Practical Subtyping for System F with Sized (Co-)Induction

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 rich, syntax-directed type system with subtyping for an extension of Curry-style System F. Our type constructors include sum and product types, universal and existential quantifiers, inductive and coinductive types. The latter two may carry annotations allowing the encoding of size invariants, which may 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 (as for Scott-encoded data types). 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. We then obtain termination using a standard (semantic) normalisation proof. To demonstrate the practicality of our 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, System F, Curry-style quantification, subtyping, polymorphism, existential types, inductive and coinductive sized-types, choice operators, realizability semantics, reducibility candidates, well-founded circular proofs, size-change principle.

ACM Reference format:
https://doi.org/0000001.0000001

1 INTRODUCTION

Polymorphism and subtyping allow for a more generic programming style. They lead to programs that are shorter, easier to understand and hence more reliable. Although polymorphism is widespread among programming languages, only limited forms of subtyping are used in practice. They usually focus on product types like records or modules [41], or on sum types like polymorphic variants [23]. 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 [28]. 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

ACM Transactions on Programming Languages and Systems, Vol. 1, No. 1, Article 1. Publication date: July 2017.
The system can be implemented using standard unification techniques, thanks to its syntax-directed typing and subtyping rules (Figures 7, page 24, and Figure 8, 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 10, page 49).

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.

\[
\begin{align*}
\Gamma \vdash t : A & \quad X \notin \text{FV}(\Gamma) \\
\Gamma \vdash \forall X.t : \forall X.A & \quad \forall_i\text{-Church} \\
\Gamma \vdash t : \forall X.A & \quad \forall_e\text{-Church}
\end{align*}
\]

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.

\[
\begin{align*}
\Gamma \vdash t : A & \quad X \notin \text{FV}(\Gamma) \\
\Gamma \vdash t : \forall X.A & \quad \forall_i\text{-Curry} \\
\Gamma \vdash t : \forall X.A & \quad \forall_e\text{-Curry}
\end{align*}
\]

However, this lighter syntax comes at the expense of the decidability of type-checking, at least in an unrestricted setting. 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 \(\forall_e\text{-Curry}\) and \(\forall_i\text{-Curry}\) rules. In general, subtyping already makes \(\forall_e\text{-Curry}\) redundant since it can be derived as follows.

\[
\begin{align*}
\Gamma \vdash t : \forall X.A & \quad X \notin \text{FV}(\Gamma) \\
\forall X.A \subseteq A[X := B] & \quad \subseteq \text{-refl} \\
\Gamma \vdash t : A[X := B] & \quad \subseteq
\end{align*}
\]

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

\[
\begin{align*}
\Gamma \vdash t : A & \quad \subseteq \text{-refl} \\
A \subseteq \forall X.A & \quad X \notin \text{FV}(A) \\
\Gamma \vdash t : \forall X.A & \quad \subseteq
\end{align*}
\]

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

\[
\begin{align*}
A \subseteq B & \quad X \notin \text{FV}(A) \\
A \subseteq \forall X.B & \quad \forall_f
\end{align*}
\]

1Languages 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.

$$
\begin{align*}
\Gamma \vdash t &\in A \subseteq A & \Rightarrow X \notin FV(\Gamma) \\
\Gamma \vdash t &\in A \subseteq \forall X.A & \subseteq \\
\Gamma \vdash t &\in \forall X.A
\end{align*}
$$

As far as the authors know, there is no other systems in which the $\forall_i$-Curry rule 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 $\Gamma$ by pushing the information it contains into $t$. This can be achieved using choice operators inspired by Hilbert’s Epsilon and Tau functions \(^27\), 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 $A \rightarrow B$. It can thus be used to build the following unusual typing rule for $\lambda$-abstractions.

$$
\varepsilon_{x \in A}(t \notin B) \vdash \lambda x.t : A \rightarrow B
$$

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

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

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.

$$
\Gamma \vdash t : A \rightarrow B \quad u : A \\
\Gamma \vdash t u : B
$$

In fact, the typing rules of our system (Figure 7, 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

\(\text{They are actually used to handle a specific membership type constructor } t \in A \text{ in the type system of PML}_2 [38].\)

\(\text{This particular point will be discussed at the end of the introduction, in relation with related work (see page 11).}\)

\(\text{Our model being based on reducibility candidates [24, 25], the interpretation of a type is never empty.}\)

\(\text{We will still use a form of context to store ordinals assumed to be nonzero (Section 4).}\)
$t$ 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.

$$
\begin{array}{c}
t \in A \subseteq B[X := t \notin B] \\
\forall_r
\end{array}$$

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 $\forall_r$-Curry may be derived as follows.

$$
\begin{array}{c}
t : A[X := \varepsilon_X(t \notin A)] \\
\forall_r
\end{array}$$

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 7, 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 [40]

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

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

$$
\begin{array}{c}
t \in A \subseteq B \\
t \in A \subseteq C \\
\forall_r
\end{array}$$

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, we conjecture that the above transitivity rule is 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 fixpoint of a covariant parametric type $F$. In our system, the subtyping rules are so fine-grained that no syntactic covariance condition is required on such types. In fact, 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 $\mu X.F(X) \subseteq F(\nu X.F(X))$.\(^7\) 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 $\kappa$ to form sized types $\mu_\kappa X.F(X)$ and $\nu_\kappa X.F(X)$ [4, 28, 54]. They intuitively correspond to $\kappa$ 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 $\kappa$, even if $F$ is not covariant. For cardinality reasons, this implies that there is an ordinal $\infty$ from

\(^6\)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.

ACM Transactions on Programming Languages and Systems, Vol. 1, No. 1, Article 1. Publication date: July 2017.
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 fixpoints of $F$ when it is covariant. If $F$ is not covariant then these stationary points are not fixpoints.

In this paper, we introduce a uniform induction rule for local subtyping, expressed in new circular proof framework (Section 3, page 16). 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 [35], which allows us to check for well-foundedness a posteriori. Our system is able to deal with subtyping involving mixed inductive and coinductive types. For example, when $F$ is covariant, it is able to derive $\mu X.\nu Y.F(X, Y) \subseteq \nu Y.\mu X.F(X, Y)$.

When we restrict ourselves to types without universal and existential quantifiers, our experiments tend to indicate that our system is in some sense complete. However, we failed to prove completeness in the presence of function types, the main problem being the mere definition of completeness in this setting. For instance, axioms like $A \subseteq (\forall X.X) \rightarrow B$ 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 fixpoint combinator $Yx.t$, reduced using the rule $Yx.t \triangleright t[x := Yx.t]$. 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 [35]. Given its simplicity, our system is surprisingly powerful. In particular, it is possible to unfold a fixpoint several times to obtain a well-founded circular proof (see Section 8, page 44).

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 a program directly, but rather show that its circular typing proof is well-founded. Normalisation is then established indirectly, using a standard semantic proof based on a well-founded induction on the typing derivation. To show that a circular typing proof is well-founded, we apply the size change principle [35] to size informations extracted from the circular structure of our proofs in a precisely defined way (see Section 3, page 16). 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 use the following type for the usual \textit{map} function, where $\text{List}(A, \alpha)$ denotes the type of lists of size $\alpha$ with elements of type $A$, defined as $\mu_\alpha L.\text{[Nil | Cons(A x L).}]

$$
\forall A.\forall B.\forall \alpha.(A \rightarrow B) \rightarrow \text{List}(A, \alpha) \rightarrow \text{List}(B, \alpha)
$$

Thanks to the quantification on the ordinal $\alpha$, 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 \textit{map} function, without loosing

\(^8\)This particular point will be discussed at the end of the introduction, in relation with related work (see page 9.
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 quick-sort requires showing that partitioning a list of size $\alpha$ produces two lists of size at most $\alpha$. To do so, the partitioning function must be defined with the following type.

$$\forall A, \forall \alpha. \text{List}(A, \alpha) \rightarrow \text{List}(A, \alpha) \times \text{List}(A, \alpha)$$

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

In this paper, the language of the ordinals that can be represented in the syntax is very limited. As in [53], it only contains a constant $\infty$, a successor symbol and variables for quantification. Working with such a small language allows us to keep things simple while still allowing the encoding of many size invariants. Nonetheless, it is clear that the system could be improved by extending the language of ordinals with function symbols such as, for example, maximum or addition.

**Properties of the system.**

A first version of the language without general recursion (i.e., without the fixpoint combinator) is defined in Section 4. It has three main properties: strong normalisation, type safety and logical consistency (Theorems 6.25, 6.27 and 6.24). These results follow from the construction of a realizability model presented in Section 6. They are consequences of the adequacy lemma (Theorem 6.23), which establishes the compatibility of the model with the language and type system.

After the introduction of the fixpoint combinator in Section 7, the properties of the system are mostly preserved (Theorems 7.17 and 7.18). However, the definition of the model needs to be changed slightly as strong normalisation (in the usual sense) is compromised by the fixpoint combinator. Indeed, the reduction rule $Yx.t \succ t[x := Yx.t]$ is obviously non-terminating. Nonetheless, we can still prove normalisation for all the weak reduction strategies (i.e., those that do not reduce under $\lambda$-abstractions or case analyses).

**Implementation.**

Typing and subtyping are likely to be undecidable in our system. Indeed, it contains Mitchell’s variant of System F, known as $\text{F}_\eta$ [13], for which both typing and subtyping are undecidable [60, 62–64]. However, we can not apply directly this result because it is an open problem whether every normalising extensions of system F or $\text{F}_\eta$ are undecidable.

Moreover, we believe that there are no practical, complete semi-algorithms for extensions of System F like ours. Instead, we propose an incomplete semi-algorithm that may fail or even diverge on a typable program. In practice we almost never meet non termination, but even in such an eventuality, the user can interrupt the program to obtain a relevant error message. Indeed, type-checking 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 based on our system. It is called SubML and is available online [36]. Aside from a few subtleties described in Section 10, the implementation is straightforward and remains very close to the typing rules of Figure 12 and to the subtyping rules of Figures 8 and 13. Although the system has a great expressive power, its simplicity allows for a very concise implementation. The main functions (type-checking and

---

9The rules of Figure 7 need to be modified slightly to handle fixpoints.
Practical Subtyping for System F

subtyping) require less than 600 lines of OCaml code. The current implementation, including parsing, evaluation and \LaTeX{} pretty printing contains less than 6500 lines of code.

We conjecture that our implementation is complete (i.e., it may succeed on all typable programs), provided that enough type annotations are given. On practical instances, the required amount of annotations seems to be reasonably small (see Section 5 and 8). Overall, the system provides a similar user experience to statically typed functional languages like OCaml or Haskell. In fact, such languages also require type annotations for advanced features like polymorphic recursion.

SubML provides literate programming features inspired by the PhoX language [48]. They can notably be used to generate \LaTeX{} documents. In particular, the examples presented in Sections 4, 5 and 8 (including proof trees) have been generated using SubML, and are therefore machine checked. Many other program examples (more than 4000 lines of code) are provided with the implementation of SubML. They are listed and some of them are detailed in section 5 and 8. We think they are sufficient to support our claim that the system is indeed usable in practice. SubML can either be installed from its source code or tried online at the following URL.

https://rlepigre.github.io/subml

Applications.

In addition to classical examples, our system allows for applications that we find very interesting (see Sections 5 and 8). As a first example, we can program with the Church encoding of algebraic data types. Although this has little practical interest (if any), it requires the full power of System F and is a good test suite for polymorphism. As Church encoding is known for having a bad time complexity, Dana Scott proposed a better alternative using a combination of polymorphism and inductive types [1]. For instance, the type of natural numbers can be defined as follows.

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

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

In standard systems, recursion on inductive data types requires specific typing rules for recursors, like in Gödel’s System T. In contrast, our system is able to type a recursor for types like \(\mathbb{N}_S\), without having to extend the language. This recursor was shown to the second author by Michel Parigot [43]. We then adapted it to other algebraic data types, showing that Scott encoding can be used to program in a strongly normalisable system with the expected asymptotic complexity.

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

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

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

As far as we known, this is the first strongly normalisable type system that can be used to encode inductive and coinductive data type with the intended asymptotic complexity and without the need for a specific combinator, i.e. using only pure lambda-calculus. Indeed, if recursor in system like Gödel system T are encoded as pure \(\lambda\)-terms, they are not normalisable.

Curry style, 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

\[\text{The online version is compiled to Javascript using js_of_ocaml (https://ocsigen.org/js_of_ocaml/).}\]
types are interpreted as intersections (and existential types as unions) in the semantics. As a consequence, the terms do not include type abstractions and type applications as in Church style, where polymorphic types are interpreted as functions (and existential types as pairs). This means that it is not possible to introduce a name for a type variable in a term, which is necessary for annotating subterms of polymorphic functions with their types.

As our system relies on choice operators for types, it never manipulates type variables. However, we found a way to name choice operators corresponding to local types using a pattern matching syntax. It can be used to extract the definition of choice operators from types and make it available to the user for giving type annotations. As an example, we can fully annotate the polymorphic identity function as follows.

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

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

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

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

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

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

The first choice operator denotes a type $T$ such that $h$ has type $\exists U. \{ f : T \to U; g : U \to T \}$. As our system never infers polymorphic or existential types, we can rely on the names that were chosen by the user for the bound variables. This new approach to abstract types seems simpler than previous work like [14].

Contributions of this paper

- Syntax-directed typing rules for a Curry-style language, meaning that there are no rules for quantifications in typing, these are handled only in subtyping.
- Local subtyping and choice operators, allowing a simple presentation and a clear semantics.
- Circular proofs for subtyping of inductive and coinductive types.
- Absence of an explicit covariance condition for inductive and coinductive types.
- Absence of an explicit semi-continuity condition for recursion.
- Using choice operators to implement ML dot notation for abstract type.
- Showing that dealing with undecidable type systems yield to simpler implementation that works in practice.

This list may seem to long for a single paper but it is required, to justify the last point, to put all these ingredients together in order to have a system which remains reasonably simple, yet powerful enough to match existing implementation of typed functional programming languages.

Related work and other approaches.

The language presented in this paper is an extension of John Mitchell’s System F$_\eta$ [13], which itself extends Jean-Yves Girard and John Reynolds’s System F [24, 52] with subtyping. Unlike previous extensions [6, 45] that deals only with inductive types, our system supports mixed induction and
coinduction with polymorphic and existential types. The latter is usually not so easy to integrate as said in section 6.4 of [15], even so that in this paper the authors do not give an implementation of their type system. Here our treatment of existential is really dual to the treatment of polymorphism.

Comparison with existing algorithm for polymorphic type. Here we focus on the part of our implementation dealing with the type of system F, seeing our type system as a reformulation of Mitchell’s system $F_\eta$. Since the initial work on rank 1 polymorphism by Damas and Milner [16] quite a few results have been obtained for arbitrary rank polymorphism. Mainly two approaches have been considered, the approach of MLF initiated in [34], that is intermediate between Church style and Curry style as it does not have a type application, but keeps a syntax for introducing type abstraction in terms. However, for fully Curry style, less work exists. We can cite the work of Didier Rémy and Julien Cretin [50] which requires type annotation for argument of application and let definitions, the work of Simon Peyton Jones et al. [44] and the work of Dunfield and Krishnaswami [20].

All these works yield to a rather more complicated algorithm than our implementation. The main reason of complexity is that they provide an algorithm (i.e. a terminating type checking procedure). In our work we give a type system which is not directly an algorithm, but which can be implemented by introducing unification variables for unknown types.

We chose to trade the decidability of type-checking for simplicity. Indeed, we chose not to look for (and prove) a decidability result for a restricted system, unlike most work on programming languages. We believe that implementing directly undecidable system is natural because the theoretical complexity of usual (decidable) type system is in general exponential or more while this theoretical bound is never meet in practical applications. Hence it was worth checking if undecidable type system could not be used directly with a well designed semi-algorithm, which our experiments showed perfectly acceptable. In particular, the user experience is not different from working with meta-variables or implicit arguments as in Coq or Agda [39, 42].

Moreover, our current implementation is in fact the most trivial one,

- directly following the rules,
- introducing unification variables for unknown types and
- solving all subtyping judgement of the form $?U \subset A$ of $A \subset ?U$ where $U$ is a unification variable by setting $?U := A$!

We show that even if it is in fact not terminating in theory, it works very well in practice, and does not require more annotation than the above mentioned solution. All the less trivial heuristics to deal with ordinals and circular proofs are unnecessary when considering only the fragment without inductive and coinductive types, yielding really a very simple algorithm.

The main point to achieve this was to remove the $\forall_i$ rule from the typing rule which seem really new, even if this is not a goal as such. One of the main advantages, is that it easily extended to all features, even some that are not described here, like some form of dependant types used in PML [49].

Comparison with existing algorithm for sized (co)inductive types. Our type system uses sized types [28] as our inductive and coinductive types carry ordinal numbers. Such a technique is widespread for handling induction [10, 11, 26, 31, 54] and even coinduction [4, 5, 53], in settings where termination is required.

The most important difference between this paper and previous works precisely lies in the handling of recursion. In particular, previous work rely on specific rules for checking size relations between ordinal parameters when using recursion such as:
In these rules $F$ must be contravariant in its parameter for the $\mu$ rule and covariant for the other. But this is not enough and a condition of semi-continuity is also required. The type $F(X) = X \to B$ or $F(X) = B \to X$ when $X$ is not free in $B$ are accepted, but $F(X) = (X \to B) \to B$ and $F(X) = ((X \to B) \to B) \to B$ are not.

In this paper, inductive and coinductive types are handled in a way that is completely orthogonal to recursion. Ordinal sizes are only manipulated in the subtyping rules related to inductive and coinductive types, while recursion is handled separately using a simple typing rule that just unfold the fixpoint combinator.

This leads to a system that has a rather simple presentation compared to previous work, relying on a general concept of circular proofs. The semi-continuity on the type $F$ is replaced by an implicit condition on the term $t$. We have some examples that would not be accepted by previous work because of this reason.

Another difference is that [5, 53] describe a system with one simple rule for recursion, while the implementation in Agda [42], MiniAgda or Coq use something more general dealing with multiple arguments and mutually recursive definitions and using a termination criterion such as the size change principle of Lee, Jones and Ben-Amram [35]. This more general recursion is rarely formally described, with some exceptions like [3, 29, 30], but then we think it is more complex than our work.

Our general framework for circular proofs makes it easier to integrate the termination checker in the formal description of the system and its normalisation proof and was for instance easy to reuse for PML [49].

Nevertheless, this is not completely satisfactory, and we would like to improve it 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 [30]. This could hopefully be adapted to our setting.

**Comparison with other work using circular proofs.** Here we will only consider system with well founded circular proofs, that is circular proof needing an external validation test to be correct. There exists work studying logical system with unrestricted circular proofs that are not really related to our work. Circular proof in this sense where first studied in the context of temporal logic as in [55–59]. It has also be applied to first-order logic [12].

However, using circular proof to define subtyping seems completely new, even it the deduction rule for subtyping share some similarities with the rule of the modal $\mu$-calculus.

For program termination, we can find some system with circular proof which enjoy a cut elimination procedure which in some sense is a form of termination [21, 22]. We should also mention the work of David Baelde, Amina Doumane and Alexis Saurin [7], which uses circular proof together with sized types in the context of linear logic. But as we said introducing circularity by a rule just unfolding the fixpoint seems new.

**Comparison with other work using Hilbert’s epsilon and tau.** The use of Hilbert’s epsilon and tau [27] (which can encode each other in classical logic) are a standard alternative to Skolemisation to eliminate quantifiers. They are used in theorem proving. For instance in [18] the author show how to use them at a reduced cost. It is also useful in linguistic [51, 61] which links it to $\lambda$-calculus via Montage approach of semantics. It has been proposed to use it in type theories [9] and is standard in HOL or PVS and often introduced in some Coq proofs.
However it is rarely used in typed programming languages. We found only one paper by Abadi, Gonthier and Werner [2] while this seems a very natural and simple approach.

However, we are not aware of any work that uses it for all kind of variables. It is only used for types in [2]. This allows to use only closed terms and types, which implies that renaming of bound variables is never needed. This would simplify a formalisation of our system in a proof assistant. It also allows for syntactic sugar relying on the name of bound variables like the dot notation. This possibility was unnoticed in [2].

Comparison with other work on subtyping. Subtyping has been extensively studied in the context of ML-like languages, starting with the work of Roberto Amadio and Luca 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 [46]. However, we are not aware of any work on subtyping that leads to a system as expressive as ours for a Curry-style extension of System F. In particular, no other system seems to be able to handle the permutation of quantifiers with other connectives as well as mixed inductive and coinductive types (see Sections 4 and 5).

Approaches without epsilons and local subtyping. First, the approach used in [44], with the following \( \forall \) rule does not work here.

\[
\frac{\Gamma \vdash A \subseteq C \vdash \ldots \vdash C_n \rightarrow B \qquad X \notin A, C_1, \ldots, C_n}{\Gamma \vdash A \subseteq C_1 \rightarrow \ldots \rightarrow C_n 
\rightarrow \nu X.B}
\]

Although it allows for the derivation of Mitchell’s containment axiom, it cannot be used to derive the \( \forall \) rule because of the constraint on \( A \). This is similar to the approach in [50], with the same comments. In this work, formulas are transformed in prenex form to get Mitchell’s containment axiom, but a constraint remains that prevent to derive the \( \forall \) rule.

However, it is actually possible to use typing contexts in subtyping rules, in such a way that Mitchell’s containment axiom and the \( \forall \) rule are derivable. Such a context could simply be a list of types. For instance, the following two rules rely on this principle.

\[
\frac{\Gamma \vdash A \subseteq B \qquad X \notin \Gamma}{\Gamma \vdash A \subseteq \forall X.B}
\]

\[
\frac{\Gamma \vdash C \subseteq A \qquad \Gamma \vdash B \subseteq D}{\Gamma \vdash A \rightarrow B \subseteq C \rightarrow D}
\]

A system relying on such contexts was used in an unpublished note of the second author in 1999 [47].

Unfortunately, \( \Gamma \vdash A \subseteq B \) has an intricate semantics which would be that for any term \( t \) such that \( \Gamma \vdash t : A \), we have \( \Gamma \vdash t : B \) which would in turn need substitution to interpret the judgement \( \Gamma \vdash t : A \) (see below).

Then a natural alternative would be to use a form of local subtyping without choice operators, which would lead to the following rules.

\[
\frac{\Gamma \vdash t \in A \subseteq B \qquad X \notin \Gamma}{\Gamma \vdash t \in A \subseteq \forall X.B}
\]

\[
\frac{\Gamma, x : C \vdash x \in C \subseteq A \qquad \Gamma, x : C \vdash t \in B \subseteq D}{\Gamma \vdash t \in A \rightarrow B \subseteq C \rightarrow D}
\]

However, if we use this approach, the adequacy lemma below requires working with substitutions and valuations, which is a pain.

**Lemma 1.1.** If \( \Gamma \vdash t \in A \subseteq B \) and for all type variable interpretations \( \phi \) and all substitutions \( \sigma \in [\Gamma]_\phi \) we have \( t\sigma \in [A]_{\phi} \), then for all type variable interpretation \( \phi \) and all substitution \( \sigma \in [\Gamma]_\phi \) we have \( t\sigma \in [B]_{\phi} \). Where \( \sigma \in [\Gamma]_\phi \) means \( \forall x \in \text{dom}(\Gamma), \sigma(x) \in [\Gamma(x)]_\phi \).

On the contrary, local subtyping used in conjunction with choice operators have a clear and elegant semantics: \([t] \in [A]\) implies \([t] \in [B]\). It does not impose any syntactic overhead, apart from the choice operators themselves.
However, if we want to study syntactic properties like preservation of typing for $\beta\eta$ reduction (which is established in [47] for a fragment using only system F types), the first approach using context of the form $\Gamma \vdash A \subseteq B$ yields better proofs. In a work in progress, we consider proving preservation of typing for $\beta\eta$ reduction for a fragment with both $\forall$ and $\exists$ going through the equivalence of the three presentations.

Contents.

(1) Introduction ................................................................. 1
(2) Syntactic ordinals and size change matrices ........................................ 13
   Definition of the syntactic ordinal and size change matrices used for sized types.
(3) Circular proofs and size change principle ........................................ 16
   General framework for circular proofs using the size change principle.
(4) Language and type system ................................................... 22
   Definition of a strongly normalisable type system without general recursion.
(5) Fixpoint-less recursion for Scott encoding ..................................... 26
   Examples using (co-)inductive types that usually require specific recursion rules.
(6) Realizability semantics ...................................................... 29
   Semantics and proof of the safety and strong normalisation of the first system.
(7) Fixpoint and termination ..................................................... 38
   Extension with general recursion, replacing strong normalisation by termination.
(8) Terminating examples ....................................................... 44
   Some examples using general recursions.
(9) Type-checking Algorithm .................................................... 49
   Consideration about the implementation.
(10) Type annotations and dot notation .......................................... 52
    Using choice operators to implement the dot notation for abstract types in modules.
(11) Perspectives and Future Work .............................................. 53
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 [28]. 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 $(e_1, \ldots, e_n)$ 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 $E[\overline{x} \leftarrow \overline{e}]$).

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

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

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

Syntactic ordinals are built using the constant $\infty$, a successor symbol, and ordinal choice operators (or witnesses) of the form $\varepsilon_{\overline{\tau} < \overline{\nu}} P(\overline{\alpha}, \overline{\kappa})_i$. Intuitively, the vector $\overline{\tau}$ defined as $\tau_i = \varepsilon_{\overline{\tau} < \overline{\nu}} P(\overline{\alpha}, \overline{\kappa})_i$ denotes syntactic ordinals that are pointwise smaller than $\overline{\nu}$, and such that "$P(\overline{\alpha}, \overline{\kappa})$ is true" (this will be made formal in Definition 2.6). In the upper bound $\overline{\nu}$, one can use the notation $\Omega$ to mean that there is no size constraint on the corresponding variable. The symbol $\Omega$ can not appear anywhere else. In other words, $\Omega$ 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 $\varepsilon_{\overline{\tau} < \overline{\nu}} P(\overline{\alpha}, \overline{\kappa})$ for the vector $(\varepsilon_{\overline{\tau} < \overline{\nu}} P(\overline{\alpha}, \overline{\kappa}))_1 \leq i \leq |\overline{\tau}|$. In the case where $|\overline{\alpha}| = |\overline{\nu}| = 1$ we will write $\varepsilon_{\tau < \nu} P(\alpha, \kappa)$ for both $\varepsilon_{\overline{\tau} < \overline{\nu}} P(\overline{\alpha}, \overline{\kappa})$ and $\varepsilon_{\overline{\tau} < \overline{\nu}} P(\overline{\alpha}, \overline{\kappa})_1$.

In the semantics, the symbol $\infty$ will be interpreted by an (actual) ordinal $\Omega$. 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, $\Omega$ will eventually be instantiated with an ordinal large enough to ensure the convergence of all the fixpoints corresponding to our inductive and coinductive types (see Section 6). Nonetheless, $\Omega$ 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 $\Omega$ be a fixed, but arbitrary ordinal. We denote $[O]$ the ordinal $\Omega + \omega$, which is also the set of all the (actual) ordinals of our semantics, and the interpretation of $O$.

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

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

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

$^{11}$In practice, ordinals such as $\infty + 1$ and further successors of $\infty$ will never be considered.
We will now give a semantical interpretation to (closed) parametric syntactic ordinals, using (actual) ordinals of $\llbracket O \rrbracket$. 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) $\llbracket - \rrbracket$ such that $\llbracket P \rrbracket \in \llbracket O \rrbracket | P | \to \{0, 1\}$ for all $P \in P$. The semantics of closed (vectors of) parametric syntactic ordinals is then defined inductively as follows.

\[
\begin{align*}
\llbracket \infty \rrbracket &= \Omega \quad \llbracket O \rrbracket = \Omega + \omega \quad \llbracket \kappa + 1 \rrbracket = \llbracket \kappa \rrbracket + 1 \\
\llbracket 0 \rrbracket &= o \quad \llbracket \overline{\kappa} \rrbracket = (\llbracket \kappa_1 \rrbracket, \ldots, \llbracket \kappa_n \rrbracket)
\end{align*}
\]

\[
\llbracket \overline{\pi < \omega P(\overline{\alpha}.\overline{\kappa})} \rrbracket = \begin{cases} 
\overline{\sigma} \in \llbracket O \rrbracket | P | \text{ such that } \overline{\sigma} < \llbracket \overline{\omega} \rrbracket \text{ and } \llbracket P(\overline{\sigma}.\overline{\kappa}) \rrbracket = 1 \text{ if it exists,} \\
\overline{0} \text{ otherwise.}
\end{cases}
\]

In the ordinal witness case, $\langle \rangle$ denotes the pointwise ordering on vectors of ordinals, and $\overline{0}$ is a vector of 0 ordinals of size $|\overline{\alpha}|$. Moreover, as there may be several possible values for $\overline{\sigma}$, we will sometimes distinguish particular models for which the choice is made in a specific way. If $M_1$ is such a model, we will use the notation $\llbracket \kappa \rrbracket | M_1$ for the corresponding interpretation. When no particular model is specified, it is considered fixed, but arbitrary.

**Lemma 2.7.** Let $M_0$ be a model, $\overline{\pi \in \omega P(\overline{\alpha}.\overline{\kappa})}$ be a vector of ordinal witnesses, and $\overline{\sigma}$ be a vector of (actual) ordinals of the corresponding size. If we have $\overline{\sigma} \in \llbracket \overline{\omega} \rrbracket | M_0$ and $\llbracket P(\overline{\sigma}.\overline{\kappa}) | M_0 \rrbracket = 1$, then there is a model $M_1$ such that $\llbracket \overline{\omega} \rrbracket | M_1 = \llbracket \overline{\omega} \rrbracket | M_0$, $\llbracket \overline{\kappa} \rrbracket | M_1 = \llbracket \overline{\kappa} \rrbracket | M_0$, and $\llbracket \overline{\pi < \omega P(\overline{\alpha}.\overline{\kappa})} \rrbracket | M_1 = \overline{\sigma}$.

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

\[
\begin{align*}
\llbracket \infty \rrbracket &= \Omega = \llbracket O \rrbracket = \llbracket 0 \rrbracket = 0 \\
\llbracket \kappa, \ldots, \kappa_n \rrbracket &= \max(h(\kappa_1), \ldots, h(\kappa_n)) \\
\llbracket \kappa + 1 \rrbracket &= 1 + h(\kappa)
\end{align*}
\]

We then construct the interpretation of the parametric syntactic ordinal $\overline{\tau}$ in the model $M_1$ by induction on $h(\tau)$. We take $\llbracket \overline{\tau} \rrbracket | M_1 = \llbracket \overline{\tau} \rrbracket | M_0$ for every $\overline{\tau}$ such that $h(\tau) < h(\overline{\pi < \omega P(\overline{\alpha}.\overline{\kappa})})$, which includes the elements of $\overline{\omega}$ and $\overline{\kappa}$. We then take $\llbracket \overline{\pi < \omega P(\overline{\alpha}.\overline{\kappa})} \rrbracket | M_1 = \overline{\sigma}$ and complete the definition by marking arbitrary choices for the remaining ordinal witnesses. \qed

In the syntax, we will compare syntactic ordinals $\overline{\tau}$ and $\overline{\kappa}$ using an ordering relation $\overline{\kappa} \leq \overline{\tau}$, and a strict ordering relation $\overline{\kappa} < \overline{\tau}$. They will be defined in terms of a third (ternary) relation $\overline{\kappa} \leq i \overline{\tau}$ 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.

\[
y, \delta ::= \emptyset \mid y, \kappa
\]

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

**Definition 2.9.** For all $i \in \mathbb{Z}$, the relation $(\leq i)$ on syntactic ordinals is defined, under an ordinal context $y$, using the deduction rules of Figure 1. We then take $\kappa \leq 0 \tau$ as the definition of $\kappa \leq \tau$, and $\kappa \leq 1 \tau$ as the definition of $\kappa < \tau$. Formally, we will also need the relation $\kappa \leq i O$, which will be always true derivable if $\kappa$ is a syntactic ordinal.

Intuitively, the relation $\kappa \leq i \tau$ can be understood as "$\kappa + i \leq \tau$" when $i \geq 0$, and as "$\kappa \leq \tau + (\neg i)$" when $i \leq 0$. Note that the deduction rule system of Figure 1 can be implemented as a deterministic
and terminating procedure. Indeed, it is easy to see that the \((s_r)\) rule commutes with the \((s_l)\), \((e)\) and \((\epsilon_w)\) rules. When the rules \((e)\) and \((\epsilon_w)\) both apply, it is always better to use \((e)\) since it yields a lower index, and thus proves more judgements according to the second item of Lemma 2.10.

**Lemma 2.10.** For every ordinal contexts \(\gamma\) and \(\delta\), for every syntactic ordinals \(\kappa_1, \kappa_2\) and \(\kappa_3\), and for every integers \(i\) and \(j\) we have:

1. if \(\gamma \vdash \kappa_1 \leq_i \kappa_2\) then \(\gamma, \delta \vdash \kappa_1 \leq_i \kappa_2\),
2. if \(\gamma \vdash \kappa_1 \leq_i \kappa_2\) and \(j \leq i\) then \(\gamma \vdash \kappa_1 \leq_j \kappa_2\),
3. if \(\gamma \vdash \kappa_1 \leq_i \kappa_2\) and \(\gamma \vdash \kappa_2 \leq_j \kappa_3\) then \(\gamma \vdash \kappa_1 \leq_{i+j} \kappa_3\).

**Proof.** The proofs of (1) and (2) are immediate by induction on the derivation. We prove (3) by induction on the sum of the sizes of the derivations of \(\gamma \vdash \kappa_1 \leq_i \kappa_2\) and \(\gamma \vdash \kappa_2 \leq_j \kappa_3\). If the last applied rule on either side is \(=\), then we have \(\kappa_1 = \kappa_2\) and \(i \leq 0\) or \(\kappa_2 = \kappa_3\) and \(j \leq 0\). In both case we can conclude using (2). If the last rule used on the left is \((s_l)\) then \(\kappa_1 = \kappa + 1\). By induction hypothesis we have \(\gamma \vdash \kappa \leq_{i+j+1} \kappa_3\) and thus \(\gamma \vdash \kappa_1 \leq_{i+j} \kappa_3\). A similar argument can be used if the last rule used on the right is \((s_r)\). If the last used rule on the left is \((e)\) or \((\epsilon_w)\) then we have \(\kappa_1 = \epsilon_{\pi < \mathcal{P}(\bar{a}, \bar{b})}\) if \(\kappa_2 \leq \kappa\). By induction hypothesis, we get \(\gamma \vdash w_m \leq_{i+j-1} \kappa_3\) if we applied the \((e)\) rule or \(\gamma \vdash w_m \leq_{i+j} \kappa_3\) if we applied the \((\epsilon_w)\) rule. In both cases this implies \(\gamma \vdash \kappa_1 \leq_{i+j} \kappa_3\). If the last rule used on the right is \((e)\) or \((\epsilon_w)\) then we must be in one of the previous cases. Indeed, the rules that can be applied on the left when \(\kappa_2\) is an ordinal witness are \(=\), \((s_l)\), \((e)\) and \((\epsilon_w)\). The only missing case (otherwise another rule already treated appears) is when the rule \((s_r)\) is applied on the left and the rule \((s_l)\) is applied on the right. Thus, we have \(\kappa_2 = \kappa + 1\) and the induction hypothesis gives \(\gamma \vdash \kappa_1 \leq_{i+j} \kappa_3\) directly.

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

**Lemma 2.11.** Let \(\gamma\) be a closed ordinal context, \(\kappa_1\) and \(\kappa_2\) be closed syntactic ordinals, and \(i\) be an integer such that \(\gamma \vdash \kappa_1 \leq_i \kappa_2\) is derivable. If \([\tau]\) \(\neq 0\) for all \(\tau \in \gamma\), then \([k_1] + i \leq [k_2]\) when \(i \geq 0\) and \([k_1] \leq [k_2] + (-i)\) when \(i \leq 0\).

**Proof.** The proof is done by induction on the derivation of \(\gamma \vdash \kappa_1 \leq_i \kappa_2\). The cases for the \(=\), \((s_l)\) and \((s_r)\) rules are immediate. In the case of the \((e)\) rule, we have \(\kappa_1 = \epsilon_{\pi < \mathcal{P}(\bar{a}, \bar{b})}\). As a consequence, \([k_1]\) is either equal to some ordinal \(\kappa < [w_j]\) or to 0. Since \([w_j] \neq 0\), we have \([k_1] < [w_j]\) in both cases and we can thus conclude by induction hypothesis. The proof is similar in the case of the \((\epsilon_w)\) rule, but it is possible that \([w_j] = 0\) so we only have \([k_1] \leq [w_j]\).

We are now going to consider the formalism that will be used to relate our syntactic ordinals to the size-change principle [35] 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.
We will now introduce an abstract notion of circular proof, with a related notion of well-foundedness. This can be seen as a complete redefinition of the size change principle [35] to define a general notion of well-founded circular proof instead of a criterion for program termination. We could not find a way to reuse directly the result in [35] and as a consequence do not rely on any result in this paper.

The idea is to represent proofs as directed acyclic graphs, and to label their edges with size-change matrices. We will thus track the evolution of the size of our syntactic ordinals throughout proofs, and rely on same condition as in the size change principle to ensure that there is some structural decrease. Of course, a proof will only be considered correct if we can establish such a decrease. In

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 \(\infty\) and \(0\), and the absorbing element of \((\circ)\) is \(\infty\).

\[
0 \circ 0 = 0 \quad x \circ \infty = \infty \circ x = \infty \quad -1 \circ x = x \circ -1 = -1 \quad \text{if} \quad x \neq \infty
\]

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 \(\infty\) 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. \(\square\)

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 \(O\), equipped with the relations \((\leq)\) and \((<)\) given in Definition 2.9.

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 \(z_j \leq y_k \leq x_i\). \(\square\)

3 CIRCULAR PROOFS AND SIZE CHANGE PRINCIPLE

We will now introduce an abstract notion of circular proof, with a related notion of well-foundedness. This can be seen as a complete redefinition of the size change principle [35] to define a general notion of well-founded circular proof instead of a criterion for program termination. We could not find a way to reuse directly the result in [35] and as a consequence do not rely on any result in this paper.

The idea is to represent proofs as directed acyclic graphs, and to label their edges with size-change matrices. We will thus track the evolution of the size of our syntactic ordinals throughout proofs, and rely on same condition as in the size change principle to ensure that there is some structural decrease. Of course, a proof will only be considered correct if we can establish such a decrease. In

ACM Transactions on Programming Languages and Systems, Vol. 1, No. 1, Article 1. Publication date: July 2017.
Section 4, we will rely on circular subtyping proofs to handle inductive and coinductive types. We will then use circular typing proofs to ensure the termination of recursive programs in Section 7.

Our circular proof framework is parameterised by so-called abstract judgements, their deduction rules and their semantics. They will for example correspond to typing judgements or to local subtyping judgements, with their respective deduction rules and interpretations. We believe that our framework could be applied to other deduction systems involving a notion of size.

**Definition 3.1.** A language of abstract judgements is given by a set $\mathcal{J}$ of symbols, representing judgements parameterised by syntactic ordinals. Every symbol $J \in \mathcal{J}$ 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 Section 2. For instance, we will work with and manipulate syntactic ordinals of the form $\varepsilon_\pi < x \gamma J(\alpha, \nu)$.

**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 using $\mathcal{P}$.

**Example 3.3.** We will use abstract judgements of the form $t \in A \subset B$ and $t : A$, and therefore predicates $\neg (t \in A \subset B)$ and $\neg (t : A)$. Hence, the corresponding choice operators will be $\varepsilon_\pi < \gamma (\neg (t \in A \subset B))$ and $\varepsilon_\pi < \gamma (\neg (t : A))$. Moreover, when a subtyping judgement $t \in A \subset B$ will be used as an abstract judgement, we will always have $t = \varepsilon_{x \in A}(x \notin B)$ meaning the we are encoding a usual subtyping judgement. Therefore, we will abbreviate the above choice operators as $\varepsilon_\pi < \gamma (A \subset B)$ and $\varepsilon_\pi < \gamma (t \notin A)$.

We will now consider the semantical interpretation of abstract judgements. Intuitively, our abstract judgements will be interpreted as predicates over (actual) ordinals.

**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 $[J] : [O]^n \rightarrow \{0, 1\}$. The predicates over ordinals built according to the previous definition are then interpreted as $\neg [J] = \bar{\sigma} \mapsto 1 - [J](\bar{\sigma})$.

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

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 index, a conjunction of strict relations between the ordinals of a given vector.

**Definition 3.6.** A list of ordinal constraints $C$ of arity $n$ is given by a function $C$ from $\{1,\ldots,n\}$ to $\{0,1,\ldots,n\}$. Given a vector of ordinals $\bar{\sigma} \in [O]^n$, we denote $C(\bar{\sigma})$ the vector of size $n$ defined as $C(\bar{\sigma})_i = [O]_i$ if $C(i) = 1$ and as $C(\bar{\sigma})_i = o_j$ if $C(i) = j \neq 0$. We say that $C$ is satisfied by $\bar{\sigma} \in [O]^n$ when $o_i < C(\bar{\sigma})_i$ for all $1 \leq i \leq n$.

Building circular proofs will require the generalisation of abstract sequents. In other words, we will sometimes need to prove that an abstract sequent is 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 which syntactic ordinals have been quantified over. It is thus of the form $\forall \bar{\sigma}(\gamma \vdash C(\bar{\sigma}) \Rightarrow J(\bar{\sigma}, \bar{\nu}))$, where $\gamma$ is an ordinal context only containing variables of $\bar{\sigma}$, $C$ is a list of ordinal constraints of arity $|\bar{\sigma}|$, $J$ is an abstract judgement
∀α(γ ⊢ C(α) ⇒ J(α))  γ[α := κ], δ ⊢ κ < C(κ) 1 ≤ i ≤ |α|  \text{G}

∀α(γ ⊢ C(α) ⇒ J(α))\]_k

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

∀(γ ⊢ (λ) ⇒ J(κ))\]_0  \emptyset  \text{G}

∀(γ ⊢ (λ) ⇒ J(κ))\]_0  \emptyset  \text{G}

Fig. 3. Clearly invalid (non-well-founded) circular proof built using a direct loop.

and \(\vec{\nu}\) is a vector of ordinals such that \(|\vec{\nu}| + |\vec{\alpha}| = |J|\). We say that the general abstract sequent \(\forall\alpha(γ ⊢ C(\alpha) ⇒ J(\alpha, \vec{\nu}))\) is valid if \([J](\vec{\alpha}, [\vec{\nu}]) = 1\) for all \(\vec{o} ∈ [O]^n\) such that \(o_i \neq 0\) whenever \(α_i ∈ γ\), and such that \(C\) is satisfied by \(\vec{\alpha}\).

Note that in a general abstract sequent, a judgement \(J\) may use ordinals \(\vec{\nu}\) that are not quantified over. In the following, we will often omit to mention them explicitly. In particular, our definition implies that the ordinal context \(γ\) and the ordinals of \(C(\alpha)\) cannot use ordinals of \(\vec{\nu}\) (therefore, they can only use ordinals of \(\alpha\)). This restriction is not essential, but simplifies definitions.

**Definition 3.8.** A circular deduction system is given by a set of deduction rules which 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 second premise). The induction rule \((I_k)\) may be used to prove a general abstract sequent using itself as an hypothesis (this is the meaning of the square brackets). Note that a natural number \(k\) (unique in a proof) is used to keep track of the originating induction rule.

**Example 3.9.** Particular instances of the \(G\) and \(I_k\) rule can be found in figure 10 page 27.

Remark 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 the proofs in a circular deduction system. In particular, it allows for clearly invalid proofs such as the following.

**Example 3.10.** The circular proof given in Figure 3 can be used to prove any abstract sequent \(γ ⊢ J(κ)\). After generalising over the empty vector of ordinals () , the new hypothesis is directly applied. Hence, there is obviously no size decrease along the loop.

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 [35] 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 3.11.** Given a proof $\Pi$ expressed in a circular proof system, a block is a subproof $B$ of $\Pi$ such that its conclusion is either the conclusion of $\Pi$ 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 $\Pi$, and it is the size of the quantified vector of ordinals $\bar{\alpha}$ in the conclusion of $B$ otherwise.

**Definition 3.12.** Let $\Pi$ be a proof expressed in a circular proof system. The call graph of $\Pi$ is the graph induced by the block structure of $\Pi$. Its vertices are the blocks of $\Pi$, and every block $B_1$ has one outgoing edge for each of its premises. Such a premise is either directly proved by a block $B_2$ or it is an hypothesis introduced by an $I_k$ rule at the beginning of a block $B_2$. In both cases, there is an edge from $B_1$ to $B_2$. Note that we may have $B_1 = B_2$.

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 4, if we only include the premises involved in the definition of the edge $(B_1, B_2)$. The $[\alpha] \times [\beta]$ matrix $M$ attached to the edge $(B_1, B_2)$ is then defined as $M_{i,j} = -1$ when $\delta[\bar{\beta} := \bar{\tau}], \delta' + \tau_j < \kappa_1$ is derivable, $M_{i,j} = 0$ when only $\delta[\bar{\beta} := \bar{\tau}], \delta' + \tau_j \leq \kappa_1$ 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 hence be computed in finite time by composing edges until saturation.

**Definition 3.13.** We say that a proof is well-founded if every idempotent loop in the transitive closure of its call graph (i.e.; closed path with label $M$ such that $MM = M$) has at least one $-1$ on the diagonal of its label.

Note that such loops are necessarily labelled with square matrices.

To illustrate the previous definitions, we will now consider two examples of circular proofs. The structure of the former has the same structure as the well-founded circular subtyping proof of Figure 10, page 27. The latter corresponds to the invalid circular proof of Example 3.10.
Example 3.14. We now consider the example of circular proof given in the upper part of Figure 5. For simplicity, the ordinals are not given explicitly. We will however assume that (besides reflexivity) it is possible to derive \( \kappa_2 \vdash \nu_2 < \kappa_2 \) and \( \tau_1 \vdash \nu_3 < \tau_1 \). The proof can be decomposed into three blocks \( B_0, B_1 \) and \( B_2 \). The corresponding call graph is given in the lower part of Figure 5. Its transitive closure contains five idempotent loops. There are none on the block \( B_0 \), two on the block \( B_1 \) with labels \( (0 \infty) \) and \( (\infty -1) \), and three on block \( B_2 \) with labels \( (\infty 0) \), \( (\infty -1) \) and \( (0 \infty) \). 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 3.15. The circular proof of Example 3.10, displayed in Figure 3, is built of two blocks, one of the block containing only the conclusion and the other 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 3.16. The two deduction rules of Figure 2 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 \([\kappa]\) that are mapped to variables of \(\gamma\) are non-zero. Moreover, the validity of the right premises tells us that the list of ordinal constraints \(C\) is satisfied by \([\kappa]\). We can thus conclude using the validity of the first premise.

For the \((I_\kappa)\) rule, we consider the semantics of the choice operators over ordinals 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 \((I_\kappa)\) rule does not involve the bracketed sequent (i.e., the new hypothesis) it introduces. The justification for such hypotheses is handled globally by our notion of well-founded proof (Definition 3.13).

Theorem 3.17. Let us consider a circular deduction system, which 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 \(M\) such that the ordinals of \(J\) are satisfied by \(M\). This establishes that the premise of our \((I_\kappa)\) rule is a false abstract sequent (i.e., the new hypothesis) it introduces. The justification for such hypotheses is handled globally by our notion of well-founded proof (Definition 3.13).

By construction, we know that the general abstract sequent \(\forall\alpha(\gamma + C(\alpha) \Rightarrow J(\alpha))\) is false in the model \(M_i\), and that \(\overline{\alpha_i}\) is a counterexample. This means that \([\gamma[\alpha := \overline{\alpha_i}]])^{M_i}\) contains only positive ordinals, \(C\) is satisfied by \(\overline{\alpha_i}\) and \([J](\overline{\alpha_i}) = 0\). Thus, using Lemma 2.7 we can define \(M_{i+1}\) to be a model such that \([\kappa]\)^{M_{i+1}} = \(\overline{\alpha_i}\). This establishes that the premise of our \((I_\kappa)\) rule is a false abstract sequent in the model \(M_{i+1}\).

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

\[
\frac{\forall\beta(\delta + D(\beta) \Rightarrow K(\beta)) \quad (\delta[\beta := \tau], \delta' + \tau_1 < D(\tau_1))_{1 \leq l \leq |\beta|} \quad G}{\delta[\beta := \tau], \delta' + K(\tau)}
\]
As the conclusion of this rule is false, the model $M_{i+1}$, we know that $[\delta[\overline{\beta}] := \overline{\tau}], \delta^n M_{i+1}$ only contains positive ordinals and that $[K][\overline{\tau}] M_{i+1}$ only contains positive ordinals and that $[K][\overline{\tau}] M_{i+1}$ only contains positive ordinals and that $[K][\overline{\tau}] M_{i+1}$ only contains positive ordinals and that $[K][\overline{\tau}] M_{i+1}$ only contains positive ordinals and that $[K][\overline{\tau}] M_{i+1}$ only contains positive ordinals and that $[K][\overline{\tau}] M_{i+1}$ only contains positive ordinals and that $[K][\overline{\tau}] M_{i+1}$ only contains positive ordinals and that $[K][\overline{\tau}] M_{i+1}$ only contains positive ordinals and that $[K][\overline{\tau}] M_{i+1}$ only contains positive ordinals and that $[K][\overline{\tau}] M_{i+1}$ only contains positive ordinals and that $[K][\overline{\tau}] M_{i+1}$ only contains positive ordinals. According to Lemma 2.11, the right premises of the G rule cannot be false. Hence, it must be that $\forall \overline{\theta}(\delta + D(\overline{\beta}) \Rightarrow K(\overline{\beta}))$ is false in the model $M_{i+1}$. Therefore, we can define $B_{i+1}$ to such that $(B_i, B_{i+1})$ is the edge corresponding to that premise and $\overline{o}_{i+1}$ to be $[\overline{\tau}] M_{i+1}$, which is indeed a counterexample for this sequent.

By definition, there is an edge linking the block $B_i$ to the block $B_{i+1}$ in the call-graph. It is labelled with a matrix $M_i$ and we will show $o_{i+1} < M_i \overline{o}_i$ to conclude the construction of our sequence. Let us take $1 \leq m \leq |\overline{o}_i|$ and $1 \leq n \leq |\overline{o}_{i+1}|$ and consider $(M_{i})_{m,n}$. If it is equal to $-1$ then there is a proof of $\delta[\overline{\beta}] := \overline{\tau}]$, $\delta^n < \tau_n < \kappa_m$ and hence proposition 2.11 gives us $[\tau_n] M_{i+1} < [\kappa_m] M_{i+1}$. We can hence conclude that $o_{i+1,n} < o_{i,m}$ since we have $[\kappa_m] M_{i+1} = o_{i,m}$ and $o_{i+1,n} = [\tau_n] M_{i+1}$ by definition of $M_{i+1}$ and $\overline{o}_{i+1}$ respectively. If $(M_{i})_{m,n}$ is equal to $0$ then similar reasoning yields $o_{i+1,n} \leq o_{i,m}$, and if it is $\infty$ then there is nothing to prove.

To conclude, we will now use the same argument as in the proof of Theorem 4 in [35] and use Ramsey theorem for pairs below:

**Theorem 3.18 (Ramsey for pairs).** Consider a finite set $X$ and a mapping $f$ from $\{(i, j) \in \mathbb{N} ; i < j\}$ to $X$. Then, there exists an infinite set $Y \subset \mathbb{N}$ such that $f$ restricted to $\{(i, j) \in Y ; i < j\}$ is constant. 

For all $0 \leq i < j$, we define $M_{i,j}$ to be the matrix $M_{1,i} \ldots M_{j-1}$. 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 and to the definition of $M_{i,j}$, this implies that $MM = M_{u_i,u_j}M_{u_i,u_j} = M_{u_i,u_j} = M$.

Finally, we can use Lemma 2.16 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, $o_{u_{i+1}} < M o_{u_i}$ implies that $o_{u_{i+1}} < o_{u_i}$ 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 and is shown strongly normalising in Section 6. Surprisingly, recursion is still possible (for specific inductive data types) using $\lambda$-calculus recursors that are typable thanks to subtyping (see Section 5). The language is formed using three syntactic entities: terms, types and syntactic ordinals (see Section 2). Syntactic ordinals are used to annotate types with a size information that is used to show the well-foundedness of subtyping proofs. They are only introduced internally and they are not accessible to the user in the implementation. However, we will see in Section 7 that the type system can be naturally extended to allow the user to express size invariants using ordinals. Although the system is Curry-style (or implicitly typed), terms, types and ordinals are defined mutually inductively due the choice operators that are contained 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 $\lambda$-variables and propositional variables respectively. The set of terms $\Lambda$, the set of types (or formulas) $\mathcal{F}$ and the set of syntactic ordinals $\mathcal{O}$ are defined mutually inductively. The terms and types are defined using the following two BNF grammars.

\[
\begin{align*}
t, u & ::= x \mid \lambda x.t \mid t \ u \mid \{ (l_i = t_i)_{i \in \mathcal{I}} \} \mid t.\mathcal{I} \mid C_k \ t \mid \{ t \mid (C_l \rightarrow t_l)_{i \in \mathcal{I}} \} \mid \varepsilon_{x \in A}(t \neq B) \\
A, B & ::= X \mid \{(l_i : A_i)_{i \in \mathcal{I}}\} \mid \{(l_i : A_i)_{i \in \mathcal{I}}\} \mid \{ (C_l \text{ of } A_i)_{i \in \mathcal{I}}\} \mid A \rightarrow B \mid \\
\forall X.A \mid \exists X.A \mid \mu_x.X.A \mid v_x.X.A \mid \varepsilon_{x}(t \in A) \mid \varepsilon_{x}(t \neq A)
\end{align*}
\]
The syntactic ordinals are build according to Definitions 2.2 and 3.2 using abstract judgments of the form \( J(t, F) = t : A \) and \( J(t, \overline{F}) = t \in A \subset B \), where the ordinals of \( \overline{F} \) may appear in the term \( t \) and formulas \( A \) and \( B \). In the BNF, \( I \) denotes a finite subset of \( \mathbb{N} \). The notation \( \{ (l_i = t_i) \mid i \in I \} \) for records denotes of vector inside braces. For example, if \( I = \{ 1, 2 \} \) then \( \{ (l_i = t_i) \mid i \in I \} \) corresponds to \( \{ l_1 = t_1; l_2 = t_2 \} \). Similar notations are used for pattern matchings, product types and sum types.

Convention 4.2. Before section 7, we will only apply the generalisation rule to subtyping judgements of the form \( \varepsilon_{x \in A}(x \notin B) \in A \subset B \), so the ordinals introduced for circular proofs that we will use will be of the form \( \varepsilon_{\pi < \kappa}(-(\varepsilon_{x \in A}(x \notin B) \in A \subset B))_1 \), which we will abbreviate as \( \varepsilon_{\pi < \kappa}(A \not\subset B)_1 \).

The term language contains the usual syntax of the \( \lambda \)-calculus extended with records, projections, constructors and pattern matching (see the reduction rules of Figure 6). A term of the form \( \varepsilon_{x \in A}(t \notin B) \) corresponds to a choice operator denoting a closed term \( u \) of type \( A \) such that \( t[x := u] \) does not have type \( B \).\(^\text{12}\) The restriction to closed choice is absolutely necessary for their interpretation in the semantics.

In addition to the usual types of System \( F \), our system provides sums and products (corresponding to variants and records), existential types, inductive types and coinductive types. Note that our product types may be either strict or extensible. A record having an extensible product type (marked with an ellipsis) will be allowed to contain more fields than those explicitly specified, while records with a strict product type will only contain the specified fields. From a subtyping point of view, extensible records are obviously more interesting. However, strict product types will allow us to express a stronger type safety result based on a semantic proof (Theorem 6.27). Our inductive and coinductive types carry size information in the form of a syntactic ordinals \( \kappa \). The symbol \( \infty \) denotes an ordinal large enough to ensure that the construction of \( \mu_{\infty} F \) and \( \nu_{\infty} F \) converges. In particular, when \( F \) is covariant they correspond to the least and greatest fixpoints of \( F \). Choice operators \( \varepsilon_X(t \in A) \) and \( \varepsilon_X(t \notin A) \) are also provided for types.\(^\text{13}\) As for our term choice operators, they correspond to witnesses of the property they denote, and they will be interpreted as such in the semantics. However, contrary to term choice operators, they do not need to be closed to be given a semantical interpretation.\(^\text{14}\)

Convention 4.3. To lighten the syntax and reduce the need for parentheses we will use some syntactic sugars. We will sometimes group binders and write \( \lambda x y. t \) for \( \lambda x. \lambda y. t \), and \( \forall X Y. A \) for \( \forall X. \forall Y. A \). Moreover, we will consider that binders have the lowest priority, which means that \( \lambda x. x \) is to be read as \( \lambda x. (x x) \), and \( \forall X. A \Rightarrow B \) as \( \forall X. (A \Rightarrow B) \). We will write \( \mu X. A \) for \( \mu_{\infty} X. A \) and \( \nu X. A \) for \( \nu_{\infty} X. A \). We will sometimes use the letter \( F \) to denote a type with one parameter \( X \) so that we can write \( F(\mu_X F) \) for \( A[X := \mu_X F] \). In pattern matchings, we will use the notation \( C_k x \rightarrow t \) to denote \( C_k \rightarrow \lambda x. t \). Finally, we will write \( t.C_k \) for the term \( t \mid C_k \rightarrow \lambda x. t \), also written \( t \mid C_k x \rightarrow x \).

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

Definition 4.4. The reduction relation \( (\succ) \subseteq \Lambda \times \Lambda \) is defined as the contextual closure of the rules given in Figure 6. Its reflexive, transitive closure is denoted \( (\succ^*) \).

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.

\(^{12}\)Note that in a choice operator like \( \varepsilon_{x \in A}(t \notin B) \), the variable \( x \) is bound in the term \( t \).

\(^{13}\)In the choice operators \( \varepsilon_X(t \in A) \) and \( \varepsilon_X(t \notin A) \) for types, the variable \( X \) is bound in \( A \) only.

\(^{14}\)We are not using this possibility currently.
\[(\lambda x.t)u > t[x := u] \quad \{ (l_i = t_i)_{i \in I} \} \ u > \Omega \]
\[
\{ (l_i : t_i)_{i \in I}, l_j > \begin{cases} t_j & \text{if } j \in I \\ \Omega & \text{otherwise} \end{cases} \}
\quad \{ (l_i : t_i)_{i \in I} \} \ u \quad \{ (l_i = t_i)_{i \in I} \} \ u > \Omega
\]
\[
[C_j u \mid (C_i \rightarrow t_i)_{i \in I}] > \begin{cases} t_j u & \text{if } j \in I \\ \Omega & \text{otherwise} \end{cases}
\quad [\{ (l_i = t_i)_{i \in I} \} \mid (C_i \rightarrow t_i)_{i \in I}] > \Omega
\]

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

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

**Definition 4.5.** In addition to rather usual typing judgments of the form \( \vdash t : A \), we introduce local subtyping judgements of the form \( \gamma \vdash t : A \subseteq B \) meaning “if \( t \) has type \( A \) then it also has type \( B \)” (in the positivity context \( \gamma \)). Usual subtyping judgments of the form \( \gamma \vdash A \subseteq B \) 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 7 and 8 respectively.

Before section 7, only subtyping judgement of the form \( \varepsilon_{x \in A}(x \notin B) \in A \subseteq B \) will be used to build well-founded circular proofs (see Section 3). To simplify, we will write \( \vdash A \subseteq B \) for \( \vdash \varepsilon_{x \in A}(x \notin B) \in A \subseteq B \). This also means that only subtyping proofs will be circular. Thus, the (G) and (I\( k \)) rules that we will really use can be written as follows.

\[
\forall \alpha(\gamma \vdash C(\alpha) \Rightarrow A \subseteq B) \quad (\gamma[\alpha := \overline{\alpha}], \delta + k_i \in C(\overline{\alpha}))_{1 \leq i \leq |\alpha|} \quad G
\]
\[
\gamma[\alpha := \overline{\alpha}] \varepsilon_{A[\alpha := \overline{\alpha}] \subseteq B[\alpha := \overline{\alpha}]} \quad \{ (C_i \text{ of } A_i)_{i \in I} \} \subseteq (C_i \rightarrow t_i)_{i \in I} \quad +_e
\]

\[
\forall \alpha(\gamma \vdash C(\alpha) \Rightarrow A \subseteq B)]_{k}
\]
\[
\gamma[\alpha := \overline{\alpha}] \varepsilon_{A[\alpha := \overline{\alpha}] \subseteq B[\alpha := \overline{\alpha}]} \quad \{ (C_i \text{ of } A_i)_{i \in I} \} \subseteq (C_i \rightarrow t_i)_{i \in I} \quad +_e
\]

Here the S subtyping rule is important, otherwise the term in subtyping judgement can only grow bottom-up and it would be impossible to form useful cycle. This rule allows for creating loop and is used before the G and I\( k \) rules to recover the same term.
The derivation of the former is given in Figure 9 (it is not circular). Note that the choice operators \( \nu \) and \( \mu \) for terms and types are all well defined (their definitions are not cyclic).

Subtyping rule applies for every two type constructors (see the beginning of Section 10).

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) rules, only one typing rule applies for every term constructor, and essentially one local subtyping rule applies for every two type constructors (see the beginning of Section 10).

**Example 4.6.** Mitchell’s containment axiom.

In our system, it is possible to derive Mitchell’s containment axiom [40], as well as one of its variations.

\[
\forall X. F(X) \rightarrow G(X) \subset (\forall X. F(X)) \rightarrow \forall X. G(X)
\]

The derivation of the former is given in Figure 9 (it is not circular). Note that the choice operators for terms and types are all well defined (their definitions are not cyclic).

**Example 4.7.** Mixed inductive and coinductive types.
Our system is suitable for handling types containing alternations of inductive and coinductive types. Let us consider the following two types $S$ and $L$ where $F(X, Y)$ is a predicate covariant in $X$ and in $Y$.

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

The elements of $S$ can be thought of as streams of $A$’s and $B$’s that only contain finitely many $A$’s. The elements of $L$ are streams that do not contain infinitely many consecutive $A$’s. In our system, it is possible to prove $S \subseteq L$ using the circular proof displayed in Figure 10. Note that the block decomposition of the proof is given in Example 3.14. We can thus conclude that it is well-founded (and thus valid).

## 5 FIXPOINT-LESS RECURSION FOR SCOTT ENCODING

In this section, we are going to demonstrate the expressivity of our system by exhibiting typable, pure $\lambda$-calculus recursors for Scott encoded data types. Scott encoding is similar to Church encoding, but it relies on (co-)inductive types as well as polymorphism. As first examples, we are going to consider the Church and Scott encodings of natural numbers. Although they have little (if any) practical interest, they demonstrate well the use of polymorphism and fixpoints. The type of Church numerals $\mathbb{N}_C$ and the type of Scott numerals $\mathbb{N}_S$ are defined below, together with their respective zero and successor functions.

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

$$0_C : \mathbb{N}_C = \lambda f \, x. x$$

$$S_C : \mathbb{N}_C \to \mathbb{N}_C = \lambda n \, f \, x. f \, (n \, f \, x)$$

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

$$0_S : \mathbb{N}_S = \lambda f \, x. x$$

$$S_S : \mathbb{N}_S \to \mathbb{N}_S = \lambda n \, f \, x. f \, n$$

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 $\mathbb{N}_S$. The latter requires some type annotations for our implementation to guess the correct instantiation of unifications variables. In particular, the type $\mathbb{N}_T = (T \to T) \to T \to T$ where $T = \mu X.((X \to \mathbb{B}) \to \mathbb{B})$ must be used for natural numbers. Note
that $T, F : \mathbb{B}$ denote booleans.

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

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

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

Scott numerals were initially introduced because they admit a constant time predecessor, whereas Church numerals do not. Usually, programming using Scott numerals requires the use of a recursor similar to that of Gödel’s System T. Such a recursor can be easily programmed using general recursion, however this would require introducing typable terms that are not strongly normalising. In our type system, we can typecheck a strongly normalisable recursor due to Michel Parigot[43]. It is displayed below together with several terms and types involved in its definition.

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

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

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

$\mathbb{N}' = \forall P. T(P) \to U(P) \to T(P) \to \mathbb{N}_S \to P$

Fig. 10. Example of circular proof involving inductive and coinductive types.
Scott numerals, a question arises about the inhabitants of the type

\[
\exists \theta : \forall P. P \rightarrow U(P) = \lambda a r q. a
\]

Theorem 6.25 implies that it is strongly normalising. The crucial point for typing the recursor is

\[
\delta : \forall P. P \rightarrow (N_S \rightarrow P \rightarrow P) \rightarrow T(P) = \lambda a f p r q. f (\text{pred} q) (p r (\zeta a) r q)
\]

\[
R_S : \forall P. P \rightarrow (N_S \rightarrow P \rightarrow P) \rightarrow N_S \rightarrow P = \lambda a f n.(n : N') (\delta a f) (\zeta a) (\delta a f) n
\]

It is easy to check that the term \(R_S\) is indeed a recursor for Scott numerals. It is similar to a \(\lambda\)-calculus
fixpoint combinator but it only allows a limited number of unfoldings. As the recursor is typable,

ACM Transactions on Programming Languages and Systems, Vol. 1, No. 1, Article 1. Publication date: July 2017.

1:28 Rodolphe Lepigre and Christophe Raffalli

able to program a strongly normalisable recursor for the usual type of unary natural numbers

but a function type is still required to program a strongly normalising recursor. We were not

the use of native records. It is in fact possible to use native sums for encoding Scott numerals,

and it is not part of the theoretical type system. In particular, the types of

\[
\zeta : \forall A. P \rightarrow (P \times Y) \rightarrow \{\text{hd} : (P \times Y) \rightarrow A; \text{tl} : Y \rightarrow P \times Y\}
\]

\[
S'(A, P) = \{\text{hd} : (P \times T(A, P)) \rightarrow A; \text{tl} : T(A, P) ; \text{st} : P \times T(A, P)\}
\]

\[
\zeta : \forall A. P. (P \rightarrow A) \rightarrow \forall X. (P \times X) \rightarrow A = \lambda f. s.1
\]

\[
\delta = \lambda f n.\{\text{hd} = \zeta f; \text{tl} = s.2; \text{st} = (n s.1, s.2)\}
\]

\[
\text{coiter} : \forall A. P. P \rightarrow (P \rightarrow A) \rightarrow (P \rightarrow P) \rightarrow S(A)
\]

\[
= \lambda s f n.\{\text{hd} = \zeta f; \text{tl} = s f n; \text{st} = (s, \delta f n)\}
\]

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

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.
6 REALIZABILITY SEMANTICS

In this section, we build a realizability model that is shown adequate with our type system. In particular, a formula $A$ is interpreted as a set of strongly normalising pure terms $[A]$. Consequently, if $\vdash t : A$ is derivable then we will have $\llbracket t \rrbracket \in [A]$, where $\llbracket t \rrbracket$ is the interpretation of $t$ as a pure term.

**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 $[A] \subseteq \Lambda$ the set of 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 rules of Figure 6 in any context (not only head contexts defined below). We denote $\mathcal{N} \subset [\Lambda]$ the set of strongly normalising pure terms.

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

$$H ::= \mathbb{1} \mid H \cdot t \mid H \cdot I_k \mid [H \mid (C_1 \rightarrow t_i)_{i \in I}]$$

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 $H > H'$ when $H[t] > H'[t]$ for any term $t \in \Lambda$ (including, for instance, $\lambda$-variables). We denote $(>_H)$ the head reduction relation defined as the contextual closure of the rules of Figure 6, restricted to contexts of $\mathcal{H}$. We say that a term is in head normal form if it cannot be reduced using $(>_H)$.

**Definition 6.3.** We say that a set of pure terms $\Phi \subset [\Lambda]$ is saturated if the following conditions hold.

1. If $\llbracket H[t[x := u]] \rrbracket \in \Phi$ and $u \in \mathcal{N}$ then $\llbracket H[(\lambda x. t) u] \rrbracket \in \Phi$.

2. If $\llbracket H[t u] \rrbracket \in \Phi$, then $\llbracket H[[C_k u \mid C_k \rightarrow t]] \rrbracket \in \Phi$.

3. If $\llbracket H[t] \rrbracket \in \Phi$ then $\llbracket [l_k = t; (l_i = t_i)_{i \in I}] \cdot l_k \rrbracket \in \Phi$ provided that $k \notin I$ and $t_i \in \mathcal{N}$ for all $i \in I$.

4. If $\llbracket H[t \mid (C_i \rightarrow t_i)_{i \in I}] \rrbracket \in \Phi$ and $I \subseteq J$ then $\llbracket H[t \mid (C_i \rightarrow t_j)_{i \in J}] \rrbracket \in \Phi$ provided that $t_j \in \mathcal{N}$ for all $j \in J \setminus I$.

5. If $t \in \Phi$ and $t >_H u$, then $u \in \Phi$ ($\Phi$ is closed by weak head reduction).

Remark: the two last conditions, in particular requiring closure under head reduction, are unusual. But they are necessary for the adequacy of the subtyping rule on sum types.

**Lemma 6.4.** If $\Phi \subset [\Lambda]$ is saturated, then for all $t \in \mathcal{N}$ and $u \in \Phi$, $t >_H u$ implies $t \in \Phi$.

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

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

**Proof.** We show that it satisfies the five conditions of Definition 6.3.

1. Let us take $\llbracket H[t[x := u]] \rrbracket \in \mathcal{N}$ and suppose, by contradiction, that $\llbracket H[(\lambda x. t) u] \rrbracket \notin \mathcal{N}$. There cannot be an infinite reduction of $H$, $t$ or $u$. Hence, an infinite reduction of $\llbracket H[(\lambda x. t) u] \rrbracket$ must start with $\llbracket H[(\lambda x. t) u] \rrbracket >^* H'[((\lambda x. t') u')] > H'[t'[x := u']]$, where $H >^* H'$, $t >^* t'$ and $u >^* u'$. We then contradict $\llbracket H[t[x := u]] \rrbracket \in \mathcal{N}$ by transforming this reduction into $\llbracket H[(\lambda x. t) u] \rrbracket > H[t[x := u]] >^* H'[t'[x := u']]$.

2. Let us take $\llbracket H[t u] \rrbracket \in \mathcal{N}$ and suppose, by contradiction, that $\llbracket H[[C_k u \mid C_k \rightarrow t]] \rrbracket \notin \mathcal{N}$. As in the previous case, there cannot be an infinite reduction of $H$, $u$ or $t$. As a consequence, an infinite reduction of $\llbracket H[[C_k u \mid C_k \rightarrow t]] \rrbracket$ necessarily starts with $\llbracket H[[C_k u \mid C_k \rightarrow t]] \rrbracket >^* H'[C_k u'] > H'[t'u']$, where $H >^* H'$, $t >^* t'$ and $u >^* u'$. This can be transformed into $\llbracket H[[C_k u \mid C_k \rightarrow t]] \rrbracket > H[t u] >^* H'[t'u']$, which contradicts $\llbracket H[t u] \rrbracket \in \mathcal{N}$.

---

ACM Transactions on Programming Languages and Systems, Vol. 1, No. 1, Article 1. Publication date: July 2017.
Let us take $H[t] \in \mathcal{N}$ and $t_i \in \mathcal{N}$ for all $i \in I$, and suppose, by contradiction, that $H[\{l_k = t; (l_i = t_i)_{i \in I}\}] \notin \mathcal{N}$. There cannot be an infinite reduction of $H$, $t$ nor any of the $t_i$. Consequently, an infinite reduction of $H[\{l_k = t; (l_i = t_i)_{i \in I}\}]$ must start with $H[\{l_k = t; (l_i = t_i)_{i \in I}\}]^* \succ H[\{l_k = t'; (l_i = t'_i)_{i \in I}\}].$ 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 $H[\{l_k = t; (l_i = t_i)_{i \in I}\}]^* \succ H[t'] > H'[t'].$

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

(5) The set $\mathcal{N}$ is obviously closed under head reduction.

**Definition 6.6.** The set of neutral terms $\mathcal{N}_0 \subset [\Lambda]$ is the smallest set such that:

1. for every $\lambda$-variable $x$ we have $x \in \mathcal{N}_0$,
2. for every $u \in \mathcal{N}$ and $t \in \mathcal{N}_0$ we have $t u \in \mathcal{N}_0$,
3. for every $i \in \mathbb{N}$ and $t \in \mathcal{N}_0$ we have $t l_i \in \mathcal{N}_0$,
4. for every $(C_i, t_i)_{i \in I} \in (C \times \mathcal{N})^I$ and $t \in \mathcal{N}_0$ we have $[t \mid (C_i \rightarrow t_i)_{i \in I}] \in \mathcal{N}_0$.

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

**Definition 6.7.** Given a set of pure values $\Phi \subseteq [\Lambda]$, we denote $\overline{\Phi} \subseteq [\Lambda]$ the smallest saturated set containing $\Phi$.

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

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

**Definition 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 $t$ will be replaced by (possibly open) pure terms in $[t]$. A formula $A \in \mathcal{F}$ will be interpreted by a saturated set of pure terms $[A]$ such that $\overline{\mathcal{N}_0} \subseteq [A] \subseteq \mathcal{N}$. Note that a syntactic ordinals $\kappa \in \mathcal{O}$ will be interpreted by an actual ordinal $[\kappa] \in [\mathcal{O}]$ according to Section 2. Of course, the interpretation of syntactic ordinals will involve the interpretation of terms and formulas through abstract 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).

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

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

**Definition 6.13.** The sets of parametric terms $\Lambda$ and the set of parametric formulas $\mathcal{F}^*$ are formed by extending the syntax of formulas with the elements of $\mathcal{F}$. Terms do not need to be extended directly, however, the definition of $\mathcal{F}^*$ impacts the definition of $\Lambda$ since terms and formulas are defined mutually inductively. A closed parametric term (resp. formula, resp. syntactic ordinal) is a parametric term (resp. formula, resp. syntactic ordinal) that does not contain free propositional variables nor free ordinal variables. Note however that $\lambda$-variables are allowed. This is due to the definition of $N_0$.

**Definition 6.14.** The interpretation of a closed parametric term $t \in \Lambda$ (resp. closed parametric formula $A \in \mathcal{F}^*$) is defined to be a pure term $[t] \in \Lambda$ (resp. a set of pure terms $[A] \in \mathcal{F}$) defined inductively according to Figure 11 and Definition 2.6. Note that the semantics of terms, types and syntactic ordinals should be defined mutually inductively due to choice operators (or witnesses). In particular, the abstract judgements used in the definition of choice operators for ordinals are interpreted in the obvious way according to Definition 3.4.

In the interpretation of choice operators of the form $\varepsilon x A(t \notin B)$, it is important that no $\lambda$-variable other that $x$ is bound in $t$. This is enforced by a syntactic restriction given in Definition 4.1. Without this restriction, a term $[\lambda y. \varepsilon x A(t \notin B)]$ with $y$ free in $t$ would correspond to a function that is not always definable using a pure term. Thus, our model would have circular (and hence invalid) definitions. Note that the axiom of choice is required to interpret the choice operators.

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

**Lemma 6.15.** The semantic interpretation of terms, formulas and syntactic ordinals commutes with the substitution of the three kinds of variables. We thus have, for example, $[t[X := A]] = [t[X := [A]]]$ or $[[A[\alpha := \kappa]]] = [[A[\alpha := \kappa]]]$.

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

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

**Proof.** Since $\Phi$ and $\Psi$ are candidates, we know $\overline{N}_0 \subseteq \Phi \subseteq N$ and $\overline{N}_0 \subseteq \Psi \subseteq N$. As a consequence, we can use Lemma 6.10 to obtain $(N \Rightarrow \overline{N}_0) \subseteq (\Phi \Rightarrow \Psi)$ (using $\Phi \subseteq N$ and $\overline{N}_0 \subseteq \Psi$) and $(\Phi \Rightarrow \Psi) \subseteq (\overline{N}_0 \Rightarrow N)$ (using $\overline{N}_0 \subseteq \Phi$ and $\Psi \subseteq N$). We then obtain $\overline{N}_0 \subseteq (\Phi \Rightarrow \Psi) \subseteq N$ with Lemma 6.11. It remains to show that $\Phi \Rightarrow \Psi$ is saturated by proving the five conditions.

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

15The set of parametric syntactic ordinals $O^*$ of Definition 2.5 should also be impacted.
We now suppose $H(t \cdot u) \in (\Phi \Rightarrow \Psi)$ and show $H([D \cdot u \mid D \rightarrow t]) \in (\Phi \Rightarrow \Psi)$. We thus take $v \in \Phi \subseteq N$ and we prove $H([D \cdot u \mid D \rightarrow t]) \in \Psi$. As $H(t \cdot u) \in (\Phi \Rightarrow \Psi)$ we know that $H(t \cdot u) \not\in \Psi$ and thus we can conclude using the saturation condition (2) of $\Psi$ with the context $H \cdot u$.

Let us now suppose that $H[t] \in (\Phi \Rightarrow \Psi)$ and that $t_i \in N$ for all $i \in I$. We need to show that $H[t; \{l_i = t_i\}_{i \in I}, \{l_k\}] \in (\Phi \Rightarrow \Psi)$ so we take $v \in \Phi \subseteq N$ and we prove $H[t; \{l_i = t_i\}_{i \in I}, \{l_k\}] \not\in \Psi$. As $H[t] \in (\Phi \Rightarrow \Psi)$ we know that $H[t] \not\in \Psi$ and thus conclude with the saturation condition (3) of $\Psi$ with the context $H \cdot v$.

We now suppose $H[t \mid (C_i \rightarrow t_i)_{i \in I}] \in (\Phi \Rightarrow \Psi)$ and $I \subseteq J$ with $t_j \in N$ for all $j \in J \setminus I$. We need to show $H[t \mid (C_i \rightarrow t_i)_{i \in I}] \in (\Phi \Rightarrow \Psi)$ so we take $v \in \Phi \subseteq N$ and we prove $H[t \mid (C_i \rightarrow t_i)_{i \in I}] \not\in \Psi$. As we have $H[t \mid (C_i \rightarrow t_i)_{i \in I}] \not\in \Psi$ and thus we can conclude using the saturation condition (4) of $\Psi$ with the context $H \cdot v$.

![Fig. 11. Semantical interpretation of closed parametric terms and types.](image-url)
(5) We now prove closure under head reduction. Let us take $t \in (\Phi \Rightarrow \Psi)$ such that $t \rightarrow_H t'$ and show that $t' \in (\Phi \Rightarrow \Psi)$. We take $u \in \Phi$ and show $t' u \in \Psi$. Since $\Psi$ is closed under head reduction and $t u \rightarrow_H t' u$ it is enough to show $t u \in \Psi$, which follows from the definition of $t \in (\Phi \Rightarrow \Psi)$. □

Lemma 6.17. If for all $i \in I$ we have $\Phi_i \in [F]$ then $[[\{C_i \mid \Phi_i \}_i]] \subseteq N$.

Proof. By definition, it is easy to see that $[[\{C_i \mid \Phi_i \}_i]] \subseteq N$. Let us take $t \in \overline{N}_0$ and show that $t \in [[\{C_i \mid \Phi_i \}_i]]$. We thus take $\Phi \in [F]$ and $t_i \in (\Phi \Rightarrow \Phi)$ for all $i \in I$, and show $[|t|_{C_i \rightarrow t_i}] \in \Phi$. This is immediate as we have $[|t|_{C_i \rightarrow t_i}] \in N_0$ and $\overline{N}_0 \subseteq \Phi$. It remains to show that $[[\{C_i \mid \Phi_i \}_i]]$ is saturated by proving the five saturation conditions.

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

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

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

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

5. We finish by proving closure under head reduction. Let us take $t \in [[\{C_i \mid \Phi_i \}_i]]$ such that $t \rightarrow_H t'$ and show that $t' \in [[\{C_i \mid \Phi_i \}_i]]$. We thus take $\Phi \in [F]$ and $t_i \in (\Phi \Rightarrow \Phi)$ for all $i \in I$, and we show $[t'] \in \Phi$. Since $t \in [[\{C_i \mid \Phi_i \}_i]]$, we know that $[t] \in (C_i \rightarrow t_i)_{i \in I} \in \Phi$. We thus conclude since $\Phi$ is saturated and $[t \rightarrow (C_i \rightarrow t_i)_{i \in I}] \rightarrow_H \{t' \rightarrow (C_i \rightarrow t_i)_{i \in I}\}$. □

Lemma 6.18. If for all $i \in I$ we have $\Phi_i \in [F]$ then $\{\{l_i : \Phi_i\}_{i \in I} : \ldots\} \in [F]$.

Proof. Similar to the proofs of Lemmas 6.16 and 6.17. □

Lemma 6.19. If for all $i \in I$ we have $\Phi_i \in [F]$ then $\{\{l_i : \Phi_i\}_{i \in I}\} \in [F]$.

Proof. Immediate if $I = \emptyset$ and similar to the proofs of Lemmas 6.16 and 6.17 otherwise. □

Theorem 6.20. For every closed parametric term $t \in \mathcal{N}$ (resp. ordinal $\kappa \in O^*$, resp. type $A \in F^*$) we have $[t] \in [\Lambda]$ (resp. $[\kappa] \in [O]$, resp. $[A] \in [F]$).

Proof. We do a proof by induction. For terms, all the cases are immediate by induction hypothesis. For instance, if $u = \{x \in A \mid t \neq B\}$ then we have $u \in [A] \subseteq \mathcal{N} \subseteq [\Lambda]$ by induction hypothesis, or $u \in \overline{N}_0 \subseteq [\Lambda]$. For ordinals, the proof is immediate by Definition 2.6 and using the induction...
hypothesis to interpret predicates in ordinal witnesses. For types of the form $\Phi \subseteq \mathcal{F}$, $\varepsilon_X(t \in A)$ or $\varepsilon_X(t \notin A)$ the proof is immediate. For types of the form $A \Rightarrow B, [(C_i \text{ of } A_i)_{i \in I}], \{(l_i : A_i)_{i \in I} \ldots \}$ or $\{(l_i : A_i)_{i \in I} \}$ then we respectively use Lemma 6.16, 6.17, 6.18 or 6.19 with the induction hypotheses and Lemma 6.15. The remaining four possible forms of types 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 using the fact that an intersection of candidates is itself a candidate.

- For types of the form $\exists X . A$, the proof is similar to the previous case, using the fact that a union of candidates is itself a candidate.

- For types of the form $\mu X . A$, we show $[\mu X . A] \in \mathcal{F}$ for all $\mu \leq \kappa$ by induction on the ordinal $\mu$. This is enough as we can then conclude using Lemma 6.15 to show $[\mu X . A] = [\mu[X] . A] = [\mu X . A] \in \mathcal{F}$. If $\mu = 0$ then we have $[\mu X . A] = \mathcal{N}_0$ and the proof is thus immediate. Otherwise, we have $[\mu X . A] = \cup_o [A[X := \nu o X . A]]$. Using the local induction hypothesis we get $[\nu o X . A] \in \mathcal{F}$ for all $o < \mu$. Using Lemma 6.15, we then obtain $[A[X := \nu o X . A]] = [A[X := \nu o X . A]] \in \mathcal{F}$ for all $o < \mu$ using the global induction hypothesis. We can then conclude using again the fact that a union of candidates is itself a candidate.

- For types of the form $\nu X . A$, we proceed in a similar way as in the previous case, using again the fact that an intersection of candidates is itself a candidate. Note that we have $[\nu_0 X . A] = \mathcal{N} \in \mathcal{F}$ in the case of the zero ordinal.

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

**Lemma 6.21.** Every strongly normalising pure term $t \in \mathcal{N}$ has a head normal form that is either a $\lambda$-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 $\lambda$-abstraction, a record, a constructor or a $\lambda$-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 $[A_i] \in \mathcal{F}$ for all $i \in I$, then we have $t \in [(C_i \text{ of } A_i)_{i \in I}]$ if and only if $t \in \mathcal{N}$ and either $t \uparrow^*_H u$ with $u \in \mathcal{N}_0$ or $t \uparrow^*_H C_k u$ with $k \in I$ and $u \in [A_k]$.

**Proof.** ($\Rightarrow$) Let us suppose that $t \in [(C_i \text{ of } A_i)_{i \in I}]$. By definition, we immediately have $t \in \mathcal{N}$, so according to Lemma 6.21 there is a head normal form $\nu$ such that $t \uparrow^*_H \nu$, and we only need to show that $\nu$ cannot be a $\lambda$-abstraction, a record, a term of the form $C_k u$ with $k \notin I$, or a term of the form $C_k u$ with $k \in I$ and $u \notin [A_k]$. To rule out the first three possibilities, we apply the definition of $[(C_i \text{ of } A_i)_{i \in I}]$ using the fact that $\lambda x . \nu \in ([A_i] \Rightarrow \mathcal{N})$ for all $i \in I$ to obtain $[t \mid (C_i \rightarrow \lambda x . \nu)_{i \in I}] \in \mathcal{N}$. We thus have $[\nu \mid (C_i \rightarrow \lambda x . \nu)_{i \in I}] \in \mathcal{N}$ since $t \uparrow^*_H u$, but this term diverges if $\nu$ has one of the first three forms. Let us now suppose that there is $k \in I$ such that $\nu = C_k u$. We consider the term $u_k = [t \mid C_k \rightarrow \lambda x . (C_i \rightarrow \lambda y . y)_{i \in I \setminus \{k\}}]$ where $y$ is a fresh variable. Obviously, we have $\lambda x . \nu \in ([A_i] \Rightarrow \mathcal{A}_i)$ and $\lambda y . \nu \in ([A_i] \Rightarrow \mathcal{N}_0) \subseteq ([A_i] \Rightarrow \mathcal{A}_i)$ for all $i \in I \setminus \{k\}$. Therefore, we can use the definition of $[(C_i \text{ of } A_i)_{i \in I}]$ to obtain $u_k \in [A_k]$. We can then conclude that $u \in [A_k]$ as $[A_k]$ is saturated and $u_k \uparrow^*_H (\lambda x . \nu) u \uparrow^*_H u$.

($\Leftarrow$) Let us now suppose that $t \in \mathcal{N}$ and that $t \uparrow^*_H u$ with either $u \in \mathcal{N}_0$ or $u = C_k u$ with $k \in I$ and $u \in [A_k]$. We need to show $t \in [(C_i \text{ of } A_i)_{i \in I}]$, so we take a set $\Phi \in \mathcal{F}$, terms
$t_i \in ([A_i] \Rightarrow \Phi)$ for all $i \in I$, and we show $[t \mid (C_i \rightarrow t_i)_{i \in I}] \in \Phi$. Since $t >_H v$ we also have $[t \mid (C_i \rightarrow t_i)_{i \in I}] >_H [v \mid (C_i \rightarrow t_i)_{i \in I}]$ and thus it is enough to show $[v \mid (C_i \rightarrow t_i)_{i \in I}] \in \Phi$ according to Lemma 6.4. Now, if $v \in N_0$ then we have $[v \mid (C_i \rightarrow t_i)_{i \in I}] \in N_0$ and we can conclude immediately. If $v = C_k u$ with $k \in I$ and $u \in [A_k]$, then we need to show $t_k u \in [A_k]$, which follows from $t_k \in ([A_k] \Rightarrow \Phi)$.

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

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

1. If $\gamma + t \in A \subseteq B$ is derivable by a well-founded proof and $[t] \in [A]$ then $[t] \in [B]$.
2. If $t : A$ is derivable by a well-founded proof then $[t] \in [A]$.

**Proof.** According to Theorem 3.17 we only have to prove that our typing and subtyping rules are correct. Note that the truth of our abstract judgements $[\tau : A] = 1$ and $[t : A \subseteq B] = 1$ is defined according to the statement of the current theorem. We thus consider all the rules of Figure 12 and 8.

($\rightarrow$) We need to show $[\lambda x.t] \in [C]$. However, according to the second induction hypothesis, it is enough to show $[\lambda x.t] \in [A \Rightarrow B] = ([A] \Rightarrow [B])$. Using the second induction hypothesis we have $[t[x := e_{x \in A}(t \notin B)] \in [B]$. By definition of the choice operator, this means that we have $[t[x := u]] = [t][x := u] \in [B]$ for all $u \in [A]$. By Theorem 6.20 we know that $[B]$ is saturated and that $[A] \subseteq N$. We then use the saturation condition (1) to get $[\lambda x.t] u \in [B]$ for all $u \in [A]$.

($\rightarrow_e$) We need to show $[t][u] \in [B]$. By induction hypothesis we have $[t] \in [A \Rightarrow B]$ and $[u] \in [A]$, so we can conclude by definition of $[A \Rightarrow B] = ([A] \Rightarrow [B])$.

(\text{e}) We need to show $[\varepsilon_{x \in A}(t \notin B)] \in [C]$. However, according to the induction hypothesis, it is enough to show $[\varepsilon_{x \in A}(t \notin B)] \in [A]$. This follows immediately from the definition of $[\varepsilon_{x \in A}(t \notin B)]$. In particular, $N_0 \subseteq [A]$ by Theorem 6.20.

($x_i$) We need to show that $[(l_i = t_i)_{i \in I}] \in [B]$. According to the first induction hypothesis, it is enough to show $[(l_i = t_i)_{i \in I}] \in [(l_i = A_i)_{i \in I}]$. By definition, we need to take $k \in I$ and show $[(l_k = t_k)_{i \in I}] \in [A_k]$. By induction hypothesis we know that $[t_k] \in [A_k]$, hence we can use the saturation condition (3) on $[A_k]$ since it is saturated by Theorem 6.20. Note that if $k \notin I$ then we immediately have $[(l_i = t_i)_{i \in I}] \in \Omega$.

($x_e$) We need to show $[t.l_k] = [t].[l_k] \in [A]$. As we have $[t] \in [(l_k : A ; \ldots)]$ by induction hypothesis, we can conclude by definition of $[(l_k : A ; \ldots)]$.

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

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

($\rightarrow$) 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 $v \in [A_1]$,
hence $[t] \nu \in [B_1]$ by definition of $t \in [A_1 \rightarrow B_1]$. Using the second induction hypothesis this gives $[t] \nu \in [B_2]$, which is a contradiction.

\((=)\) This is a trivial implication.

\((\Rightarrow)\) We assume $[t] \in [A]$ and $[t] \not\in [B]$. By definition of the semantic of $u = [\varepsilon_{x \in A}(x \not\in B)]$, this implies that $u \in [A]$ and $u \not\in [B]$ which contradicts the induction hypothesis given by the premise.

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

\((\forall_r)\) We assume $[t] \in [A]$, and we show $[t] \in [\bigvee X . B]$. Using the induction hypothesis we obtain $[t] \in [B[X := \varepsilon_X(t \not\in B)]]$. Consequently we have $[t] \in [B[X := \Phi]]$ for all $\Phi \in [\mathcal{F}]$ by definition of the choice operator, and thus $[t] \in [\bigvee X . B]$.  

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

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

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

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

\((\times_p)\) Also similar to the \((\times_s)\) case.

\((+)\) We assume $[t] \in \{(C_i \text{ of } A_i)_{i \in I} \}$ and we show $[t] \in \{(C_i \text{ of } B_i)_{i \in I} \}$. According to Lemma 6.22, we know that $t >^*_H v$ with only two possibilities for $v$. In the case where $v \in \overline{N}_0$ then we can conclude directly using Lemma 6.22 in the other direction. Otherwise, we know that $t >^*_H C_k u$ with $k \in I_1 \subseteq I_2$ and $u \in [A_k]$. We now consider the term $t.C_k$ which reduces as $t.C_k >_H \Omega \cup k \rightarrow \lambda X . x \rightarrow \Phi$ for all $\Phi \in \mathcal{F}$. Hence, we obtain $t.C_k \in [A_k]$. According to the \((\exists_r)\) case, we can deduce $u \in [B_k]$. We can then conclude using the (right to left direction of) Lemma 6.22.

\((\mu_r)\) We assume $[t] \in [A]$ and we show $[t] \in [\mu_r F]$. By the first induction hypothesis we obtain that $[t] \in [F(\mu_r F)]$, so it only remains to show $[F(\mu_r F)] \subseteq [\mu_r F]$. According to the second induction hypothesis, using Lemma 2.11, we know that $[\tau] < [\lambda X . t]$ and we thus obtain $[F(\mu[X] F)] \subseteq [\mu[X] F]$ by definition of $[\mu[X] F]$. We then obtain $[F(\mu_r F)] = [F(\mu[X] F)] \subseteq [\mu[X] F] = [\mu_r F]$ using Lemma 6.15 twice.

\((\nu_l)\) Similar to the \((\mu_r)\) case.

\((\mu_r^\omega)\) We assume $[t] \in [A]$ and we show $[t] \in [\mu^\omega F]$. By induction hypothesis, we obtain $[t] \in [F(\mu^\omega F)]$ so we only need to show $[F(\mu^\omega F)] \subseteq [\mu^\omega F]$. Since the cardinal of the ordinal $\omega^\omega$ is $2^{\omega^\omega}$, it is larger than the cardinal of $[\mathcal{F}]$ which is $2^\omega$. Hence the inductive definition of $[\mu^\omega F]$ must reach its stationary point strictly before $\omega^\omega$. As a consequence, we have $[\mu^\omega F] = [\mu^{\omega + 1} F] \supseteq [F(\mu^\omega F)]$ by definition. We can thus conclude using Lemma 6.15 on both sides.

\((\nu_l^\omega)\) Similar to the \((\mu_r^\omega)\) case.

\((\mu_l)\) Let us suppose that $[t] \in [\mu_r F]$ and show that $[t] \in [B]$. If $[\lambda X . t] = 0$ then this is immediate since in this case we have $[\mu_r F] = \overline{N}_0$, and thus $[t] \not\in [B]$ since $\overline{N}_0 \not\subseteq [B]$ according to

\[\text{This stationary point is not a fixpoint if } F \text{ is not covariant, but we do not need this information.}\]
Theorem 6.20. If $[k] \neq 0$ then by definition there must be $o < [k]$ such that $[t] \in [F(\mu_o F)]$.
By definition of the choice operator, this means that $o = \lfloor u_{<k}(t \in F(\mu_o F)) \rfloor$ does verify $o < [k]$ and $[t] \in [F(\mu_o F)]$. We can thus conclude using the induction hypothesis.

$(\nu_r)$ Similar to the $(\mu)$ 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), we have $[t] \in \overline{N}_0$ since $[\forall X. X] = [[]] = \overline{N}_0$ by definition. This is a contradiction since $\overline{N}_0$ only contains open terms.

**Theorem 6.25.** Given a closed, pure term $t \in [] \Lambda$ and a closed type $A \in \mathcal{F}$, if $\vdash t : A$ is derivable then $t$ is strongly normalising.

**Proof.** Using the adequacy lemma (Theorem 6.23), we know that $[t] \in [A]$. However, since $t$ is pure we have $[t] = t$. Moreover, according to Theorem 6.20 we have $[A] \subseteq N$, and thus we obtain $t \in N$.

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

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

**Theorem 6.27.** If $\vdash t : A$ is derivable for a closed, pure term $t$ and a simple type $A$, then $t$ reduces to a normal form $u$ such that $\vdash u : A$ is derivable.

**Proof.** According to Theorem 6.25, we know that $t$ must reduce to a normal form $u$. Moreover, $u$ is closed since no free variables are introduced by our reduction rules. We proceed by induction on the size of $u$. In the case where $A = \mu X.B$ we know that $[B[X := A]] \subseteq [A]$. Let us define $A' = B[X := A]$ if $A = \mu X.B$ and $A' = A$ otherwise. The hypotheses on least fixpoints are still true in $A'$ since $B$ cannot be equal to $X$ by hypothesis. Moreover, since pure types may not contain two consecutive fixpoints and $A$ cannot be $\mu X.B$, $A'$ is either a sum type or a strict product type.

If $A' = [(C_i : A_i)_{i \in I}]$ then, by Lemma 6.22 we know that $u = C_k v$ with $k \in I$ and $v \in [A_k]$. In particular, $u$ is in normal form (and thus in head normal form) and it cannot be open, which means that $u \notin \overline{N}_0$. Since $u$ is in normal form, we know that $v$ is also in normal form. The induction hypothesis provides us with a derivation of $\vdash v : A_k$. In the case where $A' = A = [(C_i : A_i)_{i \in I}]$ then we can conclude using the following derivation.

$$
\begin{align*}
\vdash \{k\} \subseteq I & \quad \vdash t . C_k \in A_k \subseteq A_k \\
\vdash C_k v : [(C_k : A_k) \subseteq [(C_i : A_i)_{i \in I}]] & + \\
\vdash C_k v : [(C_i : A_i)_{i \in I}] & +_i
\end{align*}
$$
Otherwise, if we have $A = \mu X.[(C_i : A_i)_{i \in I}]$ then $A' = [(C_i : A_i[X := A])_{i \in I}]$ and we can construct the following derivation. Note that in this case, $A_k$ is rather of the form $A_k[X := A]$, so we in fact have a proof of $\vdash v : A_k[X := A]$.

$$
\begin{align*}
\{k\} \subseteq I & \quad \vdash t.C_k \in A_k[X := A] \subseteq A_k[X := A] \\
\vdash C_k v \in [C of A_k] \subseteq [(C_i : A_i[X := A])_{i \in I}] & \quad + \\
\vdash C_k v \in [C of A_k] \subseteq \mu X.[(C_i : A_i)_{i \in I}] & \quad \mu^\infty \\
\vdash C_k v : \mu X.[(C_i : A_i)_{i \in I}] & \quad \vdash v : A_k[X := A] +_i
\end{align*}
$$

Now, if $A' = \{(l_i : A_i)_{i \in I}\}$ is a strict product type then the proof is similar. However, we first need to remark that $v = \{(l_i = v_i)_{i \in I}\}$ with $v_i \in [A_i]$ for all $i \in I$. Note that all the other possible forms of normal forms can be ruled out using similar techniques as in the proof of Lemma 6.22. By induction hypothesis, we can obtain a proof of $\vdash v_i : A_i$ for all $i \in I$ and then reconstruct proofs as in the case of the sum types.

To conclude this section, we will discuss the closure by head reduction imposed in our definition of saturation. This condition is not usually required, but it is needed here for a subtle reason. Although it is used in the proof of Theorem 6.27, the main aim of this condition is to allow for the correctness of the subtyping rule for sums recalled below.

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

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

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

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

### 7 FIXPOINT AND TERMINATION

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

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

Finally, proving the termination of recursive programs will generally require us to extend our typing judgments with ordinal contexts. We will then be able to assume that certain ordinals are positive while building typing proofs. For example, if we know that $l : L_{\alpha}(A)$ and we want to type the case analysis $[l \mid \text{Nil} \rightarrow u \mid \text{Cons} \rightarrow v]$, then we can assume that $\alpha > 0$ when typing $u$ and $v$. Indeed,
We extend the syntax of terms and types given in Definition 4.1 with a fixpoint combinator and new connectives as follows.

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

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

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

**Convention 7.2.** We will use the abbreviations $$A \land \gamma$$ and $$A \lor \gamma$$, where $$\gamma = \kappa_1, \ldots, \kappa_n$$ is an ordinal context, to denote $$A \land \kappa_1 \ldots \land \kappa_n$$ and $$A \lor \kappa_1 \ldots \lor \kappa_n$$ respectively. In particular, if $$\gamma = \emptyset$$ then we have $$A \land \gamma = A \lor \gamma = A$$. We will also use the notation $$\gamma_1, \gamma_2$$ for the union of the ordinals contexts $$\gamma_1$$ and $$\gamma_2$$.

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

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

Fig. 12. Typing of the system extended with general recursion.
\[
\frac{\gamma \vdash t \in A[\alpha := \kappa] \subseteq B}{\gamma \vdash t \in \forall \alpha.A \subseteq B} \quad \forall_r
\]

\[
\frac{\gamma \vdash t \in B \subseteq A[\alpha := \kappa]}{\gamma \vdash t \in B \subseteq \exists \alpha.A} \quad \exists_r
\]

\[
\frac{\gamma, \kappa, t \in A \subseteq B}{\gamma, \kappa \vdash t \in A \land \kappa \subseteq B} \quad \land_I
\]

\[
\frac{\gamma, \kappa, t \in A \subseteq B}{\gamma \vdash t \in A \lor \kappa \subseteq B} \quad \lor_r
\]

\[
\frac{\forall \alpha(\gamma \vdash C(\alpha) \Rightarrow Yx.t : A)}{\gamma[\alpha := \kappa], \delta \vdash \kappa \subseteq C(\kappa)_{1 \leq i \leq |\pi|} \quad \forall_{\alpha}(\gamma \vdash C(\alpha) \Rightarrow Yx.t : A)}
\]

\[
[\forall \alpha(\gamma \vdash C(\alpha) \Rightarrow Yx.t : A)]_k
\]

\[
\frac{\gamma[\alpha := \kappa] \vdash Yx.t : A[\alpha := \kappa]}{\forall \alpha(\gamma \vdash C(\alpha) \Rightarrow Yx.t : A)}
\]

\[
\text{id} \quad \text{id} = Yid.\lambda n.[n | Z \rightarrow z | Sp \rightarrow s \ (id \ p)]
\]

**Definition 7.4.** Our typing judgments now have the form \( \gamma \vdash t : A \), where \( \gamma \) is an ordinal context. The typing rules of the extended system are given in Figure 12. Its subtyping rules still include those of Figure 8, but the rules of Figure 13 are added to handle the new connectives.

The typing rules of the system need to be changed completely to account for the ordinal contexts. Note that they are strongly linked to the new connectives \( A \land \alpha \) and \( A \lor \alpha \) in types. Moreover, the \((\times_i)\) and \((+_i)\) rules require some terms to be weakly normal\(^{17}\) to learn the positivity of certain syntactic ordinals. Furthermore, the system includes both circular subtyping and typing proofs to handle general recursion. Note that the typing rule of the fixpoint is very simple as it only performs an unfolding. In practice, we will only need to allow circularity on typing judgments of the form \( \gamma \vdash Yx.t : A \). In this context, the \((G)\) and \((I_k)\) rules can be written as in Figure 14, where we write \( \varepsilon_{\alpha < C(\alpha)}(Yx.t \notin A) \) for the ordinal \( \varepsilon_{\alpha < C(\alpha)}(Yx.t : A) \) (see Section 3).

\(^{17}\)If we were to consider laziness, the rules would have to be changed.
Example 7.5. We consider the identity function on unary natural numbers. It can be typed using the derivation given in Figure 15, which is the simplest possible example of a circular typing proof. Following the terminology of Section 3, the proof is formed using two blocks. The former, that we will call $B_0$, starts at the root of the proof and only contains one typing rule. The latter, that we will call $B_1$, contains all the rest of the proof. The call graph corresponding to the proof contains one edge from $B_0$ to $B_1$, labelled with the empty matrix, and one edge from $B_1$ to itself, labelled with the $1 \times 1$ matrix $(-1)$ since we can prove $\kappa_0 \vdash \kappa_1 < \kappa_0$.

It is important to note that the positivity of $\kappa_0$ must be known to obtain $\kappa_1 < \kappa_0$. It is thus essential to use the type $F(\mathbb{N}_{\kappa_1}) \land \kappa_0$ (and not $F(\mathbb{N}_{\kappa_1})$) for the first premise of the $(+_e)$ rule. This allows us to assume that $\kappa_0$ is positive when typing its other premises. There would be no way of building a typing proof without doing so.

We will now modify the semantics that was given in Section 6 to account for the fixpoint combinator and the new connectives. Here, we will not be able to interpret types as subsets of $\mathcal{N}$, since the reduction rule of the fixpoint will break strong normalisation. We will however be able to preserve normalisation for all the weak reduction strategies (i.e., those that do not reduce under $\lambda$-abstractions, and thus case analyses as well).

Definition 7.6. We denote $(>_w) \subseteq \Lambda \times \Lambda$ the one step weak reduction relation. It is defined as the least relation containing the rules of Figure 6 and $Yx.t >_w t[x := Yx.t]$, and that is contextually closed for weak contexts (i.e., contexts formed without a $\lambda$-abstraction constructor). Its reflexive, transitive closure is denoted $(>_w^*)$.

Definition 7.7. We denote $\mathcal{W} \subseteq \llbracket \Lambda \rrbracket$ the set of all the pure terms that are strongly normalising for the $(>_w)$ 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 $(>_w)$.

Using the set $\mathcal{W}$ we can define a notion of saturated set, as well as a set $\mathcal{W}_0$ like in Section 6. We are then able to prove corresponding lemmas using the same techniques.

Definition 7.8. A set of pure terms $\Phi \subseteq \llbracket \Lambda \rrbracket$ is said to be weakly saturated if it satisfies the conditions of Definition 6.3, where every occurrence of $\mathcal{N}$ is replaced by $\mathcal{W}$, plus the following condition.

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

Definition 7.9. The set $\mathcal{W}_0$ is defined as $\mathcal{N}_0$ (see Definition 6.6), but using $\mathcal{W}$ instead of $\mathcal{N}$. We denote by $\overline{\mathcal{W}_0}$ the least weakly saturated set containing $\mathcal{W}_0$.

Example 7.10. The term $y \ (Yr. \lambda x. r)$ is in $\mathcal{W}_0$, but not in $\mathcal{N}_0$.

Using the above definitions, we can obtain similar properties as in Section 6. This is mainly due to the fact that the proof of theses lemmas do not considers reductions which are allowed for $(>)$ but forbidden for $(>_w)$. We will first show that $\mathcal{W}$ is weakly saturated, but this requires a small lemma that was immediate in Section 6.

Lemma 7.11. For any terms $t \in \Lambda$ and $u \in \mathcal{W}$ such that $u >_w^* u'$, if $t[x := u'] \in \mathcal{W}$ has an infinite weak reduction then $t[x := u]$ also has one.

Proof. We reason coinductively. We first distinguish the occurrences of $x$ in $t$ that appear under an abstraction by denoting them $x_0$, while denoting the others $x_1$. We hence obtain $t[x := u] = (t[x_1 := u])[x_0 := u]$ and $t[x := u'] = (t[x_1 := u'])[x_0 := u']$. Let us now consider the first step of an infinite reduction of $t[x := u'] >_w t'[x_0 := u']$, with $t[x_1 := u'] >_w t'$ (there cannot be any weak reduction for the occurrences of $u'$ replacing $x_0$). We thus have $t[x := u] = (t[x_1 := u])[x_0 := u] >_w$
(\(t[x_1 := u']\))[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.** The proof is exactly the same as that of Lemma 6.5, except for condition (1). In this case, we need to prove that if \(H[t[x := u]] \in \mathcal{W}\) and \(u \in \mathcal{W}\), then \(H[(\lambda x.t) u] \in \mathcal{W}\). We thus suppose, by contradiction, that \(H[(\lambda x.t) u] \in \mathcal{W}\) has an infinite weak reduction. Such a reduction must start with \(H[(\lambda x.t) u] \succ^*_w H'(\lambda x.t) u' \succ^*_w H'[t[x := u']]\), where \(H \succ^*_w H'\) and \(u \succ^*_w u'\). It can hence be transformed into \(H[t[x := u]] \succ^*_w H'[t[x := u]]\) and we can use Lemma 7.11 to obtain an infinite reduction of \(H'[t[x := u']]\) from the infinite reduction of \(H'[t[x := u']]\). This gives a contradiction with \(H[t[x := u]] \in \mathcal{W}\). □

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

**Definition 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 by replacing every occurrence of \(\mathcal{N}\) and \(\overline{\mathcal{N}}_0\) with \(\mathcal{W}\) and \(\overline{\mathcal{W}}_0\) respectively. The new syntactic elements are interpreted as follows.

\[
\begin{align*}
[Yx.t] &= Yx.[t] \\
[\forall \alpha.A] &= \bigcap_{\alpha \in O}[A[\alpha := 0]] \\
[\exists \alpha.A] &= \bigcup_{\alpha \in O}[A[\alpha := 0]] \\
A \land \kappa &= \begin{cases} [A] & \text{if } [\kappa] \neq 0 \\ \mathcal{W}_0 & \text{otherwise} \end{cases} \\
A \lor \kappa &= \begin{cases} [A] & \text{if } [\kappa] \neq 0 \\ \mathcal{W} & \text{otherwise} \end{cases}
\end{align*}
\]

**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 as for Theorem 6.20. The cases for the four new type constructors are immediate by induction hypothesis. □

We will now give the adequacy lemma for the new system, which will be similar to that of Section 6. For this reason, we will not state all require lemmas (e.g., the equivalent of Theorem 6.15), as their proof does not change much. We however need a small lemma for handling the weak normality condition in some of our new typing rules.

**Lemma 7.16.** If \(t \in \Lambda\) be a term such that \(t \downarrow\) (i.e., \(t\) is weakly normal), then \([t] \in \mathcal{W}\).

**Proof.** Immediate by induction, using Theorem 7.15 when \(t = \varepsilon_{x \in A}(t \not\in B)\). □

**Theorem 7.17.** Let \(\gamma\) be an ordinal context such that \([\gamma] > 0\) 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, using Theorem 3.17. For the local subtyping rules of Figure 8, the proof remains essentially the same. Occurrences of \(\mathcal{N}\) and \(\overline{\mathcal{N}}_0\) need to be replaced by \(\mathcal{W}\) and \(\overline{\mathcal{W}}_0\), and lemmas need to be modified according to the new definitions (their proofs are mostly unchanged). Similarly, the cases of the \((\varepsilon)\) and \((\times_c)\) typing rules are unchanged.
(up to the transmission of the context in the induction hypothesis). Hence, we only consider the cases of the remaining typing rules of Figure 12, and the local subtyping rules of Figure 13.

\((\rightarrow_1)\) We need to show \([\lambda x.t] \in [C]\). According to the first induction hypothesis, it is enough to show \([\lambda x.t] \in [(A \rightarrow B) \vee \gamma_0]\). If there is \(\kappa \in \gamma_0\) such that \(\lceil \kappa \rceil = 0\) then we have \(\lceil (A \rightarrow B) \vee \gamma_0 \rceil = \mathcal{W}\) and we can conclude immediately by Lemma 7.16. We can thus assume that \(\lceil \kappa \rceil \neq 0\) for all \(\kappa \in \gamma_0\). 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 can conclude \([\lambda x.t] \in [A \rightarrow B] = [(A \rightarrow B) \vee \gamma_0]\) since we know that \([A \rightarrow B]\) is weakly saturated.

\((\rightarrow_2)\) We need to show \([t \ u] \in [B]\). By the first induction hypothesis \([t] \in [(A \rightarrow B) \wedge \gamma_0]\). If \(\lceil \kappa \rceil = 0\) for some \(\kappa \in \gamma_0\), then \([(A \rightarrow B) \wedge \gamma_0] = \mathcal{W}_0\) and thus we have \([t] \in \mathcal{W}_0\), which implies \([t \ u] \in \mathcal{W}_0 \subseteq [B]\). Otherwise, we have \(\lceil \kappa \rceil \neq 0\) for all \(\kappa \in \gamma_0\), and thus \([t] \in [A \rightarrow B]\). We can hence use the second induction hypothesis to get \([u] \in [A]\) and conclude by definition of \([A \rightarrow B]\).

\((\times_1)\) We only need to prove \([\{l_i : A_i\}_{i \in I}] \in [(\{l_i : A_i\}_{i \in I}) \vee \gamma_0]\) according to the first induction hypothesis. If \(\lceil \kappa \rceil = 0\) for some \(\kappa \in \gamma_0\) and if \(l_i \downarrow\) for all \(i \in I\), then we can conclude immediately using Lemma 7.16 as \([\{l_i : A_i\}_{i \in I}] \vee \gamma_0\) = \(\mathcal{W}\). Otherwise, we can use the remaining induction hypotheses to get \([l_i] \in [A_i]\) for all \(i \in I\). From this we obtain \([\{l_i = t_i\}_{i \in I}] \in \{\{l_i : A_i\}_{i \in I}\}\) using weak saturation. We can then conclude since \([\{l_i = t_i\}_{i \in I}] \vee \gamma_0\) = \([\{l_i : A_i\}_{i \in I}]\) by definition.

\((\times_2)\) We only need to prove \([C t] \in [(C \circ A) \vee \gamma_0]\) according to the first induction hypothesis. It \(\lceil \kappa \rceil = 0\) for some \(\kappa \in \gamma_0\) and if \(t\) is weakly normal, then we can conclude immediately using Lemma 7.16. Otherwise, we must have \(\lceil \kappa \rceil \neq 0\) for all \(\kappa \in \gamma_0\). Therefore, we can use the second induction hypothesis to get \([t] \in [A]\). From this we obtain \([C t] \in [(C \circ A)]\) by saturation.

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

\((\forall)\) By definition, we have \((Y x.t) \vdash_w t[x := Y x.t]\). As a consequence, the validity of the rule follows from the weak saturation condition (6) on \([A]\).

\((\forall^0)\) If \([t] \in [\forall \alpha.A]\) then \([t] \in [A[\alpha := \kappa]] = [A[\alpha := \kappa]]\) by the substitution lemma. Hence, the induction hypothesis gives \([t] \in [B]\).

\((\forall^0)\) 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 \([t] \notin [B[x := \varepsilon_{\alpha \in O}(o \notin B)]\). We hence obtain a contradiction with \([t] \in [A]\) using the induction hypothesis.

\((\exists)\) Similar to the \((\forall^0)\) case.

\((\exists^0)\) Similar to the \((\forall^0)\) case.

\((\land_1)\) We assume that \([t] \in [A \land \kappa]\). If \(\lceil \kappa \rceil = 0\) then \([A \land \kappa] = \mathcal{W}_0\) and hence \([t] \in [B]\). If \(\lceil \kappa \rceil \neq 0\) then \([A \land \kappa] = \mathcal{A}\) and we can thus conclude by induction hypothesis.

\((\land_2)\) Since \(\kappa \in \gamma\) we know that \(\lceil \kappa \rceil \neq 0\) and thus we have \([A \land \kappa] = [A]\). We can thus conclude by induction hypothesis.

\((\lor)\) Similar to the \((\land_1)\) case.

\((\lor^0)\) Similar to the \((\lor)\) case.

\(\Box\)
\[ F(A, X) = \{ \text{Nil} \mid \text{Cons of } \{ \text{hd} : A; \text{tl} : X \} \} \]
\[ \mathcal{L}_\alpha(A) = \mu X. F(A, X) \]
\[ \mathcal{L}(A) = \mathcal{L}_\infty(A) \]
\[ \text{map} : \forall A B. \forall \alpha. (A \rightarrow B) \rightarrow \mathcal{L}_\alpha(A) \rightarrow \mathcal{L}_\alpha(B) \]
\[ = \text{Ymap}. \lambda f l. [l] \mapsto [l] \quad x :: l \rightarrow f \ x :: \text{map} \ f \ l \]
\[ \text{map}_2 : \forall A B C. \forall \alpha. (A \rightarrow B \rightarrow C) \rightarrow \mathcal{L}_\alpha(A) \rightarrow \mathcal{L}_\alpha(B) \rightarrow \mathcal{L}_\alpha(C) \]
\[ = \text{Ymap}_2. \lambda f \ l_1 \ l_2. [l_1] \mapsto [l_1] \quad x :: l_1 \rightarrow [l_2] \mapsto [l_2] \quad y :: l_2 \rightarrow f \ x :: \text{map}_2 \ f \ l_1 \ l_2 \]
\[ \text{flatten} : \forall A. \mathcal{L}(\mathcal{L}(A)) \rightarrow \mathcal{L}(A) \]
\[ = \text{Yflatten}. \lambda l_3. [l_3] \mapsto [l_3] \quad l :: l_3 \rightarrow [l] \mapsto [l] \quad x :: l \rightarrow \text{flatten} \ l_3 \quad x :: l \rightarrow x :: \text{flatten} \ (l :: l_3) \]
\[ \text{insert} : \forall \alpha. \forall A. (A \rightarrow A \rightarrow \mathcal{B}) \rightarrow A \rightarrow \mathcal{L}_\alpha(A) \rightarrow \mathcal{L}_{\alpha + 1}(A) \]
\[ = \text{Yinsert}. \lambda f \ a \ l. [l] \mapsto [l] \quad x :: l \rightarrow [f \ a \ x \mid T \rightarrow a :: l \mid F \rightarrow x :: \text{insert} \ f \ a \ l] \]
\[ \text{sort} : \forall \alpha. \forall A. (A \rightarrow A \rightarrow \mathcal{B}) \rightarrow \mathcal{L}_\alpha(A) \rightarrow \mathcal{L}_\alpha(A) \]
\[ = \text{Ysort}. \lambda f l. [l] \mapsto [l] \quad x :: l \rightarrow \text{insert} \ f \ x \ (\text{sort} \ f \ l) \]

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

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

**Proof.** The proofs are similar to those of Theorems 6.25, 6.27 and 6.24 respectively (using the results of the current section). \[\square\]

**8 TERMINATING EXAMPLES**

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

The type of lists of size \(\alpha\) given at the top of Figure 16 is straightforward. 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\textsubscript{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\textsubscript{2} on input lists of unrelated sizes, while still preserving size information about the result. A more precise and useful type for map\textsubscript{2} would require extending our syntactic ordinals with a min symbol. Indeed, we could then use the type \(\forall A B C. \forall \alpha \beta. (A \rightarrow B \rightarrow C) \rightarrow \mathcal{L}_\alpha(A) \rightarrow \mathcal{L}_\beta(B) \rightarrow \mathcal{L}_{\text{min}(\alpha, \beta)}(C)\). Nonetheless, it is important to note that the types of map and map\textsubscript{2} are subtypes of their usual...
types (with no size information). For example, we can derive

\[ \forall A B. (A \rightarrow B) \rightarrow L_0(A) \rightarrow L_0(B) \subset \forall A B. (A \rightarrow B) \rightarrow L(A) \rightarrow L(B) \]

in our system. As a consequence, the map and map₂ functions of Figure 16 are suitable for all applications. In particular, we do not need to provide two different versions (one with size information, and one without).

We will now consider the flatten function, which is also given in Figure 16. On this particular example, proving termination requires unrolling the fixpoint twice. Indeed, if we only unroll it once then our algorithm infers the general abstract sequent \( \forall \alpha_0, \alpha_1 (\vdash f : \forall A. L_{\alpha_1} (L_\alpha (A)) \rightarrow L(A)) \), 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 call graph of the function, and in particular the size change matrices labeling its edges.

\((f \rightarrow f)\) The loop on \( f \) corresponds to the first recursive call. The \( 2 \times 2 \) matrix is justified because in this call the size of the inner list \( \alpha_0 \) is constant, while \( \alpha_1 \) decreases.

\((f \rightarrow g)\) The edge from \( f \) to \( g \) represents the definition of \( g \) inside \( f \), which must be seen as \( f \) calling \( g \). In this call, the first line of the matrix is justified by \( \beta_0 < \alpha_1 \) because \( \beta_0 \) is the size of the tail of the outer list. The second line is justified because \( \beta_1 \), the size of the inner list, is equal to \( \alpha_0 \). The last line is justified because \( \beta_2 \), the size of the first element of the outer list decreases (it is smaller than \( \alpha_0 \)).

\((g \rightarrow g)\) The loop on \( g \) corresponds to the last recursive call, where \( \beta_0 \) and \( \beta_1 \) are constant (which justifies the first two lines). The first element of the list is decreasing, so \( \beta_2 \) decreases. Moreover,
\[
\text{AList}(A) = \mu X. [\text{Nil} \mid \text{Cons of } \{ \text{hd} : A; \text{tl} : X \} \mid \text{App of } \{ \text{left} : X; \text{right} : X \}]
\]

\[
\text{fromList} : \forall A. \text{List}(A) \rightarrow \text{AList}(A)
= \lambda l. l
\]

\[
\text{toList} : \forall A. \text{AList}(A) \rightarrow \text{List}(A)
= Y \text{toList}. \Lambda l. \begin{cases}
\text{[]} \rightarrow \text{[]}

\text{e : } l \rightarrow \text{e : } \text{toList } l

\text{App\{left = l; right = r\} } \rightarrow \text{append (toList } l\text{) (toList } r\text{)}
\end{cases}
\]

Fig. 18. Append lists as a supertype of lists.

as we keep in the general abstract sequent the information that \(\beta_2 < \beta_1\), we also have a \(-1\) in the middle of the last line.

\((g \rightarrow f)\) Finally, the edge from \(g\) to \(f\) corresponds to the third recursive call where we have \(\alpha_0 = \beta_1\), \(\alpha_1 = \beta_0\) and \(\beta_2\) become useless (hence the two \(\infty\) on the last column).

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

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

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

To conclude this section, we will now give an example mixing inductive and coinductive types. We consider the type of streams \(S(A)\) and the type of filter on streams \(F\) defined at the top of Figure 19. 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 \(F\), the inner type \(\mu Y. (\{\} \rightarrow \text{R of } Y \mid \text{K of } X)\) imposes that we can only have finitely many \(R\) constructors between \(K\) constructors. As a filter must contain infinitely many \(K\) constructors, this ensures the productivity of the filter function, applying a filter to a stream, and the cmp function composing two filters.

As in the example of the flatten function on lists, both filter and cmp require some unrolling. To avoid this, we may replace the type \(F\) with \(F' = \mu Y. (\{\} \rightarrow \text{R of } Y \mid \text{K of } F)\). Indeed, although \(F \subset F'\) and \(F' \subset F\) are both derivable, \(F'\) carries an ordinal representing the initial number of \(R\) constructors in the type. The call-graph for cmp is given in Figure 20 and gives an example of a non trivial instance of the size change principle.

Note also that \(F\) 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 [36]. They contain, for example, the GCD function for binary natural numbers, and the basic operations for exact real arithmetic (using the signed digits representation). In particular, all of these examples are proved terminating by our implementation.

9 LIST OF EXAMPLES

We now shortly list all worth mentioning examples we have tested with our prototype. All the examples are in the lib and tests folders of our prototype. They represent a total of more than 5000 lines.

Standard inductive data types and their usual functions boolean, option type and coproduct in prelude.typ, unary natural numbers in nat.typ, unary integers with unary naturals as subtype in int.typ, lists in list.typ and append lists in applist.typ.

Finite map both as association lists in assoc.typ and 2-3 trees in tree23.typ.

Set as unbalanced binary search tree in set.typ.

Stream without internal state in stream.typ and with an existentially quantified internal state in state_stream.typ.

State monad for arrays in state_array.typ.

Trie data structure in trie.typ.

Signed digit representation of real numbers as in [8] in real.typ. It is worth noticing that it is a good stress test for the termination checker on coinductive data types.

Sorting algorithm: insertion sort in insert_sort.typ (with a type ensuring size preservation), quick sort in quick_sort.typ and heap sort in heap_sort.typ.
Church’s encoding of usual data types including boolean in church/bool.typ, unary natural numbers in church/nat.typ, pairs, triples and coproduct in church/data.typ, lists in church/list.typ, error monad in church/error.typ, state monad in church/state.typ and streams in church/stream.typ.

René David’s infimum of system F that computes the minimum of two unary Church’s numerals n and m in time $O(\min(n, m) \ln(\min(n, m)))$ [17].

Scott’s encoding of (co)inductive data types including unary natural numbers in scott/nat.typ, lists in scott/list.typ, streams in scott/stream.typ, binary naturals in scott/natbin.typ, binary trees in scott/tree.typ and a variant and the encoding of unary natural numbers using records in scott/nat_as_record.typ.

Other tests for the termination checker with various tests in size.typ, lazy natural numbers in lazy_nat.typ, functions that permutes their arguments in permute.typ, composition and application of filters on streams in stream_filter.typ, an example of terminating...
function with a non equicontinuous type in hard.typ and a few examples that fails in the tests folder.

**Ordinal representation** including addition in ordinal.typ. It contains two representations requiring a large ordinal (larger than $\omega$) for the convergence of the semantics of the inductive type. We provide two additions that pass the termination checker.

**$\lambda$-calculus implementation** with a parametric higher-order abstract syntax for pure $\lambda$-calculus in lambda.typ and simply typed $\lambda$-calculus using Debruijn’s indices in simply.typ.

**Red black trees** as a subtype of binary trees in tree.typ.

**Tests for dot notation** really using dot notation in dotproj.typ and using the “such that” notation in dotprojeps.typ

## 10 TYPE-CHECKING ALGORITHM

Our system can be implemented by transforming the deduction rule systems given in this paper into recursive functions. This can be done relatively easily because the system is mostly syntax-directed. For instance, only one typing rule applies for each term constructor, and at most two subtyping rules apply for each pair of type constructors. It is easy to see that when two subtyping rules may apply (one left rule and one right rule), then they commute (e.g., quantifier rules). This is due to the fact that they do not modify the term carried by the judgment, and that choice operators are constructed using only the term and the type on the side where it is applied. This means that the order in which such rules are applied does not matter. Moreover, if the rule for implication, product or sum can be applied, then it is easy to see that no other rule can be applied (except generalisation).

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

Nonetheless, several subtle details need further discussion. We will here give some guidelines explaining parts of our implementation. We encourage the reader to look at the code of our prototype [36], which should be relatively accessible (at least to readers familiar with the implementation of type systems). According to the previous remarks, the only implementation freedom is in the management of the rules introducing unknown types or ordinals (namely $(\forall l), (\exists r), (\forall o l), (\exists o r), (\mu r)$ and $(\nu l)$), in the management of the ordinal contexts with the $A \land \gamma$ and $A \lor \gamma$ connective, and in the construction of circular typing and local subtyping proofs.

### Unification variables.

For handling unknown types and ordinals in subtyping, the natural solution is to extend their syntax with a set of unification variables. In types, we will use the letters $U$ and $V$ to denote unification variables, which correspond to unknown types until their value is inferred. In our prototype implementation [36], unification variables are handled as follows.

- If we encounter $\gamma \vdash t \in U \subseteq U$ then we use reflexivity.
- If we encounter $\gamma \vdash t \in U \subseteq V$, then we set $U := V$.
- If we encounter $\gamma \vdash t \in U \subseteq A$ or $\gamma \vdash t \in A \subseteq U$, then we decide that $U$ is equal to $A$, provided that it does not occur in $A$. Note that it is essential to check occurrence inside choice operators for them to be well-defined (i.e., not cyclic). Moreover, when $U$ occurs only positively in $A$ we may use $\mu X. A[U := X]$ as a definition for $U$, thus allowing the system to build new recursive types.
In fact, this approach is a bit too naive in the case where we have a projection \( t.l_k \) and the type of \( t \) is a unification variable. Indeed, it is usually not sufficient to fix the type of \( t \) to be a record type with only the field \( l_k \) (the dual problem arises with variants). To solve this issue, our unification variables carry a state keeping track of projected fields (or constructed variants). The state of a unification variable is initialised or updated when we encounter \( \gamma \vdash t \in U \subseteq \{l_1 : A_1, \ldots, l_n : A_n, \ldots \} \) or \( \gamma \vdash t \in [C_1 of A_1, \ldots, C_n of A_n] \subseteq U \). This state can be seen as a subtyping constraint (upper bound for record types, lower bound for variant types) which is delayed until we have a subtyping constraint on the other side.

Unification variables are also required for syntactic ordinals to handle the \((\mu_r), (\nu_l), (\nu_r^o)\) and \((\exists^o)\) rules. In syntactic ordinals, we will use the letters \( O \) and \( P \) to denote unifications variables. As for types, an ordinal unification variable \( O \) may carry constraints like \( \tau \leq \kappa \), to delay instantiation until we have a constraint \( O \leq \kappa \). 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 define \( O \) to be the first ordinal in \( \gamma \) satisfying the constraints on \( O \). If there is none, then we instantiate it with the successor of a unification variable or with \( \infty \). We do this because we must fail if there is no positive solution for \( O \). Otherwise, the subtyping procedure would often loop by building decreasing chains of unification variables.

**Circular subtyping proofs.**

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

Note that when there are no quantifiers, only a finite number of distinct general abstract sequents can be produced, thus implying the termination of our algorithm. Indeed, when when proving a subtyping judgement \( \gamma \vdash t \in A \subseteq B \), 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. 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 \( \mu \)-calculus [32]. 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 happen very often in practice.

**Circular typing proofs.**

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

**Breadth-first search for typing fixpoint.**

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

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

**Generalisation and unification variables.**

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

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

**Dealing with type errors**

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

For readability, it is important to note that it is never required to display choice operators in full. Indeed, we can limit ourselves to the name of the variable they bind, and the position of the variable it was substituted to in the source code. Note however that the error messages of the current prototype are not optimal. They have been optimised for the debugging of the prototype itself rather than for debugging programs written using the prototype. We believe that we could
C(O, M) = \{dom : M → O; cod : M → O; cmp : M → M → M\}

\Cat = \exists O M.C(O, M)

dual : \Cat → \Cat

= \lambda c. \begin{cases} 
    \text{dom} : c.M → c.O = c.cod; \\
    \text{cod} : c.M → c.O = c.dom; \\
    \text{cmp} : c.M → c.M → c.M = \lambda x y.c.cmp y x
\end{cases}

dual2 : \Cat → \Cat

= \lambda c. \text{let } O, M \text{ such that } c : \Cat(O, M) \text{ in }

\begin{cases} 
    \text{dom} : M → O = c.cod; \\
    \text{cod} : M → O = c.dom; \\
    \text{cmp} : M → M → M = \lambda x y.c.cmp y x
\end{cases}

Fig. 21. Example involving dot projection (dual category).

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

11 TYPE ANNOTATIONS AND DOT NOTATION.

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

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

\[
\vdash t : A \\
\vdash t \in A \subseteq B \\
\vdash t : A : B
\]

However, such type annotations are often required to reference bound type variables, and a type
abstraction constructor \( \Lambda X. t \) 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 \text{ such that } x : A(X) \text{ in }

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

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

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

Moreover, this kind of annotations may be used to define dot notation on existential types. It
may be used to replace the usual dot notation for abstract types. Indeed, if a \( \lambda \)-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 \( y \vdash t \in A(X_0, Y_0) \subseteq \exists X \exists Y A(X, Y) \). This will leads to \( X_0 = \varepsilon X t : \exists Y A(X, Y) \) and
Practical Subtyping for System F

\[ Y_0 = \epsilon Y t : A(X_0, Y) \] Yet, this notation style is too heavy and in this particular case, we prefer writing \( x.X \) and \( x.Y \), which rely on the name of bound variables to build the same witnesses as above from the type of \( x \), or more precisely from the type of the term witness that will be substituted to \( x \). It is important to remark that the implementation never needs to rename a bound variable because we substitute closed terms, types or ordinals to variables and renaming is never necessary in this case. As an example, we can define a type for categories using two abstract types \( O \) and \( M \) for objects and morphisms. We can then use both ways to annotate the definition of a function "dual" computing the opposite of a category (see Figure 21).

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

12 PERSPECTIVES AND FUTURE WORK

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

**Higher-order types.**

In our system, only types and ordinals can be quantified over. We had to introduce second order unification variables and the implementation might be more natural with higher-order types. The main difficulty for extending our system to higher-order is purely practical. The handling of unification variables needs to be generalised into a form of higher-order pattern matching. However, our system allows us to avoid computing the variance of higher-order expressions (which is not completely trivial), thanks to the absence of syntactic covariance condition on our inductive and coinductive types.

**Dependent types and proofs of programs.**

One of our motivations for this work is the integration of subtyping to the realisability models defined in a previous work by Rodolphe Lepigre [37]. To achieve this goal, the system needs to be extended with a first-order layer having terms as individuals. Two new type constructors \( t \in A \) (singleton types) and \( A \upharpoonright t = u \) (meaning \( A \) when \( t \) and \( u \) are observationally equal and \( \forall X.X \) otherwise) are then required to encode dependent products and program specifications. These two ingredients would be a first step toward program proving in our system.

**Extensible sums and products.**

The proposed system is relatively expressive, however it lacks flexibility for records and pattern-matching. A form of inheritance allowing extensible records and sums is desirable. Moreover, features like record opening are required to recover the full power of ML modules and functors. We also expect that such a feature will allow for a better type inference, and thus simplify the development of complex programs.

**Completeness without quantifiers.**

Our algorithm seems terminating for the fragment without \( \forall \) and \( \exists \) quantifiers. We are actually able to prove its completeness if we also remove the function type, but a few problems remain when dealing with arrow types, mainly the mere sense of completeness. Various possibilities exist, for instance depending on whether we want to have \( \vdash A \subset ([] \to B) \) for any types \( A \) and \( B \).
A larger complete Subsystem.

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

REFERENCES

Practical Subtyping for System F


[34] Didier Le Botlan and Didier Rémy. MLF: Raising ML to the power of system $\mathrm{F}$. *SIGPLAN Not.*, 35(9):27–38, August 2003.


[58] Christoph Sprenger and Mads Dam. A note on global induction mechanisms in a $\mu$-calculus with explicit approximations, 1999.


Received July 2017; revised March 2018