages 130–143, New York, NY, USA, 2005. ACM.

[52] Christian Retoré. Typed hilbert epsilon operators and the semantics of determiner phrases. In Glyn Morrill, Reinhard Muskens, Rainer Osswald, and Frank Richter, editors, *Formal Grammar*, pages 15–33, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

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

[54] 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.

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

[56] 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.

[57] Luigi Santocanale. From parity games to circular proofs. *Electr. Notes Theor. Comput. Sci.*, 65(1):305–316, 2002.

[58] Ulrich Schöpp and Alex Simpson. Verifying temporal properties using explicit approximants: Completeness for context-free processes. In *In FOSSACS '02*, pages 372–386. Springer-Verlag, 2002.

[59] Christoph Sprenger and Mads Dam. A note on global induction mechanisms in a μ-calculus with explicit approximations, 1999.

[60] Christoph Sprenger and Mads Dam. On the structure of inductive reasoning: Circular and tree-shaped proofs in the μ-calculus. In *IN PROCEEDINGS OF FOSSACS 2003*, pages 425–440. Springer, 2003.

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

[62] Klaus Von Heusinger. *Definite Descriptions and Choice Functions*, pages 61–91. Springer Netherlands, Dordrecht, 1997.

[63] 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.

[64] J. B. Wells. Typability is undecidable for F+eta. Technical report, Boston, MA, USA, 1996.

[65] 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.

Received July 2017; revised March 2018