SEMI-CONTINUOUS SIZED TYPES AND TERMINATION

ANDREAS ABEL

Institut für Informatik, Ludwig-Maximilians-Universität München
e-mail address: abel@tcs.ifi.lmu.de

ABSTRACT. A type-based approach to termination uses sized types: an ordinal bound for
the size of a data structure is stored in its type. A recursive function over a sized type
is accepted if it is visible in the type system that recursive calls occur just at a smaller
size. This approach is only sound if the type of the recursive function is admissible, i.e.,
depends on the size index in a certain way. To explore the space of admissible functions in
the presence of higher-kinded data types and impredicative polymorphism, a semantics is
developed where sized types are interpreted as functions from ordinals into sets of strongly
normalizing terms. It is shown that upper semi-continuity of such functions is a sufficient
semantic criterion for admissibility. To provide a syntactical criterion, a calculus for semi-
continuous functions is developed.

1. INTRODUCTION

Termination of computer programs has received continuous interest in the history of
computer science, and classical applications are total correctness and termination of par-
tial evaluation. In languages with a notion of computation on the type-level, such as
dependently-typed languages or rich typed intermediate languages in compilers [CW99],
termination of expressions that compute a type is required for type checking and type sound-
ness. Further, theorem provers that are based on the Curry-Howard Isomorphism and offer
a functional programming language to write down proofs usually reject non-terminating pro-
grams to ensure consistency. Since the pioneering work of Mendler [Men87], termination
analysis has been combined with typing, with much success for strongly-typed languages
[HPS96, ACG98, Gim98, Xi01, BFG+04, Bla04]. The resulting technique, *type-based termi-
nation checking*, has several advantages over a purely syntactical termination analysis: (1) It
is *robust* w.r.t. small changes of the analyzed program, since it is working on an abstraction
of the program: its type. So if the reformulation of a program (e.g., by introducing a redex)
still can be assigned the same sized type, it automatically passes the termination check. (2)

1

In design and justification, type-based termination rests on a technology extensively studied for several decades: types. (3) Type-based termination is essentially a refinement of the typing rules for recursion and for introduction and elimination of data. This is *orthogonal* to other language constructs, like variants, records, and modules. Thus, a language can be easily enriched by such constructs without change to the termination checker. This is not true if termination checking is a separate static analysis. Orthogonality has an especially pleasing effect: (4) Type-based termination scales to *higher-order functions* and *polymorphism*. (5) Last but not least, it effortlessly creates a termination *certificate*, which is just the typing derivation.

Type-based termination especially plays its strength when combined with higher-order datatypes and higher-rank polymorphism, i.e., occurrence of $\forall$ to the left of an arrow. Let us see an example. We consider the type of generalized rose trees $\mathsf{GRose}\,F\,A$ parameterized by an element type A and the branching type F. It is given by two constructors:

$$\begin{aligned}
\mathsf{leaf} \quad &: \quad \mathsf{GRose}\,F\,A \\
\mathsf{node} \quad &: \quad A \to F\,(\mathsf{GRose}\,F\,A) \to \mathsf{GRose}\,F\,A
\end{aligned}$$

Generalized rose trees are either a leaf or a $\mathsf{node}\,a\,fr$ of a label a of type A and a collection of subtrees fr of type $F\,(\mathsf{GRose}\,F\,A)$. Instances of generalized rose trees are binary trees ($FA = A \times A$), finitely branching trees ($FA = \mathsf{List}\,A$), or infinitely branching trees ($FA = \mathsf{Nat} \to A$). Programming a generic equality function for generalized rose trees that is polymorphic in F and A, we will end up with the following equations:

$\mathsf{Eq}\,A = A \to A \to \mathsf{Bool}$

$\mathsf{eqGRose} : (\forall A.\,\mathsf{Eq}\,A \to \mathsf{Eq}\,(FA)) \to \forall A.\,\mathsf{Eq}\,A \to \mathsf{Eq}\,(\mathsf{GRose}\,F\,A)$

$\mathsf{eqGRose}\;eqF\;eqA\;\mathsf{leaf}\;\mathsf{leaf} = \mathsf{true}$

$\mathsf{eqGRose}\;eqF\;eqA\;(\mathsf{node}\;a\;fr)\;(\mathsf{node}\;a'\;fr') = \begin{aligned}[t] &(eqA\;a\;a')\;\wedge \\ &(eqF\;(\mathsf{eqGRose}\;eqF\;eqA)\;fr\;fr')\end{aligned}$

$\mathsf{eqGRose}\;eqF\;eqA\;\_\;\_ = \mathsf{false}$

The generic equality $\mathsf{eqGRose}$ takes two parametric arguments, eqF and eqA. The second one is a placeholder for an equality test for type A, the first one lifts an equality test for an arbitrary type A to an equality test for the type FA. The equality test for generalized rose trees, $\mathsf{eqGRose}\;eqF\;eqA$, is then defined by recursion on the next two arguments. In the case of two nodes we would expect a recursive call, but instead, the function itself is passed as an argument to eqF, one of its own arguments! Nevertheless, $\mathsf{eqGRose}$ is a total function, provided its arguments are total and well-typed. However, with traditional methods, which only take the computational behavior into account, it will be hard to verify termination of $\mathsf{eqGRose}$. This is due to the fact that the polymorphic nature of eqF plays a crucial role. It is easy to find an instance of eqF of the wrong type which makes the program loop. Take, for instance:

$$\begin{aligned}
eqF &: \mathsf{Eq}\,(\mathsf{GRose}\,F\,\mathsf{Nat}) \to \mathsf{Eq}\,(F\,(\mathsf{GRose}\,F\,\mathsf{Nat})) \\
eqF\;eq\;fr\;fr' &= eq\,(\mathsf{node}\,0\,fr)\,(\mathsf{node}\,0\,fr')
\end{aligned}$$

A type-based termination criterion however passes $\mathsf{eqGRose}$ with ease: Consider the indexed type $\mathsf{GRose}^{\imath}\,F\,A$ of generalized rose trees whose height is smaller than $\imath$. The types of the constructors are refined as follows:

$$\begin{aligned}
\mathsf{leaf} \quad &: \quad \forall F \forall A \forall \imath.\;\mathsf{GRose}^{\imath+1}\,F\,A \\
\mathsf{node} \quad &: \quad \forall F \forall A \forall \imath.\;A \to \mathsf{GRose}^{\imath}\,F\,A \to \mathsf{GRose}^{\imath+1}\,F\,A
\end{aligned}$$

When defining eqGRose for trees of height $< \imath + 1$, we may use eqGRose on trees of height $< \imath$. Hence, in the clause for two nodes, term eqGRose eqF eqA has type $\mathsf{Eq}\,(\mathsf{GRose}^{\imath}\,F A)$, and eqF (eqGRose eqF eqA) gets type $\mathsf{Eq}\,(F\,(\mathsf{GRose}^{\imath}\,F A))$, by instantiation of the polymorphic type of eqF. Now it is safe to apply the last expression to fr and fr' which are in $F\,(\mathsf{GRose}^{\imath}\,F A)$, since $\mathsf{node}\,a\,fr$ and $\mathsf{node}\,a'\,fr'$ were assumed to be in $\mathsf{GRose}^{\imath+1}\,F A$.

In essence, type-based termination is a stricter typing of the fixed-point combinator fix which introduces recursion. The unrestricted use, via the typing rule (1), is replaced by a rule with a stronger hypothesis (2):

$$(1)\quad \frac{f : A \to A}{\mathsf{fix}\,f : A} \qquad\qquad (2)\quad \frac{f : \forall \imath.\, A(\imath) \to A(\imath + 1)}{\mathsf{fix}\,f : \forall n.\, A(n)}$$

Soundness of rule (2) can be shown by induction on n. To get started, we need to show $\mathsf{fix}\,f : A(0)$ which requires $A(\imath)$ to be of a special shape, for instance $A(\imath) = \mathsf{GRose}^{\imath}\,F\,B \to C$ (this corresponds to Hughes, Pareto, and Sabry's *bottom check* [HPS96]). Then $A(0)$ denotes functions which have to behave well for all arguments in $\mathsf{GRose}^0\,F\,B$, i.e., for no arguments, since $\mathsf{GRose}^0\,F\,B$ is empty. Trivially, any program fulfills this condition. In the step case, we need to show $\mathsf{fix}\,f : A(n+1)$, but this follows from the equation $\mathsf{fix}\,f = f\,(\mathsf{fix}\,f)$ since $f : A(n) \to A(n+1)$, and $\mathsf{fix}\,f : A(n)$ by induction hypothesis.

In general, the index $\imath$ in $A(\imath)$ will be an *ordinal* number. Ordinals are useful when we want to speak of objects of unbounded size, e.g., generalized rose trees of height $< \omega$ that inhabit the type $\mathsf{GRose}^{\omega}\,F A$. Even more, ordinals are required to denote the height of infinitely branching trees: take generalized rose trees with $F A = \mathsf{Nat} \to A$. Other examples of infinite branching, which come from the area of type-theoretic theorem provers, are the W-type, Brouwer ordinals and the accessibility predicate [PM92].

In the presence of ordinal indices, rule (2) has to be proven sound by transfinite induction. In the case of a limit ordinal λ, we have to infer $\mathsf{fix}\,f : A(\lambda)$ from the induction hypothesis $\mathsf{fix}\,f : \forall \alpha < \lambda.\, A(\alpha)$. This imposes extra conditions on the shape of a so-called *admissible* type A, which are the object of this article. Of course, a monotone A is trivially admissible, but many interesting types for recursive functions are not monotone, like $A(\alpha) = \mathsf{Nat}^{\alpha} \to \mathsf{Nat}^{\alpha} \to \mathsf{Nat}^{\alpha}$ (where Nat^{α} contains the natural numbers $< \alpha$). We will show that all those types $A(\alpha)$ are admissible that are *upper semi-continuous* in α, meaning $\limsup_{\alpha \to \lambda} \mathcal{A}(\alpha) \subseteq \mathcal{A}(\lambda)$ for limit ordinals λ. Function types $C(\alpha) = A(\alpha) \to B(\alpha)$ will be admissible if A is *lower semi-continuous* $(A(\lambda) \subseteq \liminf_{\alpha \to \lambda} \mathcal{A}(\alpha))$ and B is upper semi-continuous. Similar laws will be developed for the other type constructors and put into the form of a kinding system for semi-continuous types.

Before we dive into the mathematics, let us make sure that semi-continuity is relevant for termination. A type which is not upper semi-continuous is $A(\imath) = (\mathsf{Nat}^{\omega} \to \mathsf{Nat}^{\imath}) \to \mathsf{Nat}^{\omega}$ (see Sect. 5). Assuming we can nevertheless use this type for a recursive function, we can construct a loop. First, define successor $\mathsf{succ} : \forall \imath.\, \mathsf{Nat}^{\imath} \to \mathsf{Nat}^{\imath+1}$ and predecessor $\mathsf{pred} : \forall \imath.\, \mathsf{Nat}^{\imath+1} \to \mathsf{Nat}^{\imath}$. Note that the size index is an upper bound and ω is the biggest such bound for the case of natural numbers, thus, we have the subtyping relations $\mathsf{Nat}^{\imath} \leq \mathsf{Nat}^{\imath+1} \leq \cdots \leq \mathsf{Nat}^{\omega} \leq \mathsf{Nat}^{\omega+1} \leq \mathsf{Nat}^{\omega}$.

We make the following definitions:

$$A(\imath) \quad := \quad (\mathsf{Nat}^\omega \to \mathsf{Nat}^\imath) \to \mathsf{Nat}^\omega$$

$$f \quad : \quad \forall \imath.\, A(\imath) \to A(\imath + 1)$$

$$f \quad := \quad \lambda loop \lambda g.\ loop\,(\mathsf{shift}\,g)$$

$$\mathsf{shift} \quad : \quad \forall \imath.\, (\mathsf{Nat}^\omega \to \mathsf{Nat}^{\imath + 1})$$
$$\to \mathsf{Nat}^\omega \to \mathsf{Nat}^\imath$$

$$\mathsf{loop} \quad : \quad \forall \imath.\, A(\imath)$$

$$\mathsf{shift} \quad := \quad \lambda g \lambda n.\, \mathsf{pred}\,(g\,(\mathsf{succ}\,n))$$

$$\mathsf{loop} \quad := \quad \mathsf{fix}\,f$$

Since $\mathsf{Nat}^\omega \to \mathsf{Nat}^0$ is empty, A passes the bottom check. Still, instantiating types to $\mathsf{succ} : \mathsf{Nat}^\omega \to \mathsf{Nat}^\omega$ and $\mathsf{loop} : (\mathsf{Nat}^\omega \to \mathsf{Nat}^\omega) \to \mathsf{Nat}^\omega$ we convince ourselves that the execution of $\mathsf{loop}\,\mathsf{succ}$ indeed runs forever.

1.1. **Related Work and Contribution.** Ensuring termination through typing is quite an old idea, just think of type systems for the λ-calculus like simple types, System F, System F^ω, or the Calculus of Constructions, which all have the normalization property. These systems have been extended by special recursion operators, like primitive recursion in Gödel's T, or the recursors generated for inductive definitions in Type Theory (e. g., in Coq). These recursion operators preserve normalization but limit the definition of recursive functions to special patterns, namely instantiations of the recursion scheme dictated by the recursion operator. Taming general recursion $\mathsf{fix}\,f$ through typing, however, which allows the definition of recursive functions in the intuitive way known from functional programming, is not yet fully explored. Mendler [Men87] pioneered this field; he used a certain polymorphic typing of the functional f to obtain primitive (co)recursive functions over arbitrary datatypes. Amadio and Coupet-Grimal [ACG98] and Giménez [Gim98] developed Mendler's approach further, until a presentation using ordinal-indexed (co)inductive types was found and proven sound by Barthe et al. [BFG$^+$04]. The system $\lambda^\frown$ presented in loc. cit. restricts types $A(\imath)$ of recursive functions to the shape $\mu^\imath F \to C(\imath)$ where the domain must be an inductive type $\mu^\imath F$ indexed by $\imath$ and the codomain a type $C(\imath)$ that is monotonic in $\imath$. This criterion, which has also been described by the author [Abe04], allows for a simple soundness proof in the limit case of the transfinite induction, but excludes interesting types like the considered

$$\mathsf{Eq}\,(\mathsf{GRose}^\imath\,F\,A) = \mathsf{GRose}^\imath\,F\,A \to \mathsf{GRose}^\imath\,F\,A \to \mathsf{Bool}$$

which has an antitonic codomain $C(\imath) = \mathsf{GRose}^\imath\,F\,A \to \mathsf{Bool}$. The author has in previous work widened the criterion, but only for a type system without polymorphism [Abe03]. Other recent works on type-based termination [Bla04, Bla05, BGP05] stick to the restriction of $\lambda^\frown$. Xi [Xi01] uses dependent types and lexicographic measures to ensure termination of recursive programs in a call-by-value language, but his indices are natural numbers instead of ordinals; this excludes infinite objects we are interested in.

Closest to the present work is the sized type system of Hughes, Pareto, and Sabry [HPS96], *Synchronous Haskell* [Par00], which admits ordinal indices up to ω. Index quantifiers as in $\forall \imath.\, A(\imath)$ range over natural numbers, but can be instantiated to ω if $A(\imath)$ is ω-*undershooting.* Sound semantic criteria for ω-undershooting types are already present, but in a somewhat ad-hoc manner. We cast these criteria in the established mathematical framework of semi-continuous functions and provide a syntactical implementation in form of a derivation system. Furthermore, we allow ordinals beyond ω and infinitely branching inductive types that invalidate some criteria for the only finitely branching tree types in *Synchronous Haskell.* Finally, we allow polymorphic recursion, impredicative polymorphism and higher-kinded inductive and coinductive types such as GRose. This article summarizes

the main results of the author's dissertation [Abe06b]. A shorter version has appeared in the CSL'06 proceedings [Abe06c].

1.2. **Contents.** In Section 2 we introduce the syntax of $\widehat{\mathsf{F}_\omega}$, our λ-calculus with higher-kinded polymorphism, recursion over higher-kinded inductive types and corecursion into higher-kinded coinductive types. Static semantics (i.e., typing rules) and dynamic semantics (i.e., reduction rules) are presented there, and we formally express the eqGRose-example from the introduction in $\widehat{\mathsf{F}_\omega}$. In Section 3 we model the types of $\widehat{\mathsf{F}_\omega}$ as saturated sets of strongly normalizing terms in order to show termination of well-typed programs. After these two technical sections we come to the main part of this article: In Section 4 we identify compositional criteria for semi-continuous types and in Section 5 we justify the absence of certain composition schemes by giving counterexamples. These results are put in the form of a calculus for semi-continuous types in Section 6, culminating in syntactic rules for admissible (co)recursion types. We close by giving some practical examples for admissible types.

1.3. **Preliminaries.** We assume that the reader is to some extent acquainted with the higher-order polymorphic lambda-calculus, System F^ω (see Pierce's text book [Pie02]) and has some knowledge of ordinals, inductive types, and strong normalization.

2. Overview of System $\widehat{\mathsf{F}_\omega}$

In this section we introduce $\widehat{\mathsf{F}_\omega}$, an *a posteriori* strongly normalizing extension of System F^ω with higher-kinded inductive and coinductive types and (co)recursion combinators. Figure 1 summarizes the syntactic entities.

2.1. **Type constructors.** We seek to model sized types like $\mathsf{GRose}^i F A$ whose first parameter F is a type *constructor* of kind $* \to *$, meaning that it maps types to types. It is therefore suggestive to take F^ω as basis, which formalizes type constructors of arbitrary kind and, e.g., lays the foundation for the purely functional language Haskell. In the introduction, we have presented GRoses as built from two (data) constructors leaf and node; however, for a theoretic analysis it is more convenient to consider $\mathsf{GRose}\, F A$ as the least fixed-point of the type constructor $\lambda X.\, 1 + (A \times F\, X)$. For this we write

$$\mathsf{GRose}\, F A := \mu \lambda X.\, 1 + (A \times F\, X).$$

Herein, 1 is the unit type and $+$ the disjoint sum. Taking the empty tuple $\langle \rangle : 1$ to be the inhabitant of the unit type and $\mathsf{inl} : A \to (A + B)$ and $\mathsf{inr} : B \to (A + B)$ the two injections into the disjoint sum lets us *define* the original data constructors:

$$
\begin{array}{lcl}
\mathsf{leaf} & : & \mathsf{GRose}\, F A \\
\mathsf{leaf} & := & \mathsf{in}\,(\mathsf{inl}\,\langle\rangle) \\[4pt]
\mathsf{node} & : & A \to F\,(\mathsf{GRose}\, F A) \to \mathsf{GRose}\, F A \\
\mathsf{node} & := & \lambda a \lambda fr.\, \mathsf{in}\,(\mathsf{inr}\,\langle a, fr \rangle)
\end{array}
$$

(The tag in introduces a inductive type, see below.)

Polarities, kinds, constructors, kinding contexts.

$$
\begin{array}{llll}
p & ::= & + \mid - \mid \circ & \text{polarity} \\
\kappa & ::= & * \mid \mathsf{ord} \mid p\kappa \to \kappa' & \text{kind} \\
\kappa_* & ::= & * \mid p\kappa_* \to \kappa'_* & \text{pure kind} \\
a,b,A,B,F,G & ::= & C \mid X \mid \lambda X\!:\!\kappa.\,F \mid F\,G & \text{(type) constructor} \\
C & ::= & 1 \mid + \mid \times \mid \to \mid \forall_\kappa \mid \mu_{\kappa_*} \mid \nu_{\kappa_*} \mid \mathsf{s} \mid \infty & \text{constructor constants} \\
\Delta & ::= & \diamond \mid \Delta, X\!:\!p\kappa & \text{kinding context}
\end{array}
$$

Constructor constants and their kinds ($\kappa \xrightarrow{p} \kappa'$ means $p\kappa \to \kappa'$).

$$
\begin{array}{lll}
1 & : & * & \text{unit type} \\
+ & : & * \xrightarrow{+} * \xrightarrow{+} * & \text{disjoint sum} \\
\times & : & * \xrightarrow{+} * \xrightarrow{+} * & \text{cartesian product} \\
\to & : & * \xrightarrow{-} * \xrightarrow{+} * & \text{function space} \\
\forall_\kappa & : & (\kappa \xrightarrow{\circ} *) \xrightarrow{+} * & \text{quantification} \\
\mu_{\kappa_*} & : & \mathsf{ord} \xrightarrow{+} (\kappa_* \xrightarrow{+} \kappa_*) \xrightarrow{+} \kappa_* & \text{inductive constructors} \\
\nu_{\kappa_*} & : & \mathsf{ord} \xrightarrow{-} (\kappa_* \xrightarrow{+} \kappa_*) \xrightarrow{+} \kappa_* & \text{coinductive constructors} \\
\mathsf{s} & : & \mathsf{ord} \xrightarrow{+} \mathsf{ord} & \text{successor of ordinal} \\
\infty & : & \mathsf{ord} & \text{infinity ordinal}
\end{array}
$$

Objects (terms), values, evaluation frames, typing contexts.

$$
\begin{array}{llll}
r,s,t & ::= & c \mid x \mid \lambda xt \mid r\,s & \text{term} \\
c & ::= & \langle\rangle \mid \mathsf{pair} \mid \mathsf{fst} \mid \mathsf{snd} \mid \mathsf{inl} \mid \mathsf{inr} \mid \mathsf{case} \mid \mathsf{in} \mid \mathsf{out} \mid \mathsf{fix}^\mu_n \mid \mathsf{fix}^\nu_n & \text{constant } (n \in \mathbb{N}) \\
v & ::= & \lambda xt \mid \mathsf{pair}\,t_1\,t_2 \mid \mathsf{inl}\,t \mid \mathsf{inr}\,t \mid \mathsf{in}\,t \mid c \mid \mathsf{pair}\,t \mid \mathsf{fix}^\nabla_n s\,t_{1..m} & \text{value } (m \le n) \\
e(\_) & ::= & \_\,s \mid \mathsf{fst}\,\_ \mid \mathsf{snd}\,\_ \mid \mathsf{case}\,\_ \mid \mathsf{out}\,\_ \mid \mathsf{fix}^\mu_n s\,t_{1..n}\,\_ & \text{evaluation frame} \\
E(\_) & ::= & e_1(\ldots e_n(\_)\ldots) & \text{evaluation context } (n \ge 0) \\
\Gamma & ::= & \diamond \mid \Gamma, x\!:\!A \mid \Gamma, X\!:\!p\kappa & \text{typing context}
\end{array}
$$

Reduction $t \longrightarrow t'$.

$$
\begin{array}{lllcll}
(\lambda xt)\,s & \longrightarrow & [s/x]t & & \mathsf{out}\,(\mathsf{in}\,r) & \longrightarrow\ r \\
\mathsf{fst}\,\langle r,s\rangle & \longrightarrow & r & & \mathsf{fix}^\mu_n s\,t_{1..n}\,(\mathsf{in}\,t) & \longrightarrow\ s\,(\mathsf{fix}^\mu_n s)\,t_{1..n}\,(\mathsf{in}\,t) \\
\mathsf{snd}\,\langle r,s\rangle & \longrightarrow & s & & \mathsf{out}\,(\mathsf{fix}^\nu_n s\,t_{1..n}) & \longrightarrow\ \mathsf{out}\,(s\,(\mathsf{fix}^\nu_n s)\,t_{1..n}) \\
\mathsf{case}\,(\mathsf{inl}\,r) & \longrightarrow & \lambda x\lambda y.\,x\,r & (*) & & \\
\mathsf{case}\,(\mathsf{inr}\,r) & \longrightarrow & \lambda x\lambda y.\,y\,r & (*) & + \text{ closure under all term constructs} &
\end{array}
$$

$(*)\ x,y \notin \mathsf{FV}(r)$.

Figure 1: $\mathsf{F}\widehat{\omega}$: Syntax and operational semantics.

2.2. Polarized kinds. Negative recursive types such as $\mu\lambda X.\,X \to 1$ allow the coding of Y and other fixed-point combinators as pure λ-terms, so one can write recursive programs without special syntax for recursion [Men87]. For our purposes, this is counter-productive—type systems for termination need to identify all uses of recursion. Therefore, we restrict to positive recursive types μH where H is monotone. In the case of GRose, the underlying constructor $H\,X = 1 + (A \times F\,X)$ must be monotone, which is the case if F is monotone. So $\mathsf{GRose}\,F\,A$ is only well-formed for monotone F. To distinguish type constructors by their monotonicity behavior, also called *variance*, we equip function kinds with polarities p [Ste98], which are written before the domain or on top of the arrow. Polarity $+$ denotes co-variant constructors, $-$ contravariant constructors and $\circ$ mixed-variant constructors [DC99]. For instance:

$$
\begin{array}{lcl}
\lambda X.\,X \to 1 & : & * \xrightarrow{-} * \\
\lambda X.\,X \to X & : & * \xrightarrow{\circ} * \\
\lambda X.\,\mathsf{Int} \to (1 + X) & : & * \xrightarrow{+} * \\
\mathsf{GRose} & : & (* \xrightarrow{+} *) \xrightarrow{+} * \xrightarrow{+} *
\end{array}
$$

Abel [Abe06a] and Matthes [AM04] provide more explanation on polarities.

2.3. Sized inductive types. We refine inductive types μF to sized inductive types $\mu^a F$. The first argument, a, to μ, which we usually write as superscript, denotes the upper bound for the height of data represented by terms of the inductive type. The index a is a constructor of kind ord and denotes an ordinal; the relevant ordinal expressions are given by the grammar

$$
a ::= \imath \mid \mathsf{s}\,a \mid \infty
$$

with $\imath$ an ordinal variable.[1] If a actually denotes a finite ordinal (a natural number), then the height is simply the number of data constructors on the longest path in the tree structure of any element of $\mu^a F$. Since a is only an upper bound, $\mu^a F$ is a subtype of $\mu^b F$, written $\mu^a F \le \mu^b F$ for $a \le b$, meaning that μ is covariant in the index argument. Finally, $F \le F'$ implies $\mu^a F \le \mu^a F'$, so we get the kinding

$$
\mu : \mathsf{ord} \xrightarrow{+} (* \xrightarrow{+} *) \xrightarrow{+} *
$$

for the least fixed-point constructor. For the *closure ordinal* ∞, we have

$$
\mu^\infty F = \mu^{\infty+1} F,
$$

where $\infty + 1$ is a shorthand for $\mathsf{s}\infty$, $\mathsf{s} : \mathsf{ord} \xrightarrow{+} \mathsf{ord}$ being the successor on ordinals.

Because ∞ denotes the closure ordinal, the axiom $\mathsf{s}\infty = \infty$ is justified. Equality on type constructors is defined as the least congruent equivalence relation closed under this equation and $\beta\eta$.

At this point, let us stress that the *syntax* of ordinals is extremely simple, hence, equality of types and subtyping is decidable. The user can think of ordinals as of natural numbers with infinity, although they will be interpreted as real ordinals up to a fairly large closure ordinal in Section 3.

---

[1]One could add a constant for the ordinal 0, but for our purposes it is enough that each concrete data structure inhabits $\mu^\infty F$. For checking termination relative sizes are sufficient, which can be expressed using ordinal variables and successor.

Example 2.1 (Some sized types).

Nat	:	$\mathsf{ord} \xrightarrow{+} *$		GRose	:	$\mathsf{ord} \xrightarrow{+} (* \xrightarrow{+} *) \xrightarrow{+} * \xrightarrow{+} *$
Nat	:=	$\lambda\imath.\, \mu^{\imath}\lambda X.\, 1 + X$		GRose	:=	$\lambda\imath\lambda F\lambda A.\, \mu^{\imath}\lambda X.\, 1 + A \times F\,X$
List	:	$\mathsf{ord} \xrightarrow{+} * \xrightarrow{+} *$		Tree	:	$\mathsf{ord} \xrightarrow{+} * \xrightarrow{-} * \xrightarrow{+} *$
List	:=	$\lambda\imath\lambda A.\, \mu^{\imath}\lambda X.\, 1 + A \times X$		Tree	:=	$\lambda\imath\lambda B\lambda A.\, \mathsf{GRose}^{\imath}\,(\lambda X.\, B \to X)\,A$

2.4. Sized coinductive types. Dually to inductive or least fixed-point types μF we have coinductive or greatest fixed-point types νF to model infinite structures. For instance $\mathsf{Stream}\,A = \nu X.\, A \times X$ contains the infinite sequences over A. The dual to the height of an inductive data structure is the *depth* of a coinductive one, i.e., how often one can unwind the structure. So the size a of a sized coinductive type $\nu^{a} F$ is a lower bound on the depth of its inhabitants. Since it is a lower bound, coinductive types are contravariant in their size index:

$$\nu : \mathsf{ord} \xrightarrow{-} (* \xrightarrow{+} *) \xrightarrow{+} *.$$

As for inductive types, the equation $\nu^{\infty} F = \nu^{\infty+1} F$ holds.

Example 2.2 (Sized streams). On a stream in $\mathsf{Stream}^{a} A$ one can safely read off the first a elements.

Stream	:	$\mathsf{ord} \xrightarrow{-} * \xrightarrow{+} *$
Stream	:=	$\lambda\imath\lambda A.\, \nu^{\imath}\lambda X.\, A \times X$

2.5. Heterogeneous datatypes. If we consider not only fixed-point *types*, but also fixed-point *constructors*, we can treat programs involving so-called nested or heterogeneous types. A simple example of a heterogeneous type is the type of powerlists $\mathsf{PList}\,A$ which contains lists of As whose length is a power of two [Hin00a]. The type constructor $\mathsf{PList} : * \xrightarrow{+} *$ can be modeled as $\mu\lambda X\lambda A.\, A + X\,(A \times A)$ which is the least fixed-point of a type constructor of kind $(* \xrightarrow{+} *) \xrightarrow{+} (* \xrightarrow{+} *)$.

Sized heterogeneous types are obtained by simply generalizing μ and ν to

$$\mu_{\kappa} \quad : \quad \mathsf{ord} \xrightarrow{+} (\kappa \xrightarrow{+} \kappa) \xrightarrow{+} \kappa$$
$$\nu_{\kappa} \quad : \quad \mathsf{ord} \xrightarrow{-} (\kappa \xrightarrow{+} \kappa) \xrightarrow{+} \kappa.$$

The kind κ is required to be *pure*, i.e., a kind not mentioning ord, for reasons explained in Section 3.4. All our examples work for pure κ.

Example 2.3 (Sized heterogeneous types).

PList	:	$\mathsf{ord} \xrightarrow{+} * \xrightarrow{+} *$
PList	:=	$\lambda\imath.\, \mu^{\imath}\lambda X\lambda A.\, A + X\,(A \times A)$
Bush	:	$\mathsf{ord} \xrightarrow{+} * \xrightarrow{+} *$
Bush	:=	$\lambda\imath.\, \mu^{\imath}\lambda X\lambda A.\, 1 + A \times X\,(X\,A)$
Lam	:	$\mathsf{ord} \xrightarrow{+} * \xrightarrow{+} *$
Lam	:=	$\lambda\imath.\, \mu^{\imath}\lambda X\lambda A.\, A + X\,A \times X\,A + X\,(1 + A)$

The second type, $\mathsf{Bush}^a\, A$, bushy lists, models finite maps from unlabeled binary trees of height $< a$ into A [Alt01, Hin00b]. The third type, $\mathsf{Lam}^a\, A$, is inhabited by de Bruijn representations of untyped lambda terms of height $< a$ with free variables in A [BP99, AR99].

2.6. **Programs.** The term language of $\mathsf{F}\widehat{_\omega}$ is the λ-calculus plus the standard constants to introduce and eliminate unit (1), sum (+), and product (×) types. We write $\langle t_1, t_2 \rangle$ for $\mathsf{pair}\, t_1\, t_2$. Further, there is folding, in, and unfolding, out, of (co)inductive types. The complete listing of the typing rules can be found in Figure 6 in the appendix, here we discuss the most important ones. Let $\kappa = \vec{p\kappa} \to *$ a pure kind, $F : +\kappa \to \kappa$, $G_i : \kappa_i$ for $1 \le i \le |\vec{\kappa}|$, $a : \mathsf{ord}$, and $\nabla \in \{\mu, \nu\}$, then we have the following (un)folding rules:

$$\text{TY-FOLD} \quad \frac{\Gamma \vdash t : F\,(\nabla_\kappa^a\, F)\,\vec{G}}{\Gamma \vdash \mathsf{in}\, t : \nabla_\kappa^{a+1} F\,\vec{G}} \qquad \text{TY-UNFOLD} \quad \frac{\Gamma \vdash r : \nabla_\kappa^{a+1} F\,\vec{G}}{\Gamma \vdash \mathsf{out}\, r : F\,(\nabla_\kappa^a\, F)\,\vec{G}}$$

Finally, there are fixed-point combinators fix_n^μ and fix_n^ν for each $n \in \mathbb{N}$ on the term level. The term $\mathsf{fix}_n^\mu\, s$ denotes a recursive function with n leading non-recursive arguments; the $n+1$st argument must be of an inductive type. Similarly, $\mathsf{fix}_n^\nu\, s$ is a corecursive function which takes n arguments and produces an inhabitant of a coinductive type. We abbreviate $f\, t_1\, \ldots\, t_n$ by $f\, t_{1..n}$ or $f\,\vec{t}$.

One-step reduction $t \longrightarrow t'$ is defined by the β-reduction axioms given in Figure 1 plus congruence rules. Its transitive closure is denoted by $\longrightarrow^+$, and $\longrightarrow^*$ is the reflexive-transitive closure. Interesting are the reduction rules for recursion and corecursion:

$$\begin{aligned}
\mathsf{fix}_n^\mu\, s\, t_{1..n}\,(\mathsf{in}\, t) &\longrightarrow s\,(\mathsf{fix}_n^\mu\, s)\, t_{1..n}\,(\mathsf{in}\, t) \\
\mathsf{out}\,(\mathsf{fix}_n^\nu\, s\, t_{1..n}) &\longrightarrow \mathsf{out}\,(s\,(\mathsf{fix}_n^\nu\, s)\, t_{1..n})
\end{aligned}$$

A recursive function is only unfolded if its recursive argument is a value, i.e., of the form $\mathsf{in}\, t$. This condition is required to ensure strong normalization; it is present in the work of Mendler [Men87], Giménez [Gim98], Barthe et al. [BFG$^+$04], and the author [Abe04]. Dually, corecursive functions are only unfolded on demand, i.e., in an evaluation context, the matching one being $\mathsf{out}\, \_$.

$$\begin{array}{ll}
p \le p' & \text{polarity ordering} \\
\Delta \vdash F : \kappa & \text{kinding} \\
\Delta \vdash F = F' : \kappa & \text{constructor equality} \\
\Delta \vdash F \le F' : \kappa & \text{higher-order subtyping} \\
t \longrightarrow t' & \text{reduction} \\
\Gamma \vdash t : A & \text{typing} \\
\Gamma \vdash A\ \mathsf{fix}_n^\nabla\text{-adm} & \text{admissible recursion type}
\end{array}$$

Figure 2: $\mathsf{F}\widehat{_\omega}$: Judgements.

Figure 2 lists the basic judgements of $\mathsf{F}_\omega^{\widehat{\,}}$, their rules can be found in the appendix. As pointed out in the introduction, recursion is introduced by the rule

$$\text{TY-REC} \quad \frac{\Gamma \vdash A \; \mathsf{fix}_n^\nabla\text{-adm} \qquad \Gamma \vdash a : \mathsf{ord}}{\Gamma \vdash \mathsf{fix}_n^\nabla : (\forall \imath : \mathsf{ord}.\, A\, \imath \to A\,(\imath + 1)) \to A\, a}.$$

Herein, ∇ stands for μ or ν, and the judgement $A\;\mathsf{fix}_n^\nabla\text{-adm}$ (see Def. 6.3) ensures that type A is admissible for (co)recursion, as discussed in the introduction. In this article, we will find out which types are admissible.

Example 2.4. Now we can code the example from the introduction in $\mathsf{F}_\omega^{\widehat{\,}}$, with a suitable coding of true, false and $\wedge$.

$$
\begin{array}{lll}
\mathsf{eqGRose} & : & (\forall A.\, \mathsf{Eq}\, A \to \mathsf{Eq}\,(F A)) \to \forall A.\, \mathsf{Eq}\, A \to \forall \imath.\, \mathsf{Eq}\,(\mathsf{GRose}^\imath F A) \\
\mathsf{eqGRose} & := & \lambda eqF\, \lambda eqA. \\
& & \quad \mathsf{fix}_0^\mu \lambda eq\, \lambda t_1\, \lambda t_2.\, \mathsf{case}\,(\mathsf{out}\, t_1) \\
& & \qquad (\lambda\_.\, \mathsf{case}\,(\mathsf{out}\, t_2)\,(\lambda\_.\, \mathsf{true})\,(\lambda n_2.\, \mathsf{false})) \\
& & \qquad (\lambda n_1.\, \mathsf{case}\,(\mathsf{out}\, t_2)\,(\lambda\_.\, \mathsf{false}) \\
& & \qquad\quad (\lambda n_2.\, (eqA\,(\mathsf{fst}\, n_1)\,(\mathsf{fst}\, n_2)) \;\wedge \\
& & \qquad\qquad (eqF\; eq\,(\mathsf{snd}\, n_1)\,(\mathsf{snd}\, n_2))))
\end{array}
$$

Typing succeeds, by the following assignment of types to variables:

$$
\begin{array}{lll \qquad lll}
eqF & : & \forall A.\, \mathsf{Eq}\, A \to \mathsf{Eq}\,(F A) \\
eqA & : & \mathsf{Eq}\, A \\
eq & : & \mathsf{Eq}\,(\mathsf{GRose}^\imath F A)
\end{array}
$$

$$
\begin{array}{lll}
t_1, t_2 & : & \mathsf{GRose}^{\imath+1} F A \\
\_ & : & 1 \\
n_1, n_2 & : & A \times F\,(\mathsf{GRose}^\imath F A)
\end{array}
$$

More examples, including programs over heterogeneous types, can be found in the author's thesis [Abe06b].

3. Semantics

Hughes, Pareto, and Sabry [HPS96] give a domain-theoretic semantics of sized types. We, however, follow Barthe et al. [BFG$^+$04] and interpret types as sets of terminating *open* expressions and show that any reduction sequence starting with a well-typed expression converges to a normal form. This is more than showing termination of programs (closed expressions); our results can be applied to partial evaluation and testing term equality in type-theoretic proof assistants.

The material in this section is quite technical, but provides the necessary basis for our considerations in the following sections. The reader may browse it, take a closer look at the interpretation of types (Sec. 3.5) and then continue with Section 4, coming back when necessary.

Let $\mathcal{S}$ denote the set of strongly normalizing terms. We interpret a type A as a semantic type $[\![A]\!] \subseteq \mathcal{S}$, and the function space is defined extensionally:

$$[\![A \to B]\!] = \{r \mid r\, s \in [\![B]\!] \text{ for all } s \in [\![A]\!]\}.$$

As main theorem, we show that given a well-typed term $x_1 : A_1, \ldots x_n : A_n \vdash t : C$ and replacements $s_i \in [\![A_i]\!]$ for each occurring variable x_i, the substitution $[\vec{s}/\vec{x}]t$ inhabits $[\![C]\!]$. The proof proceeds by induction on the typing derivation, and in the λ-case (here simplified)

$$\frac{x : A \vdash t : B}{\vdash \lambda x t : A \to B}$$

it suffices to show $(\lambda xt)\, s \in [\![B]\!]$ for any $s \in [\![A]\!]$. However, by induction hypothesis we know only $[s/x]t \in [\![B]\!]$. We therefore require semantic types to be closed under weak head expansion to make this case go through.

Since we are interested in normalization of open terms, we need to set aforementioned replacements s_i to variables x_i. This is possible if each semantic type contains all variables, which has to be generalized to all neutral terms, i.e. terms $E[x]$ with a variable in evaluation position. These observations motivate our definition of semantic types.

3.1. **Semantic types.** We define *safe* (weak head) reduction $\rhd$ by the following axioms. The idea is that semantic types are closed under $\rhd$-expansion.

$$
\begin{array}{llllll}
(\lambda xt)\, s & \rhd\ [s/x]t & \text{if } s \in \mathcal{S} & \mathsf{case}\,(\mathsf{inl}\, r) & \rhd\ \lambda x \lambda y.\, x\, r & (*) \\
\mathsf{fst}\,(\mathsf{pair}\, r\, s) & \rhd\ r & \text{if } s \in \mathcal{S} & \mathsf{case}\,(\mathsf{inr}\, r) & \rhd\ \lambda x \lambda y.\, y\, r & (*) \\
\mathsf{snd}\,(\mathsf{pair}\, r\, s) & \rhd\ s & \text{if } r \in \mathcal{S} & \mathsf{fix}_n^\mu s\, t_{1..n}\,(\mathsf{in}\, r) & \rhd\ s\,(\mathsf{fix}_n^\mu s)\, t_{1..n}\,(\mathsf{in}\, r) \\
\mathsf{out}\,(\mathsf{in}\, r) & \rhd\ r & & \mathsf{out}\,(\mathsf{fix}_n^\nu s\, t_{1..n}) & \rhd\ \mathsf{out}\,(s\,(\mathsf{fix}_n^\nu s)\, t_{1..n})
\end{array}
$$

Side condition (*): $x, y \notin \mathsf{FV}(r)$. Additionally, we close safe reduction under evaluation contexts and transitivity:

$$
\begin{array}{lll}
E(t) & \rhd\ E(t') & \text{if } t \rhd t' \\
t_1 & \rhd\ t_3 & \text{if } t_1 \rhd t_2 \text{ and } t_2 \rhd t_3
\end{array}
$$

One-step safe reduction is deterministic, hence, if $r \rhd s$ and $r \rhd t$ then either $s = t$ or $s \rhd t$ or $t \rhd s$.

$$
\mathcal{V} := \{v, E(x) \mid v \text{ value}, E \text{ evaluation context}\}
$$

is the set of $\rhd$-normal forms, not counting junk terms like $\mathsf{fst}\,(\lambda xt)$.

The relation is defined such that $\mathcal{S}$ is closed under $\rhd$-expansion, meaning $t \rhd t' \in \mathcal{S}$ implies $t \in \mathcal{S}$. In other words, $\rhd$ used in the expansion direction does not introduce diverging terms. Let $^\rhd\mathcal{A}$ denote the closure of term set $\mathcal{A}$ under $\rhd$-expansion. In general, the *closure* of term set $\mathcal{A}$ is defined as

$$
\overline{\mathcal{A}} = \,^\rhd(\mathcal{A} \cup \{E(x) \mid x \text{ variable}, E(x) \in \mathcal{S}\}).
$$

Closure preserves strong normalization: If $\mathcal{A} \subseteq \mathcal{S}$ then $\overline{\mathcal{A}} \subseteq \mathcal{S}$. A term set is *closed* if $\overline{\mathcal{A}} = \mathcal{A}$. The least closed set is the set of neutral terms $\mathcal{N} := \overline{\emptyset} \neq \emptyset$. Intuitively, a neutral term never reduces to a value, it necessarily has a free variable, and it can be substituted into any term without creating a new redex. A term set $\mathcal{A}$ is *saturated* if $\mathcal{A}$ is closed and $\mathcal{N} \subseteq \mathcal{A} \subseteq \mathcal{S}$ (this makes sure that $\mathcal{A}$ contains all variables). A saturated set is called a *semantic type*.

3.2. **Interpretation of kinds.** When types are interpreted as sets of terms, the easiest interpretation of type constructors are set-theoretical operators on term sets, or as we go higher-order, on operators.

The saturated sets form a complete lattice $[\![*]\!]$ with least element $\bot^* := \mathcal{N}$ and greatest element $\top^* := \mathcal{S}$. It is ordered by inclusion $\sqsubseteq^* := \subseteq$ and has set-theoretic infimum $\inf^* := \bigcap$ and supremum $\sup^* := \bigcup$. Let $[\![\mathsf{ord}]\!] := \mathsf{O}$ where $\mathsf{O} = [0; \top^{\mathsf{ord}}]$ is an initial segment of the set-theoretic ordinals. With the usual ordering on ordinals, O constitutes a complete lattice as well. For lattices $\mathfrak{L}$ and $\mathfrak{L}'$, let $\mathfrak{L} \xrightarrow{+} \mathfrak{L}'$ denote the space of monotone functions from $\mathfrak{L}$ to $\mathfrak{L}'$ and $\mathfrak{L} \xrightarrow{-} \mathfrak{L}'$ the space of antitone ones. The mixed-variant function kind $[\![\circ\kappa \to \kappa']\!]$ is interpreted as set-theoretic function space $[\![\kappa]\!] \to [\![\kappa']\!]$; the covariant function

kind $+\kappa \to \kappa'$ denotes the monotonic function space $[\![\kappa]\!] \xrightarrow{+} [\![\kappa']\!]$ and the contravariant kind $-\kappa \to \kappa'$ the antitonic space $[\![\kappa]\!] \xrightarrow{-} [\![\kappa']\!]$. For all function kinds, ordering is defined pointwise: $\mathcal{F} \sqsubseteq^{p\kappa \to \kappa'} \mathcal{F}' :\Longleftrightarrow \mathcal{F}(\mathcal{G}) \sqsubseteq^{\kappa'} \mathcal{F}'(\mathcal{G})$ for all $\mathcal{G} \in [\![\kappa]\!]$. Similarly, $\bot^{p\kappa \to \kappa'}(\mathcal{G}) := \bot^{\kappa'}$ is defined pointwise, and so are $\top^{p\kappa \to \kappa'}$, $\inf^{p\kappa \to \kappa'}$, and $\sup^{p\kappa \to \kappa'}$.

3.3. **Limits and iteration.** Inductive types $[\![\mu^a F]\!]$ are constructed by iterating the operator $[\![F]\!]$ $[\![a]\!]$-times, starting with the least semantic type $\bot$. At limit ordinals, we take the supremum. If $[\![a]\!]$ is big enough, latest if $[\![a]\!] = \top^{\mathsf{ord}}$, the least fixed-point is reached, but our type system also provides notation for the approximation stages below the fixed-point. For coinductive types, we start with the biggest semantic type $\top$ and take the infimum at limits. It is possible to unify these two forms of iteration, by taking the $\lim \sup$ instead of infimum or supremum at the limits. The notion of $\lim \sup$ and iteration can be defined for arbitrary lattices:

In the following $\lambda \in \mathsf{O}$ will denote a limit ordinal. (We will only consider proper limits, i. e., $\lambda \neq 0$.) For $\mathfrak{L}$ a complete lattice and $f \in \mathsf{O} \to \mathfrak{L}$ we define:

$$
\begin{aligned}
\liminf_{\alpha \to \lambda} f(\alpha) &:= \sup_{\alpha_0 < \lambda} \inf_{\alpha_0 \leq \alpha < \lambda} f(\alpha) \\
\limsup_{\alpha \to \lambda} f(\alpha) &:= \inf_{\alpha_0 < \lambda} \sup_{\alpha_0 \leq \alpha < \lambda} f(\alpha)
\end{aligned}
$$

Using $\inf_\lambda f$ as shorthand for $\inf_{\alpha < \lambda} f(\alpha)$, and analogous shorthands for sup, lim inf, and lim sup, we have $\inf_\lambda f \sqsubseteq \liminf_\lambda f \sqsubseteq \limsup_\lambda f \sqsubseteq \sup_\lambda f$. If f is monotone, then even $\liminf_\lambda f = \sup_\lambda f$, and if f is antitone, then $\inf_\lambda f = \limsup_\lambda f$.

If $f \in \mathfrak{L} \to \mathfrak{L}$ and $g \in \mathfrak{L}$, we define transfinite iteration $f^\alpha(g)$ by recursion on α as follows:

$$
\begin{aligned}
f^0 \ (g) &:= g \\
f^{\alpha+1}(g) &:= f(f^\alpha(g)) \\
f^\lambda \ (g) &:= \limsup_{\alpha \to \lambda} f^\alpha(g)
\end{aligned}
$$

This definition of iteration works for any f, not just monotone ones. For monotone f, we obtain the usual approximants of least and greatest fixed-points as $\boldsymbol{\mu}^\alpha f = f^\alpha(\bot)$ and $\boldsymbol{\nu}^\alpha f = f^\alpha(\top)$: It is easy to check that $\boldsymbol{\mu}^\lambda f = \sup_{\alpha < \lambda} \boldsymbol{\mu}^\alpha f$ and $\boldsymbol{\nu}^\lambda f = \inf_{\alpha < \lambda} \boldsymbol{\nu}^\alpha f$, so our definition coincides with the usual one.

3.4. **Closure ordinal.** We can calculate an upper bound for the ordinal $\top^{\mathsf{ord}}$ at which all fixed-points are reached as follows: Let $\beth_n$ be a sequence of cardinals defined by $\beth_0 = |\mathbb{N}|$ and $\beth_{n+1} = |\mathcal{P}(\beth_n)|$. For a pure kind κ, let $|\kappa|$ be the number of $*$s in κ. Since $[\![*]\!]$ consists of countable sets, $|[\![*]\!]| \leq |\mathcal{P}(\mathbb{N})| = \beth_1$, and by induction on κ, $|[\![\kappa]\!]| \leq \beth_{|\kappa|+1}$. Since an (ascending or descending) chain in $[\![\kappa]\!]$ is shorter than $|[\![\kappa]\!]|$, each fixed point is reached latest at the $|[\![\kappa]\!]|$th iteration. Hence, the closure ordinal for all (co)inductive types can be approximated from above by $\top^{\mathsf{ord}} = \beth_\omega$.

This calculation does not work if we allow fixed-points of constructors involving ord. Then the closure ordinal of such a fixed-point would depend on which ordinals are in the semantics of ord, which in turn would depend on what the closure ordinal for all fixed-points was—a vicious cycle. However, I do not see a practical example where one want to construct the fixed point of a sized-type transformer $F : (\mathsf{ord} \xrightarrow{\circ} \kappa) \xrightarrow{+} (\mathsf{ord} \xrightarrow{\circ} \kappa)$. Note that this does not exclude fixed-points inside fixed-points, such as

$$
\mathsf{BTree}^{i,j} A = \mu^i \lambda X. \ 1 + X \times (\mu^j \lambda Y. \ 1 + A \times X \times Y),
$$

"B-trees" of height $< \imath$ with each node containing $< \jmath$ keys of type A.

Example 3.1 (Number classes)**.** Here we show that higher-kinded inductive types may require strictly higher closure ordinals than inductive types of kind $*$. Following Hancock [Han02], we can define the number classes as inductive types as follows:

$$
\begin{array}{lllll}
\mathsf{NC}_0 & := & \mu^\infty \lambda X.\, 1 & \cong & 1 \\
\mathsf{NC}_1 & := & \mu^\infty \lambda X.\, 1 + (\mathsf{NC}_0 \to X) & \cong & \mathsf{Nat}^\infty \\
\mathsf{NC}_2 & := & \mu^\infty \lambda X.\, 1 + (\mathsf{NC}_0 \to X) + (\mathsf{NC}_1 \to X) & \cong & \mu^\infty \lambda X.\, 1 + X + (\mathsf{Nat}^\infty \to X) \\
\mathsf{NC}_3 & := & \mu^\infty \lambda X.\, 1 + (\mathsf{NC}_0 \to X) + (\mathsf{NC}_1 \to X) + (\mathsf{NC}_2 \to X) & & \\
\;\;\vdots & & & &
\end{array}
$$

The second number class NC_2 is also known as *Brouwer ordinals*. The law behind this scheme is: $\mathsf{NC}_n = \mu F_n$, where $F_0 X = 1$ and $F_{n+1} X = F_n X + (\mu F_n \to X)$. Each number class requires a higher closure ordinal, and their limit is the closure ordinal of all inductive types of kind $*$. Now let

$$
\begin{array}{lll}
\mathsf{NumClTree} & : & \mathsf{ord} \xrightarrow{+} (* \xrightarrow{+} *) \xrightarrow{\circ} * \\
\mathsf{NumClTree} & := & \lambda \imath.\, \mu^\imath \lambda Y \lambda F.\, 1 + (\mu^\infty F \to Y\, (\lambda X.\, F\, X + (\mu^\infty F \to X))).
\end{array}
$$

Then $\mathsf{NumClTree}^\infty\,(\lambda X.\, 1)$ is the type of trees branching over the nth number class at the nth level.

3.5. Interpretation of types.

For r a term, e an evaluation frame, and $\mathcal{A}$ a term set, let $r \cdot \mathcal{A} = \{r\, s \mid s \in \mathcal{A}\}$ and $e^{-1}\mathcal{A} = \{r \mid e(r) \in \mathcal{A}\}$. If e is strongly normalizing and $\mathcal{A}$ saturated, then $e^{-1}\mathcal{A}$ is again saturated. For saturated sets $\mathcal{A}, \mathcal{B} \in [\![*]\!]$ we define the following saturated sets:

$$
\begin{array}{lllclll}
\mathcal{A} \boxplus \mathcal{B} & := & \overline{\mathsf{inl} \cdot \mathcal{A}} \cup \overline{\mathsf{inr} \cdot \mathcal{B}} & \qquad & \boxed{1} & := & \overline{\{\langle\rangle\}} \\[4pt]
\mathcal{A} \boxtimes \mathcal{B} & := & (\mathsf{fst}\, \_)^{-1}\mathcal{A} \cap (\mathsf{snd}\, \_)^{-1}\mathcal{B} & & \mathcal{A}^\mu & := & \overline{\mathsf{in} \cdot \mathcal{A}} \\[4pt]
\mathcal{A} \boxminus\!\!\to \mathcal{B} & := & \bigcap_{s \in \mathcal{A}} (\_\, s)^{-1}\mathcal{B} & & \mathcal{A}^\nu & := & (\mathsf{out}\, \_)^{-1}\mathcal{A}
\end{array}
$$

The last two notations are lifted pointwise to operators $\mathcal{F} \in [\![p\kappa \to \kappa']\!]$ by setting $\mathcal{F}^\nabla(\mathcal{G}) = (\mathcal{F}(\mathcal{G}))^\nabla$, where $\nabla \in \{\mu, \nu\}$.

Remark 3.2. Our definition of product and function space (inspired by Vouillon [Vou04]) makes it immediate that $\boxtimes$ and $\boxminus\!\!\to$ operate on saturated sets. But it is just a reformulation of the usual $\mathcal{A} \boxtimes \mathcal{B} = \{r \mid \mathsf{fst}\, r \in \mathcal{A} \text{ and } \mathsf{snd}\, r \in \mathcal{B}\}$ and $\mathcal{A} \to \mathcal{B} = \{r \mid r\, s \in \mathcal{B} \text{ for all } s \in \mathcal{A}\}$.

Notice that the *finitary* or (in the logical sense) *positive* connectives 1, $+$, and μ are defined via introductions, while the *infinitary* or *negative* connectives $\to$ and ν are defined via eliminations. (The product fits in either category.)

For a constructor constant $C : \kappa$, the semantics $[\![C]\!] \in [\![\kappa]\!]$ is defined as follows:

$$
\begin{aligned}
[\![+]\!](\mathcal{A}, \mathcal{B} \in [\![*]\!]) &:= \mathcal{A} \boxplus \mathcal{B} & [\![1]\!] &:= \boxed{1} \\
[\![\times]\!](\mathcal{A}, \mathcal{B} \in [\![*]\!]) &:= \mathcal{A} \boxtimes \mathcal{B} & [\![\infty]\!] &:= \top^{\mathsf{ord}} \\
[\![\rightarrow]\!](\mathcal{A}, \mathcal{B} \in [\![*]\!]) &:= \mathcal{A} \boxarrow \mathcal{B} & [\![\mathsf{s}]\!](\top^{\mathsf{ord}}) &:= \top^{\mathsf{ord}} \\
[\![\mu_\kappa]\!](\alpha)(\mathcal{F} \in [\![\kappa]\!] \xrightarrow{+} [\![\kappa]\!]) &:= \boldsymbol{\mu}^\alpha \mathcal{F}^\mu & [\![\mathsf{s}]\!](\alpha < \top^{\mathsf{ord}}) &:= \alpha + 1 \\
[\![\nu_\kappa]\!](\alpha)(\mathcal{F} \in [\![\kappa]\!] \xrightarrow{+} [\![\kappa]\!]) &:= \boldsymbol{\nu}^\alpha \mathcal{F}^\nu \\
[\![\forall_\kappa]\!](\mathcal{F} \in [\![\kappa]\!] \rightarrow [\![*]\!]) &:= \bigcap_{\mathcal{G} \in [\![\kappa]\!]} \mathcal{F}(\mathcal{G})
\end{aligned}
$$

This semantics is extended to arbitrary constructors in the usual way. Let $\mathsf{U} = \bigcup_\kappa [\![\kappa]\!]$. For a valuation θ which partially maps constructor variables X to their interpretation $\mathcal{G} \in \mathsf{U}$, we define the partial map $[\![-]\!]_\theta$ from constructors F to their interpretation in U by recursion on F.

$$
\begin{aligned}
[\![C]\!]_\theta &:= [\![C]\!] \\
[\![X]\!]_\theta &:= \theta(X) \\
[\![F\,G]\!]_\theta &:= [\![F]\!]_\theta([\![G]\!]_\theta)
\end{aligned}
$$

$$
[\![\lambda X : \kappa.\, F]\!]_\theta := \begin{cases} \mathcal{F} & \text{if } \mathcal{F} \in [\![\kappa]\!] \rightarrow [\![\kappa']\!] \text{ for some } \kappa' \\ \text{undef.} & \text{else} \end{cases}
$$
$$
\text{where } \mathcal{F}(\mathcal{G} \in [\![\kappa]\!]) := [\![F]\!]_{\theta[X \mapsto \mathcal{G}]}
$$

In the last clause, $\mathcal{F}$ is a partial function from $[\![\kappa]\!]$ to U.

The interpretation $[\![F]\!]_\theta$ is well-defined for well-kinded F, and these are the only constructors we are interested in, but we chose to give a (possibly undefined) meaning to all constructors. If one restricts the interpretation to well-kinded constructors, one has to define it by recursion on kinding derivation and show coherence: If a constructor has two kinding derivations ending in the same kind, then the two interpretations coincide. This alternative requires a bit more work than our choice.

Lemma 3.3 (Basic properties of interpretation)**.**
(1) *Relevance:* If $\theta(X) = \theta'(X)$ for all $X \in \mathsf{FV}(F)$, then $[\![F]\!]_\theta = [\![F]\!]_{\theta'}$.
(2) *Substitution:* $[\![[G/X]F]\!]_\theta = [\![F]\!]_{\theta[X \mapsto [\![G]\!]_\theta]}$.

Proof. Each by induction on F. For (2), consider case $F = \lambda Y : \kappa.\, F'$. W.l.o.g., $Y \notin \mathsf{FV}(G)$. By induction hypothesis,

$$
\mathcal{F}(\mathcal{H}) := [\![[G/X]F]\!]_{\theta[Y \mapsto \mathcal{H}]} = [\![F]\!]_{\theta[Y \mapsto \mathcal{H}][X \mapsto [\![G]\!]_{\theta[Y \mapsto \mathcal{H}]}]} = [\![F]\!]_{\theta[X \mapsto [\![G]\!]_\theta][Y \mapsto \mathcal{H}]},
$$

using (1) on G. Hence, $[\![[G/X](\lambda Y : \kappa.\, F)]\!]_\theta = [\![\lambda Y : \kappa.\, F]\!]_{\theta[X \mapsto [\![G]\!]_\theta]}$. $\qquad \square$

Although the substitution property holds even for ill-kinded constructors, we only have for well-kinded constructors that $[\![(\lambda X : \kappa.\, F)\,G]\!]_\theta = [\![[G/X]F]\!]_\theta$. In general, the left hand side is less defined than the right hand side, e.g., $[\![(\lambda X : *.\, 1)\,\infty]\!]_\theta$ is undefined, whereas the interpretation $[\![1]\!]_\theta$ of its β-reduct is well-defined. In the following we show that for well-kinded constructors the interpretation is well-defined and invariant under β.

Theorem 3.4 (Soundness of kinding, equality, and subtyping for constructors)**.** *Let* $\theta \sqsubseteq \theta' \in [\![\Delta]\!]$, *meaning that for all* $(X : p\kappa') \in \Delta$ *it holds that* $\mathcal{G} := \theta(X) \in [\![\kappa']\!]$ *and* $\mathcal{G}' := \theta'(X) \in [\![\kappa']\!]$, *and* $\mathcal{G} = \mathcal{G}'$ *if* $p = \circ$, $\mathcal{G} \sqsubseteq \mathcal{G}'$ *if* $p = +$, *and* $\mathcal{G}' \sqsubseteq \mathcal{G}$ *if* $p = -$.
(1) *If* $\Delta \vdash F : \kappa$ *then* $[\![F]\!]_\theta \sqsubseteq [\![F]\!]_{\theta'} \in [\![\kappa]\!]$.
(2) *If* $\Delta \vdash F = F' : \kappa$ *then* $[\![F]\!]_\theta = [\![F']\!]_{\theta'} \in [\![\kappa]\!]$.

(3) *If $\Delta \vdash F \leq F' : \kappa$ then $[\![F]\!]_\theta \sqsubseteq [\![F']\!]_{\theta'} \in [\![\kappa]\!]$.*

Proof. Simultaneously by induction on the derivation. $\qquad\square$

Now we can compute the semantics of types, e.g., $[\![\mathsf{Nat}^\imath]\!]_{(\imath \mapsto \alpha)} = \mathcal{N}at^\alpha = \boldsymbol{\mu}^\alpha(\mathcal{X} \mapsto (\boxed{1}\boxplus \mathcal{X})^\mu)$. Similarly, the semantic versions of List, Stream, etc. are denoted by $\mathcal{L}ist$, $\mathcal{S}tream$, etc.

3.6. Semantic admissibility and strong normalization.
For the main theorem to follow, we assume semantic soundness of our yet to be defined syntactical criterion of admissibility (Def. 6.3).

Assumption 3.5 (Semantic admissibility). *If $\Gamma \vdash A$ fix^∇_n-adm and $\theta(X) \in [\![\kappa]\!]$ for all $(X\!:\!\kappa) \in [\![\Gamma]\!]$ then $\mathcal{A} := [\![A]\!]_\theta \in [\![\mathsf{ord}]\!] \to [\![*]\!]$ has the following properties:*

(1) *Shape: $\mathcal{A}(\alpha) = \bigcap_{k \in K} \mathcal{B}_1(k, \alpha) \boxed{\to} \ldots \boxed{\to} \mathcal{B}_n(k, \alpha) \boxed{\to} \mathcal{B}(k, \alpha)$ for some K and some $\mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{B} \in K \times [\![\mathsf{ord}]\!] \to [\![*]\!]$. In case $\nabla = \mu$, $\mathcal{B}(k, \alpha) = \mathcal{I}(k, \alpha)^\mu \boxed{\to} \mathcal{C}(k, \alpha)$ for some $\mathcal{I}, \mathcal{C}$. Otherwise, $\mathcal{B}(k, \alpha) = \mathcal{C}(k, \alpha)^\nu$ for some $\mathcal{C}$.*
(2) *Bottom-check: $\mathcal{I}(k, 0)^\mu = \bot^*$ in case $\nabla = \mu$ and $\mathcal{C}(k, 0)^\nu = \top^*$ in case $\nabla = \nu$.*
(3) *Limit-check: $\inf_{\alpha < \lambda} \mathcal{A}(\alpha) \subseteq \mathcal{A}(\lambda)$ for all limit ordinals $\lambda \in [\![\mathsf{ord}]\!] \setminus \{0\}$.*

In case of recursion ($\nabla = \mu$), the condition (1) ensures that $\mathsf{fix}^\mu_n s$ really produces a function whose $n+1$st argument is of something that looks like an inductive type ($\mathcal{I}(k, \alpha)^\mu$). The function can be polymorphic, therefore the intersection $\bigcap_{k \in K}$ over an index set K. Condition (2) requires $\mathcal{I}$ to exhibit at least for $\alpha = 0$ the behavior of an inductive type: $\mathcal{I}(k, 0)^\mu = \bot^*$, which is equivalent to $\mathcal{I}(k, 0) = \emptyset$. The technical condition (3) is used in the following theorem and will occupy our attention for the remainder of this article. In case of corecursion ($\nabla = \nu$), condition (1) ensures that $\mathsf{fix}^\nu_n s$ maps n arguments into something like a coinductive type ($\mathcal{C}(k, \alpha)^\nu$), which needs to cover the whole universe $\top^*$ of terms for $\alpha = 0$.

Now we show soundness of our typing rules, which entails strong normalization. Let $t\theta$ denote the simultaneous substitution of $\theta(x)$ for each $x \in \mathsf{FV}(t)$ in t.

Theorem 3.6 (Soundness of typing). *Assume that the judgement $\Gamma \vdash A$ fix^∇_n-adm is sound, as stated above. Let $\theta(X) \in [\![\kappa]\!]$ for all $(X\!:\!\kappa) \in \Gamma$ and $\theta(x) \in [\![A]\!]_\theta$ for all $(x\!:\!A) \in \Gamma$. If $\Gamma \vdash t : B$ then $t\theta \in [\![B]\!]_\theta$.*

Proof. By induction on the typing derivation. We consider the recursion rule (TY-REC for $\nabla = \mu$).

$$\text{TY-REC} \quad \frac{\Gamma \vdash A \; \mathsf{fix}^\mu_n\text{-adm} \qquad \Gamma \vdash a : \mathsf{ord}}{\Gamma \vdash \mathsf{fix}^\mu_n : (\forall \imath\!:\!\mathsf{ord}.\; A\,\imath \to A\,(\imath + 1)) \to A\,a}$$

By hypothesis, $\mathcal{A} := [\![A]\!]_\theta \in [\![\mathsf{ord}]\!] \to [\![*]\!]$ is admissible, and $\alpha := [\![a]\!]_\theta \in [\![\mathsf{ord}]\!]$. Assume an $s \in [\![\forall \imath : \mathsf{ord}.\; A\,\imath \to A\,(\imath + 1)]\!]_\theta \subseteq \bigcap_{\beta < \top^{\mathsf{ord}}} \mathcal{A}(\beta) \boxed{\to} \mathcal{A}(\beta + 1)$. We show $\mathsf{fix}^\mu_n s \in \mathcal{A}(\alpha)$ by transfinite induction on α.

In the base case $\alpha = 0$, $\mathcal{A}(0) = \bigcap_{k \in K} \mathcal{B}_{1..n}(k, 0) \boxed{\to} \bot^* \boxed{\to} \mathcal{C}(k, 0)$. We assume $k \in K$, $t_i \in \mathcal{B}_i(k, 0)$, then $e(r) := \mathsf{fix}^\mu_n s\, t_{1..n}\, r$ is a strongly normalizing evaluation frame. Since each $r \in \bot^* = \mathcal{N}$ is neutral, we have $e(r) \in \mathcal{N} \subseteq \mathcal{C}(k, 0)$.

In the step case, $\mathcal{A}(\alpha + 1) = \bigcap_{k \in K} \mathcal{B}_{1..n}(k, \alpha + 1) \boxed{\to} \mathcal{I}(k, \alpha + 1)^\mu \boxed{\to} \mathcal{C}(k, \alpha + 1)$. We assume $k \in K$, $t_i \in \mathcal{B}_i(k, \alpha + 1)$, and $r \in \mathcal{I}(k, \alpha + 1)^\mu$, which means that either r is neutral—then we continue as in the previous case—or $r \rhd \mathsf{in}\, r'$. Now $\mathsf{fix}^\mu_n s\, \vec{t}\, r \rhd \mathsf{fix}^\mu_n s\, \vec{t}\,(\mathsf{in}\, r') \rhd s\,(\mathsf{fix}^\mu_n s)\, \vec{t}\,(\mathsf{in}\, r')$. The last term inhabits $\mathcal{C}(k, \alpha + 1)$, since $\mathsf{fix}^\mu_n s \in \mathcal{A}(\alpha)$ by induction hypothesis and therefore $s\,(\mathsf{fix}^\mu_n s) \in \mathcal{A}(\alpha + 1)$.

Finally, in the limit case, $\mathsf{fix}_n^\mu s \in \mathcal{A}(\alpha)$ for all $\alpha < \lambda$ by induction hypothesis. Since $\bigcap_{\alpha<\lambda} \mathcal{A}(\alpha) = \inf_{\alpha<\lambda} \mathcal{A}(\alpha) \subseteq \mathcal{A}(\lambda)$, we are done. $\square$

Corollary 3.7 (Strong normalization). *If $\Gamma \vdash t : B$ then t is strongly normalizing.*

Proof. From soundness of typing, taking a valuation θ with $\theta(x) = x$ for all term variables x and $\theta(X) = \top^\kappa$ for all $(X\!:\!p\kappa) \in \Gamma$. $\square$

4. Semi-Continuity

As motivated in the introduction, only types $\mathcal{C} \in [\![\mathsf{ord}]\!] \to [\![*]\!]$ which satisfy the limit-check $\inf_\lambda \mathcal{C} \sqsubseteq \mathcal{C}(\lambda)$ can be admissible for recursion. In this section we develop a compositional criterion for admissible types. The limit-check itself is not compositional since it does not sensibly distribute over function spaces: To show $\inf_{\alpha<\lambda}(\mathcal{A}(\alpha) \boxed{\to} \mathcal{B}(\alpha)) \sqsubseteq \mathcal{A}(\lambda) \boxed{\to} \mathcal{B}(\lambda)$ from $\inf_\lambda \mathcal{B} \sqsubseteq \mathcal{B}(\lambda)$ requires $\mathcal{A}(\lambda) \sqsubseteq \inf_\lambda \mathcal{A}$, which is not even true for $\mathcal{A}(\alpha) = \mathcal{N}at^\alpha$ at limit ω. However, the criterion $\limsup_\lambda \mathcal{C} \sqsubseteq \mathcal{C}(\lambda)$ entails the limit-check, and it distributes reasonably over the function space:

Proposition 4.1. *If $\mathcal{A}(\lambda) \sqsubseteq \liminf_\lambda \mathcal{A}$ and $\limsup_\lambda \mathcal{B} \sqsubseteq \mathcal{B}(\lambda)$ then $\limsup_\lambda(\mathcal{A}(\alpha) \boxed{\to} \mathcal{B}(\alpha)) \sqsubseteq \mathcal{A}(\lambda) \boxed{\to} \mathcal{B}(\lambda)$.*

This proposition will reappear (and proven) as Cor. 4.8. Note that $\mathcal{N}at^\omega = \liminf_\omega \mathcal{N}at$, hence, $\mathcal{A}(\alpha) = \mathcal{N}at^\alpha$ can now serve as the domain of an admissible function space, which is the least we expect.

The conditions on $\mathcal{A}$ and $\mathcal{B}$ in the lemma are established mathematical terms: They are subconcepts of continuity. In this article, we consider only functions $f \in \mathsf{O} \to \mathfrak{L}$ from ordinals into some lattice $\mathfrak{L}$. For such f, the question whether f is continuous in point α only makes sense if α is a limit ordinal, because only then there are infinite non-stationary sequences which converge to α; and since every strictly decreasing sequence is finite on ordinals (well-foundedness!), it only makes sense to look at *ascending* sequences, i.e., approaching the limit from the left. Hence, function f is *upper semi-continuous* in λ, if $\limsup_\lambda f \sqsubseteq f(\lambda)$, and *lower semi-continuous*, if $f(\lambda) \sqsubseteq \liminf_\lambda f$. If f is both upper and lower semi-continuous in λ, then it is continuous in λ (then upper and lower limit coincide with $f(\lambda)$).

In the following we identify sufficient criteria for sum, product, function, inductive, and coinductive types to be semi-continuous.

4.1. **Semi-continuity from monotonicity.**

Obviously, any monotone function is upper semi-continuous, and any antitone function is lower semi-continuous. Now consider a monotone f with $f(\lambda) = \sup_\lambda f$, as it is the case for an inductive type $f(\alpha) = \boldsymbol{\mu}^\alpha \mathcal{F}$ (where $\mathcal{F}$ does not depend on α). Since for monotone f, $\sup_\lambda f = \liminf_\lambda f$, f is lower semi-continuous. This criterion can be used with Prop. 4.1 to show upper semi-continuity of function types with inductive domain, such as $\mathsf{Eq}(\mathsf{GRose}^\imath F A)$ (see introduction) and, e.g.,

$$\mathcal{C}(\alpha) = \mathcal{N}at^\alpha \boxed{\to} \mathcal{L}ist^\alpha(\mathcal{A}) \boxed{\to} \mathcal{C}'(\alpha)$$

where $\mathcal{C}'(\alpha)$ is any monotonic type-valued function, for instance, $\mathcal{L}ist^\alpha(\mathcal{N}at^\alpha)$, and $\mathcal{A}$ is some constant type: The domain types, $\mathcal{N}at^\alpha$ and $\mathcal{L}ist^\alpha(\mathcal{A})$, are lower semi-continuous according the just established criterion and the monotonic codomain $\mathcal{C}'(\alpha)$ is upper semi-continuous,

hence, Prop. 4.1 proves upper semi-continuity of $\mathcal{C}$. Note that this criterion fails us if we replace the domain $\mathcal{L}ist^\alpha(\mathcal{A})$ by $\mathcal{L}ist^\alpha(\mathcal{N}at^\alpha)$, or even $\boldsymbol{\mu}^\alpha(\mathcal{F}(\mathcal{N}at^\alpha))$ for some monotone $\mathcal{F}$, since it is not immediately obvious that

$$\boldsymbol{\mu}^\omega(\mathcal{F}(\mathcal{N}at^\omega)) = \sup_{\alpha<\omega} \boldsymbol{\mu}^\alpha(\mathcal{F}(\sup_{\beta<\omega} \mathcal{N}at^\beta)) \stackrel{?}{=} \sup_{\gamma<\omega} \boldsymbol{\mu}^\gamma(\mathcal{F}(\mathcal{N}at^\gamma)).$$

However, domain types where one indexed inductive type is inside another inductive type are useful in practice, see Example 6.6. Before we consider lower semi-continuity of such types, let us consider the dual case.

For $f(\alpha) = \boldsymbol{\nu}^\alpha\mathcal{F}$, $\mathcal{F}$ not dependent on α, f is antitone and $f(\lambda) := \limsup_\lambda f = \inf_\lambda f$, hence, f is continuous in all limits. This establishes upper semi-continuity of a type involved in stream-zipping,

$$\mathcal{S}tream^\alpha(\mathcal{A}) \boxed{\to} \mathcal{S}tream^\alpha(\mathcal{B}) \boxed{\to} \mathcal{S}tream^\alpha(\mathcal{C}).$$

However, types like $\mathcal{S}tream^\alpha(\mathcal{N}at^\alpha)$ are not yet covered, but now we will develop concepts that allow us to look inside (co)inductive types.

4.2. **Simple semi-continuous types.** First we will investigate how disjoint sum, product, and function space operate on semi-continuous types.

Definition 4.2. Let $f \in \mathfrak{L} \to \mathfrak{L}'$. We say $\limsup$ *pushes through* f, or f is $\limsup$-*pushable*, if for all $g \in \mathsf{O} \to \mathfrak{L}$,

$$\limsup_{\alpha\to\lambda} f(g(\alpha)) \sqsubseteq f(\limsup_\lambda g).$$

Analogously, f is $\liminf$-*pullable*, or $\liminf$ *can be pulled out of* f, if for all g,

$$f(\liminf_\lambda g) \sqsubseteq \liminf_{\alpha\to\lambda} f(g(\alpha)).$$

These notions extend straightforwardly to fs with several arguments.

Lemma 4.3 (Facts about limits).
(1) $\sup_{i\in I} \liminf_{\alpha\to\lambda} h(\alpha,i) \sqsubseteq \liminf_{\alpha\to\lambda} \sup_{i\in I} h(\alpha,i)$.
(2) $\limsup_{\alpha\to\lambda} \inf_{i\in I} h(\alpha,i) \sqsubseteq \inf_{i\in I} \limsup_{\alpha\to\lambda} h(\alpha,i)$.
(3) $\limsup_{\alpha\to\lambda} \inf_{i\in I(\alpha)} h(\alpha,i) \sqsubseteq \inf_{i\in\liminf_\lambda I} \limsup_{\alpha\to\lambda} h(\alpha,i)$. $\qquad\square$

Fact (2) states that $\limsup$ pushes through infimum, setting $\mathfrak{L} = K \to \mathfrak{L}'$ for some set $K \supseteq I$, $f(g') = \inf_{i\in I} g'(i)$, and $g(\alpha)(i) = h(\alpha,i)$ in the above definition. Thus, universal quantification is $\limsup$-pushable, which justifies rule CONT-$\forall$ in Figure 3 (see Sect. 6). The dual fact (1) expresses that $\liminf$ can be pulled out of a supremum.

Fact (3) is a generalization of (2) which we will need to show semi-continuity properties of the function space.

Proof. In the following proof of (3), let all ordinals range below λ. First we derive

$$
\begin{array}{rcll}
h(\alpha,i) & \sqsubseteq & \sup_{\alpha\geq\alpha_0} h(\alpha,i) & \text{for } \alpha \geq \alpha_0 \\
\inf_{i\in I(\alpha)} h(\alpha,i) & \sqsubseteq & \inf_{i\in\bigcap_{\alpha\geq\alpha_0} I(\alpha)} \sup_{\alpha\geq\alpha_0} h(\alpha,i) & \text{for } \alpha \geq \alpha_0 \\
\sup_{\alpha\geq\alpha_0} \inf_{i\in I(\alpha)} h(\alpha,i) & \sqsubseteq & \inf_{i\in\inf_{\alpha\geq\alpha_0} I(\alpha)} \sup_{\alpha\geq\alpha_0} h(\alpha,i). &
\end{array}
$$

($I(\alpha)$ is a set, so intersection = infimum.) Secondly, note that

$$\inf_{\alpha_0} \inf_{i\in J(\alpha_0)} g(\alpha_0,i) = \inf_{\alpha_0,i\in J(\alpha_0)} \inf_{\alpha_0} g(\alpha_0,i) = \inf_{i\in\sup_{\alpha_0} J(\alpha_0)} \inf_{\alpha_0} g(\alpha_0,i).$$

With $g(\alpha_0, i) := \sup_{\alpha \geq \alpha_0} h(\alpha, i)$ and $J(\alpha_0) := \inf_{\alpha \geq \alpha_0} I(\alpha)$ we finally derive

$$
\begin{aligned}
\limsup_{\alpha \to \lambda} \inf_{i \in I(\alpha)} h(\alpha, i) &= \\
\inf_{\alpha_0} \sup_{\alpha \geq \alpha_0} \inf_{i \in I(\alpha)} h(\alpha, i) &\sqsubseteq \quad \inf_{\alpha_0} \inf_{i \in \inf_{\alpha \geq \alpha_0} I(\alpha)} \sup_{\alpha \geq \alpha_0} h(\alpha, i) \\
&= \quad \inf_{i \in \sup_{\alpha_0} \inf_{\alpha \geq \alpha_0} I(\alpha)} \inf_{\alpha_0} \sup_{\alpha \geq \alpha_0} h(\alpha, i) \\
&= \quad \inf_{i \in \liminf_\lambda I} \limsup_{\alpha \to \lambda} h(\alpha, i).
\end{aligned}
$$

$\square$

Lemma 4.4 (lim inf can be pulled out of the building blocks of saturated sets)**.**
- (1) $r \cdot \liminf_\lambda \mathcal{A} \qquad\qquad \subseteq \liminf_{\alpha \to \lambda}(r \cdot \mathcal{A}(\alpha))$.
- (2) $e^{-1}(\liminf_\lambda \mathcal{A}) \qquad\quad \subseteq \liminf_{\alpha \to \lambda} e^{-1}(\mathcal{A}(\alpha))$.
- (3) $\liminf_\lambda \mathcal{A} \cap \liminf_\lambda \mathcal{B} \subseteq \liminf_{\alpha \to \lambda}(\mathcal{A}(\alpha) \cap \mathcal{B}(\alpha))$.
- (4) $^{\triangleright}(\liminf_\lambda \mathcal{A}) \qquad\quad \subseteq \liminf_{\alpha \to \lambda} {}^{\triangleright}(\mathcal{A}(\alpha))$.

Proof. All propositions have easy proofs. Let all introduced ordinals range below λ. For proposition (3), assume $r \in \liminf_\lambda \mathcal{A} \cap \liminf_\lambda \mathcal{B}$, which means that there are α_0, β_0 such that $r \in \mathcal{A}(\alpha)$ for all α with $\alpha_0 \leq \alpha$ and $r \in B(\beta)$ for all β with $\beta_0 \leq \beta$. We have to show that there exists γ_0 such that $r \in \mathcal{A}(\gamma) \cap \mathcal{B}(\gamma)$ for all γ with $\gamma_0 \leq \gamma$. Choose $\gamma_0 := \max(\alpha_0, \beta_0)$. Notice that even $\liminf_\lambda \mathcal{A} \cap \liminf_\lambda \mathcal{B} = \liminf_{\alpha \to \lambda}(\mathcal{A}(\alpha) \cap \mathcal{B}(\alpha))$, since the reverse direction is follows immediately from $\mathcal{A}(\alpha) \cap \mathcal{B}(\alpha) \subseteq \mathcal{A}(\alpha), \mathcal{B}(\alpha)$.

For proposition (4), assume $r \triangleright r' \in \liminf_\lambda \mathcal{A}$. There exists α_0 such that for all $\alpha \geq \alpha_0$, we have $r' \in A(\alpha)$, and thus, $r \in {}^{\triangleright}(\mathcal{A}(\alpha))$. It follows that $r \in \liminf_{\alpha \to \lambda} {}^{\triangleright}(\mathcal{A}(\alpha))$. $\square$

The last lemma can be dualized to lim sup:

Lemma 4.5 (lim sup pushes through the building blocks of saturated sets)**.**
- (1) $\limsup_{\alpha \to \lambda}(r \cdot \mathcal{A}(\alpha)) \qquad \subseteq r \cdot \limsup_\lambda \mathcal{A}$.
- (2) $\limsup_{\alpha \to \lambda} e^{-1}(\mathcal{A}(\alpha)) \qquad \subseteq e^{-1}(\limsup_\lambda \mathcal{A})$.
- (3) $\limsup_{\alpha \to \lambda}(\mathcal{A}(\alpha) \cup \mathcal{B}(\alpha)) \subseteq \limsup_\lambda \mathcal{A} \cup \limsup_\lambda \mathcal{B}$. *(only classically!)*
- (4) $\limsup_{\alpha \to \lambda} {}^{\triangleright}(\mathcal{A}(\alpha)) \qquad \subseteq {}^{\triangleright}(\limsup_\lambda \mathcal{A})$ *if $\mathcal{A}(\alpha) \subseteq \mathcal{V}$ for all $\alpha < \lambda$.*

Proof. The first three propositions follow trivially since the infimum and supremum considered are set-theoretic intersection and union. Note that proposition (3) is valid in classical logics but not in intuitionistic logics: An r which inhabits infinitely many unions $\mathcal{A}(\alpha) \cup \mathcal{B}(\alpha)$, must classically inhabit infinitely many $\mathcal{A}(\alpha)$ or infinitely many $\mathcal{B}(\alpha)$. But we cannot tell which of these alternatives holds, so the proposition has no intuitionistic proof. However, we will not need this proposition for the results to follow.

For the last proposition, assume $r \in \limsup_{\alpha \to \lambda} {}^{\triangleright}(\mathcal{A}(\alpha))$. If we require all following ordinals $< \lambda$, this means that for arbitrary α_0 there exist $\alpha \geq \alpha_0$ and $r' \in \mathcal{A}(\alpha)$ such that $r \triangleright r'$. Since each $r' \in \mathcal{V}$ and safe reduction into $\mathcal{V}$ is deterministic, there is in fact a unique $r' \in \bigcup_{\alpha \geq \alpha_0} \mathcal{A}(\alpha)$ with $r \triangleright r'$ for all α, hence, $r \in {}^{\triangleright}(\limsup_\lambda \mathcal{A})$. $\square$

Proposition (4) of Lemma 4.5 fails if we drop the condition $\mathcal{A}(\alpha) \subseteq \mathcal{V}$: Let $t_i = \mathsf{out}^{i+1}(\mathsf{fix}_0^\nu\, \mathsf{out})$ and observe that $t_i \triangleright t_{i+1}$. Setting $\mathcal{A}(n) := \{t_i \mid i \geq n\}$ we have $t_0 \in \inf_{n < \omega} {}^{\triangleright}(\mathcal{A}(n)) \subseteq \limsup_{n \to \omega} {}^{\triangleright}(\mathcal{A}(n))$, but $\limsup_\omega \mathcal{A} = \inf_\omega \mathcal{A} = \emptyset$, thus, $t_0 \notin {}^{\triangleright}(\limsup_\omega \mathcal{A})$. It did not help that the $\mathcal{A}(n)$ were closed unter $\triangleright$-reduction.

Lemma 4.6. *Binary sums $\boxplus$ and products $\boxtimes$ and the operations $(-)^\mu$ and $(-)^\nu$ are* lim sup*-pushable and* lim inf*-pullable.*

Proof. Directly by the last lemmata. For instance,

$$
\begin{aligned}
(\liminf_\lambda \mathcal{A}) \boxtimes (\liminf_\lambda \mathcal{B}) \;&=\; (\mathsf{fst}^{-1}\liminf_\lambda \mathcal{A}) \cap (\mathsf{snd}^{-1}\liminf_\lambda \mathcal{B}) \\
&\subseteq\; (\liminf_{\alpha\to\lambda}\mathsf{fst}^{-1}\mathcal{A}(\alpha)) \cap (\liminf_{\beta\to\lambda}\mathsf{snd}^{-1}\mathcal{B}(\beta)) \\
&\subseteq\; \liminf_{\gamma\to\lambda}(\mathsf{fst}^{-1}\mathcal{A}(\gamma) \cap \mathsf{snd}^{-1}\mathcal{B}(\gamma)) \\
&=\; \liminf_{\gamma\to\lambda}(\mathcal{A}(\gamma) \boxtimes \mathcal{B}(\gamma)).
\end{aligned}
$$

Because we wish to avoid classical reasoning (Lemma 4.5 (3)) as much as possible, pushing $\limsup$ through disjoint sums requires a closer look: Assume $r \in \limsup_{\gamma\to\lambda}(\mathcal{A}(\gamma) \boxplus \mathcal{B}(\gamma))$, hence, for some γ, either $r \rhd \mathsf{inl}\, r'$ for some $r' \in \mathcal{A}(\gamma)$, or $r \rhd \mathsf{inr}\, r'$ for some $r' \in \mathcal{B}(\gamma)$, or $r \in \mathcal{N}$. Since safe reduction $\rhd$ is deterministic, and $\mathcal{N}$ does not contain values, one of these three alternatives must hold whenever $r \in \mathcal{A}(\gamma) \boxplus \mathcal{B}(\gamma)$ for some γ. So either $r \in {}^{\rhd}(\mathsf{inl} \cdot (\limsup_\lambda \mathcal{A}))$, or $r \in {}^{\rhd}(\mathsf{inr} \cdot (\limsup_\lambda \mathcal{B}))$, or $r \in \mathcal{N}$, which means $r \in (\limsup_\lambda \mathcal{A}) \boxplus (\limsup_\lambda \mathcal{B})$.

Analogously, we show that $\limsup$ pushes through $(\cdot)^\mu$. $\qquad\square$

Using monotonicity of the product constructor, the lemma entails that $\mathcal{A}(\alpha) \boxtimes \mathcal{B}(\alpha)$ is upper/lower semi-continuous if $\mathcal{A}(\alpha)$ and $\mathcal{B}(\alpha)$ are. This applies also for $\boxplus$.

Theorem 4.7 (Pushing $\limsup$ through function space).
$\limsup_{\alpha\to\lambda} (\mathcal{A}(\alpha) \boxminus\!\!\!\to \mathcal{B}(\alpha)) \subseteq (\liminf_\lambda \mathcal{A}) \boxminus\!\!\!\to \limsup_\lambda \mathcal{B}.$

Proof. We use Lemma 4.3 (3).

$$
\begin{aligned}
\limsup_{\alpha\to\lambda} (\mathcal{A}(\alpha) \boxminus\!\!\!\to \mathcal{B}(\alpha)) \;&=\; \limsup_{\alpha\to\lambda} \bigcap_{s\in\mathcal{A}(\alpha)} (\_s)^{-1}\mathcal{B}(\alpha) \\
&\subseteq\; \bigcap_{s\in\liminf_\lambda \mathcal{A}} \limsup_{\alpha\to\lambda}(\_s)^{-1}\mathcal{B}(\alpha) \\
&\subseteq\; \bigcap_{s\in\liminf_\lambda \mathcal{A}} (\_s)^{-1}(\limsup_\lambda \mathcal{B}) \\
&=\; (\liminf_\lambda \mathcal{A}) \boxminus\!\!\!\to \limsup_\lambda \mathcal{B}.
\end{aligned}
$$

$\square$

Corollary 4.8. *If* $\mathcal{A}(\lambda) \sqsubseteq \liminf_\lambda \mathcal{A}$ *and* $\limsup_\lambda \mathcal{B} \sqsubseteq \mathcal{B}(\lambda)$ *then* $\limsup_\lambda(\mathcal{A}(\alpha) \boxminus\!\!\!\to \mathcal{B}(\alpha)) \sqsubseteq \mathcal{A}(\lambda) \boxminus\!\!\!\to \mathcal{B}(\lambda).$

4.3. **Coinductive types preserve upper semi-continuity.** We have already seen that $\mathit{Stream}^\alpha(\mathcal{N}at^\omega)$ is upper semi-continuous. In this section, we establish means to show that also a type like $\mathit{Stream}^\alpha(\mathcal{N}at^\alpha)$ is upper semi-continuous (which is, by the way, isomorphic to $\mathcal{N}at^\alpha \boxminus\!\!\!\to \mathcal{N}at^\alpha$).

Definition 4.9. A family $\mathcal{F}_\alpha \in \mathfrak{L} \to \mathfrak{L}'$ $(\alpha \in \mathsf{O})$ is called $\limsup$-pushable if for any $\mathcal{G}_{(-)} \in \mathsf{O} \to \mathfrak{L}$,

$$
\begin{aligned}
\limsup_{\alpha\to\lambda} \mathcal{F}_\gamma(\mathcal{G}_\alpha) \;&\sqsubseteq\; \mathcal{F}_\gamma(\limsup_{\alpha\to\lambda} \mathcal{G}_\alpha) && \text{for all } \gamma \in \mathsf{O}, \\
\limsup_{\alpha\to\lambda} \mathcal{F}_\alpha(\mathcal{G}_\alpha) \;&\sqsubseteq\; \mathcal{F}_\lambda(\limsup_{\alpha\to\lambda} \mathcal{G}_\alpha) && \text{for all limits } \lambda > 0.
\end{aligned}
$$

The first property is easier to prove, but not entailed by the second property. Usually we will confine in showing the second.

Lemma 4.10. *Let* $\mathcal{F}_\alpha \in \vec{\mathfrak{L}} \to \mathfrak{L} \xrightarrow{+} \mathfrak{L}$ *be a family which is* $\limsup$-*pushable in all arguments.. Then for all* β *the family*

$$
\begin{aligned}
&\mathcal{H}_\alpha \in \vec{\mathfrak{L}} \to \mathfrak{L} \\
&\mathcal{H}_\alpha(\vec{\mathcal{G}}) = \boldsymbol{\nu}^\beta(\mathcal{F}_\alpha(\vec{\mathcal{G}}))
\end{aligned}
$$

is lim sup-*pushable*.

Proof. By transfinite induction on β we prove for all $\mathcal{G}_i \in \mathsf{O} \to \mathfrak{L}_i$ that

$$\limsup_{\alpha \to \lambda} \boldsymbol{\nu}^\beta(\mathcal{F}_\alpha(\vec{\mathcal{G}}_\alpha)) \sqsubseteq \boldsymbol{\nu}^\beta \mathcal{F}_\lambda(\limsup_\lambda \vec{\mathcal{G}}).$$

In case $\beta = 0$, both sides become the maximum element of $\mathfrak{L}$. In the successor case we have

$$
\begin{aligned}
\limsup_{\alpha \to \lambda} \boldsymbol{\nu}^{\beta+1}(\mathcal{F}_\alpha(\vec{\mathcal{G}}_\alpha)) &= \limsup_{\alpha \to \lambda} \mathcal{F}_\alpha(\vec{\mathcal{G}}_\alpha)(\boldsymbol{\nu}^\beta(\mathcal{F}_\alpha(\vec{\mathcal{G}}_\alpha))) \\
&\sqsubseteq \mathcal{F}_\lambda(\limsup_\lambda \vec{\mathcal{G}})(\limsup_{\alpha \to \lambda} \boldsymbol{\nu}^\beta(\mathcal{F}_\alpha(\vec{\mathcal{G}}_\alpha))) \quad \mathcal{F}_\alpha \text{ pushable} \\
&\sqsubseteq \mathcal{F}_\lambda(\limsup_\lambda \vec{\mathcal{G}})(\boldsymbol{\nu}^\beta(\mathcal{F}_\lambda(\limsup_\lambda \vec{\mathcal{G}}))) \quad \mathcal{F}_\alpha \text{ monotone, i.h.} \\
&= \boldsymbol{\nu}^{\beta+1}(\mathcal{F}_\lambda(\limsup_\lambda \vec{\mathcal{G}})).
\end{aligned}
$$

In the remaining case $\beta = \lambda$ we exploit that lim sup pushes through infima (Lemma 4.3.2). $\qquad\square$

In the remainder of this part we will show that $\limsup_{\alpha < \lambda} \boldsymbol{\nu}^{\phi(\alpha)} \mathcal{F}_\alpha \sqsubseteq \boldsymbol{\nu}^{\liminf_\lambda \phi} \mathcal{F}_\lambda$. This will enable us to show that types like $\mathcal{S}tream^\alpha(\mathcal{N}at^\alpha)$ are lim sup-pushable.

In the following, we will need additional properties of lim sup. For the value of $\liminf_\lambda f$ and $\limsup_\lambda f$, only the behavior of f on a final segment of $[0; \lambda[$ is relevant:

Lemma 4.11 (Limit starting later). *Let all ordinal range below λ.*

$$
\begin{aligned}
(1) \quad & \inf_{\beta_0 \geq \alpha_0} \sup_{\beta \geq \beta_0} f(\beta) &=& \inf_{\gamma_0 \geq 0} \sup_{\gamma \geq \gamma_0} f(\gamma). \\
(2) \quad & \sup_{\beta_0 \geq \alpha_0} \inf_{\beta \geq \beta_0} f(\beta) &=& \sup_{\gamma_0 \geq 0} \inf_{\gamma \geq \gamma_0} f(\gamma).
\end{aligned}
$$

Proof. We show (1), the proof of (2) is analogous. Direction $\sqsupseteq$ follows by monotonicity of inf. For $\sqsubseteq$, we have to show that for all γ_0 there exists a $\beta_0 \geq \alpha_0$ such that

$$\sup_{\beta \geq \beta_0} f(\beta) \sqsubseteq \sup_{\gamma \geq \gamma_0} f(\gamma).$$

Take $\beta_0 = \max(\gamma_0, \alpha_0)$. $\qquad\square$

Lemma 4.12 (Splitting limits).

$$
\begin{aligned}
(1) \quad & \limsup_{\beta \to \lambda} h(\beta, \beta) \sqsubseteq \limsup_{\alpha \to \lambda} \limsup_{\beta \to \lambda} h(\alpha, \beta) \quad \text{for } h \in \mathsf{O} \xrightarrow{-} \mathsf{O} \to \mathfrak{L}, \\
(2) \quad & \liminf_{\alpha \to \lambda} \liminf_{\beta \to \lambda} h(\alpha, \beta) \sqsubseteq \liminf_{\beta \to \lambda} h(\beta, \beta) \quad \text{for } h \in \mathsf{O} \xrightarrow{+} \mathsf{O} \to \mathfrak{L}.
\end{aligned}
$$

Proof. Again, we show just (1). Because of antitonicity, we have for all α,

$$
\begin{array}{llll}
h(\beta, \beta) & \sqsubseteq & h(\alpha, \beta) & \text{for all } \beta \geq \alpha \\
\sup_{\beta \geq \beta_0} h(\beta, \beta) & \sqsubseteq & \sup_{\beta \geq \beta_0} h(\alpha, \beta) & \text{for all } \beta_0 \geq \alpha \\
\inf_{\beta_0 \geq \alpha} \sup_{\beta \geq \beta_0} h(\beta, \beta) & \sqsubseteq & \inf_{\beta_0 \geq \alpha} \sup_{\beta \geq \beta_0} h(\alpha, \beta) & \\
\limsup_{\beta \to \lambda} h(\beta, \beta) & \sqsubseteq & \limsup_{\beta \to \lambda} h(\alpha, \beta) & \text{by Lemma 4.11.}
\end{array}
$$

The goal follows by taking lim sup on the r.h.s. $\qquad\square$

Next, we show how to push a $\limsup$ into $\boldsymbol{\nu}^{(-)}$.

Lemma 4.13. *Let $\phi \in \mathsf{O} \to \mathsf{O}$ and $I \subseteq \mathsf{O}$. Then*

(1) $\sup_{\alpha \in I} \boldsymbol{\nu}^{\phi(\alpha)} \sqsubseteq \boldsymbol{\nu}^{\inf_I \phi}$,

(2) $\sup_{\alpha \in I} \boldsymbol{\nu}^{\phi(\alpha)} \sqsupseteq \boldsymbol{\nu}^{\inf_I \phi}$,

(3) $\inf_{\alpha \in I} \boldsymbol{\nu}^{\phi(\alpha)} \sqsupseteq \boldsymbol{\nu}^{\sup_I \phi}$, *and*

(4) $\inf_{\alpha \in I} \boldsymbol{\nu}^{\phi(\alpha)} \sqsubseteq \boldsymbol{\nu}^{\sup_I \phi}$.

Proof. (1) and (3) follow directly from antitonicity. For (2), remember that each set of ordinals is left-closed, hence $\inf_I \phi = \phi(\alpha)$ for some $\alpha \in I$. The remaining proposition (4) is proven by cases on $\sup_I \phi$. If $\sup_I \phi$ is not a limit ordinal then $\sup_I \phi = \phi(\alpha)$ for some $\alpha \in I$. For this α, clearly $\boldsymbol{\nu}^{\phi(\alpha)} \sqsubseteq \boldsymbol{\nu}^{\sup_I \phi}$, which entails the lemma. Otherwise, if $\sup_I \phi$ is a limit ordinal, then by definition of $\boldsymbol{\nu}$ at limits we have to show $\inf_{\alpha \in I} \boldsymbol{\nu}^{\phi(\alpha)} \sqsubseteq \boldsymbol{\nu}^{\beta}$ for all $\beta < \sup_I \phi$. By definition of the supremum, $\beta < \phi(\alpha)$ for some α. Since $\boldsymbol{\nu}$ is antitone, $\boldsymbol{\nu}^{\phi(\alpha)} \sqsubseteq \boldsymbol{\nu}^{\beta}$ from which we infer our subgoal by forming the infimum on the left hand side. $\square$

Corollary 4.14. $\limsup_{\alpha \to \lambda} \boldsymbol{\nu}^{\phi(\alpha)} = \boldsymbol{\nu}^{\liminf_\lambda \phi}$.

Now we have the tools in hand to prove that coinductive types preserve upper semi-continuity. In the second part of the following theorem, we make use of the fact that our coinductive types close at ordinal ω.

Theorem 4.15 (Upper semi-continuity of coinductive types)**.** *Let $\mathcal{F}_\alpha \in \vec{\mathfrak{L}} \to \mathfrak{L} \xrightarrow{+} \mathfrak{L}$ be a family which is $\limsup$-pushable in all arguments and $\phi \in \mathsf{O} \xrightarrow{+} \mathsf{O}$. Then*

$$\limsup_{\alpha \to \lambda} \boldsymbol{\nu}^{\phi(\alpha)}(\mathcal{F}_\alpha(\vec{\mathcal{G}}_\alpha)) \sqsubseteq \boldsymbol{\nu}^{(\liminf_\lambda \phi)}(\mathcal{F}_\lambda(\limsup_\lambda \vec{\mathcal{G}})).$$

If ϕ is constant or affine, then even

$$\limsup_{\alpha \to \lambda} \boldsymbol{\nu}^{\phi(\alpha)}(\mathcal{F}_\alpha(\vec{\mathcal{G}}_\alpha)) \sqsubseteq \boldsymbol{\nu}^{\phi(\lambda)}(\mathcal{F}_\lambda(\limsup_\lambda \vec{\mathcal{G}})).$$

Proof. Direct. Note that $\boldsymbol{\nu}^{\phi(-)}$ is antitone, so we can split the $\limsup$.

$$
\begin{array}{lll}
\limsup_{\alpha \to \lambda} \boldsymbol{\nu}^{\phi(\alpha)}(\mathcal{F}_\alpha(\vec{\mathcal{G}}_\alpha)) & \sqsubseteq & \limsup_{\alpha \to \lambda} \limsup_{\gamma \to \lambda} \boldsymbol{\nu}^{\phi(\alpha)}(\mathcal{F}_\gamma(\vec{\mathcal{G}}_\gamma)) \quad \text{Lemma 4.12} \\
& \sqsubseteq & \limsup_{\alpha \to \lambda} \boldsymbol{\nu}^{\phi(\alpha)}(\mathcal{F}_\lambda(\limsup_\lambda \vec{\mathcal{G}})) \quad \text{Lemma 4.10} \\
& \sqsubseteq & \boldsymbol{\nu}^{\liminf_{\alpha \to \lambda} \phi(\alpha)}(\mathcal{F}_\lambda(\limsup_\lambda \vec{\mathcal{G}})) \quad \text{Cor. 4.14.}
\end{array}
$$

Now if ϕ is constant, then $\liminf_\lambda \phi = \phi(\lambda)$. If ϕ is affine, then surely $\phi(\lambda) \geq \omega$, and also $\liminf_\lambda \phi \geq \omega$. We only need to show that in our case the greatest fixed-point is reached already at iteration ω. Observe that $\mathcal{H}(\mathcal{X}) := \mathcal{F}_\lambda(\limsup_\lambda \vec{\mathcal{G}})(\mathcal{X})$ is $\limsup$-pushable, since the family $\mathcal{F}$ is. It suffices to show that $\boldsymbol{\nu}^\omega \mathcal{H} \sqsubseteq \boldsymbol{\nu}^{\omega+1} \mathcal{H}$.

$$
\begin{array}{lllll}
\boldsymbol{\nu}^\omega \mathcal{H} & = & \limsup_{\beta \to \omega} \boldsymbol{\nu}^\beta \mathcal{H} & = & \limsup_{\beta \to \omega} \boldsymbol{\nu}^{\beta+1} \mathcal{H} \quad = \quad \limsup_{\beta \to \omega} \mathcal{H}(\boldsymbol{\nu}^\beta \mathcal{H}) \\
& \sqsubseteq & \mathcal{H}(\limsup_{\beta \to \omega} \boldsymbol{\nu}^\beta \mathcal{H}) & = & \boldsymbol{\nu}^{\omega+1} \mathcal{H}.
\end{array}
$$

Thus, $\boldsymbol{\nu}^{\omega+1} \mathcal{H} = \boldsymbol{\nu}^\omega \mathcal{H}$, which means that $\boldsymbol{\nu}^\beta \mathcal{H} = \boldsymbol{\nu}^\omega \mathcal{H}$ for all $\beta \geq \omega$; the fixed-point is reached. $\square$

For example, since $\mathcal{F}_\alpha(\mathcal{X}) = (\mathcal{N}at^\alpha \boxtimes \mathcal{X})^\nu$ is lim sup-pushable, we have can infer upper semi-continuity of $\mathcal{S}tream^\alpha(\mathcal{N}at^\alpha) = \boldsymbol{\nu}^\alpha \mathcal{F}_\alpha$.

4.4. **Inductive types preserve lower semi-continuity.** We can dualize the results of the last section and prove that inductive types preserve lower semi-continuity and lim sup-pushability.

Definition 4.16. A family $\mathcal{F}_\alpha \in \mathfrak{L} \to \mathfrak{L}'$ ($\alpha \in \mathsf{O}$) is called lim inf-pullable if for all $\mathcal{G}_{(-)} \in \mathsf{O} \to \mathfrak{L}$,

$$\begin{aligned}
\mathcal{F}_\gamma(\liminf\nolimits_{\alpha \to \lambda} \mathcal{G}_\alpha) &\sqsubseteq \liminf\nolimits_{\alpha \to \lambda} \mathcal{F}_\gamma(\mathcal{G}_\alpha) && \text{for all } \gamma \in \mathsf{O}, \\
\mathcal{F}_\lambda(\liminf\nolimits_{\alpha \to \lambda} \mathcal{G}_\alpha) &\sqsubseteq \liminf\nolimits_{\alpha \to \lambda} \mathcal{F}_\alpha(\mathcal{G}_\alpha) && \text{for all limits } \lambda > 0.
\end{aligned}$$

Lemma 4.17. *Let $\mathcal{F}_\alpha \in \vec{\mathfrak{L}} \to \mathfrak{L} \xrightarrow{+} \mathfrak{L}$ be a family which is* lim inf*-pullable in all arguments. Then,*

(1) $\boldsymbol{\mu}^\beta(\mathcal{F}_{(-)}(-))$ *is a* lim inf*-pullable family,*

(2) $\boldsymbol{\mu}^{(\liminf_\lambda \phi)} = \liminf\nolimits_{\alpha \to \lambda} \boldsymbol{\mu}^{\phi(\alpha)}$.

Theorem 4.18 (Lower semi-continuity of inductive types). *Let $\mathcal{F}_\alpha \in \vec{\mathfrak{L}} \to \mathfrak{L} \xrightarrow{+} \mathfrak{L}$ be a family which is* lim inf*-pullable in all arguments and $\phi \in \mathsf{O} \xrightarrow{+} \mathsf{O}$. Then*

$$\boldsymbol{\mu}^{(\liminf_\lambda \phi)}(\mathcal{F}_\lambda(\liminf_\lambda \vec{\mathcal{G}})) \sqsubseteq \liminf_{\alpha \to \lambda} \boldsymbol{\mu}^{\phi(\alpha)}(\mathcal{F}_\alpha(\vec{\mathcal{G}}_\alpha)).$$

If ϕ is lower semi-continuous, then even

$$\boldsymbol{\mu}^{\phi(\lambda)}(\mathcal{F}_\lambda(\liminf_\lambda \vec{\mathcal{G}})) \sqsubseteq \liminf_{\alpha \to \lambda} \boldsymbol{\mu}^{\phi(\alpha)}(\mathcal{F}_\alpha(\vec{\mathcal{G}}_\alpha)).$$

Putting together the conditions ϕ is required to be monotone and continuous in the second statement of the theorem. Since ϕ is coming from a size expression in our case, such a ϕ will either be the identity or a constant function. (The successor function is *not* continuous!)

Using Thm. 4.18, we can establish lower semi-continuity of $\mathcal{L}ist^\alpha(\mathcal{N}at^\alpha)$.

5. Non Semi-Continuous Types

This section is devoted to show that our list of criteria for semi-continuity is somewhat complete. Concretely, we demonstrate that there is no compositional scheme to establish lower semi-continuity of function types or upper semi-continuity of inductive types.

5.1. **Function space and lower semi-continuity.** One may wonder whether Cor. 4.8 can be dualized, i. e., does upper semi-continuity of $\mathcal{A}$ and lower semi-continuity of $\mathcal{B}$ entail lower semi-continuity of $\mathcal{C}(\alpha) = \mathcal{A}(\alpha) \boxright \mathcal{B}(\alpha)$? The answer is no, e. g., consider $\mathcal{C}(\alpha) = \mathcal{N}at^\omega \boxright \mathcal{N}at^\alpha$. Although $\mathcal{A}(\alpha) = \mathcal{N}at^\omega$ is trivially upper semi-continuous, and $\mathcal{B}(\alpha) = \mathcal{N}at^\alpha$ is lower semi-continuous, $\mathcal{C}$ is not lower semi-continuous: For instance, the identity function is in $\mathcal{C}(\omega)$ but in no $\mathcal{C}(\alpha)$ for $\alpha < \omega$, hence, also not in $\liminf_\omega \mathcal{C}$. And indeed, if this $\mathcal{C}$ was lower semi-continuous, then our criterion would be unsound, because then by Cor. 4.8 the type $(\mathcal{N}at^\omega \boxright \mathcal{N}at^\alpha) \boxright \mathcal{N}at^\omega$, which admits a looping function (see introduction), would be upper semi-continuous.

Nevertheless, there are some lower semi-continuous function spaces, for instance, if the domain is a finite type. For example, $\mathcal{B}ool \boxright \mathcal{N}at^\alpha$ is lower semi-continuous in α, which

implies that $(\mathcal{B}ool \boxed{\rightarrow} \mathcal{N}at^\alpha) \boxed{\rightarrow} \mathcal{N}at^\alpha$ could be admissible. This is the type of a maximum function taking its two inputs in form of a function over booleans ($\mathcal{B}ool \boxed{\rightarrow} \mathcal{N}at^\alpha \cong \mathcal{N}at^\alpha \boxed{\times} \mathcal{N}at^\alpha$). However, this example is somewhat contrived; it is not clear whether such cases appear in practice, so we will not pursue this further here.

5.2. Inductive types and upper semi-continuity.

Pareto [Par00] proves that inductive types are (in our terminology) $\lim\sup$-pushable. His inductive types denote only finitely branching trees, but we also consider infinite branching, arising from function space embedded in inductive types. Such an infinitely branching type is the type of hungry functions (which consumes one argument after the other):

$$\begin{aligned}
\text{Hungry} &: \quad \text{ord} \xrightarrow{+} * \xrightarrow{-} * \\
\text{Hungry} &:= \quad \lambda\imath\lambda A.\ \mu^\imath \lambda X.\ A \to X.
\end{aligned}$$

We are interested in the special case of $\text{Hungry}^\imath(\text{Nat}^\imath)$. In the following we show that accepting such a type as the result of recursion will lead to accepting a non-terminating program. As a consequence, infinitely branching inductive data types, such as $\mu^\imath \lambda X.\ \text{Nat}^\imath \to X$, do not inherit upper semi-continuity from their defining body, here, $\text{Nat}^\imath \to X$ (recall that $\text{Nat}^\imath$ is lower semi-continuous, hence $\text{Nat}^\imath \to X$ is upper semi-continuous). But remember that inductive types can still be upper semi-continuous, e. g., $\text{Hungry}^\imath(\text{Nat}^\infty) = \mu^\imath \lambda X.\ \text{Nat}^\infty \to X$, which is covariant in its size index.

Semantically, we set $\mathcal{H}^\alpha = \boldsymbol{\mu}^\alpha \mathcal{F}_\alpha$, where $\mathcal{F}_\alpha(\mathcal{X}) = (\mathcal{N}at^\alpha \boxed{\rightarrow} \mathcal{X})^\mu$. Since

$$\begin{aligned}
\lim\sup_{\alpha\to\lambda} \mathcal{F}_\alpha(\mathcal{G}(\alpha)) &\sqsubseteq \quad ((\lim\inf_{\alpha\to\lambda} \mathcal{N}at^\alpha) \boxed{\rightarrow} (\lim\sup_\lambda \mathcal{G}))^\mu \\
&= \quad \mathcal{F}_\lambda(\lim\sup_\lambda \mathcal{G}),
\end{aligned}$$

the family $\mathcal{F}_\alpha$ is $\lim\sup$ pushable. If we had a result like

$$\lim\sup_{\alpha\to\lambda} \boldsymbol{\mu}^{\phi(\alpha)} \mathcal{F}_\alpha \sqsubseteq \boldsymbol{\mu}^{\lim\sup_\lambda \phi} \lim\sup_\lambda \mathcal{F},$$

then $\mathcal{H}$ would be upper semi-continuous, and it would be legal to write the following recursive function:

$$\begin{aligned}
\text{h} &: \quad \forall\imath.\ \text{Nat}^\imath \to \text{Hungry}^\imath(\text{Nat}^\imath) \\
\text{h} &:= \quad \text{fix}_0^\mu \lambda h \lambda_-.\ \text{in}\ (\text{s} \circ h \circ \text{pred})
\end{aligned}$$

$$\text{h}\ (\text{in}\ \_) \longrightarrow^+ \text{in}\ (\text{s} \circ \text{h} \circ \text{pred})$$

We will show that the existence of h destroys normalization. In the body of h we refer to an auxiliary function s. As well as its inverse, p, it can be defined by induction on $\imath$:

$$\begin{aligned}
\text{s} &: \quad \forall\imath\forall\jmath.\ \text{Hungry}^\imath(\text{Nat}^\jmath) \to \text{Hungry}^\imath(\text{Nat}^{\jmath+1}) \\
\text{s} &:= \quad \text{fix}_0^\mu \lambda s \lambda h.\ \text{in}\ (s \circ (\text{out}\ h) \circ \text{pred})
\end{aligned}$$

$$\text{s}\ (\text{in}\ f) \longrightarrow^+ \text{in}\ (\text{s} \circ f \circ \text{pred})$$

$$\begin{aligned}
\text{p} &: \quad \forall\imath\forall\jmath.\ \text{Hungry}^\imath(\text{Nat}^{\jmath+1}) \to \text{Hungry}^\imath(\text{Nat}^\jmath) \\
\text{p} &:= \quad \text{fix}_0^\mu \lambda p \lambda h.\ \text{in}\ (p \circ (\text{out}\ h) \circ \text{succ})
\end{aligned}$$

$$\text{p}\ (\text{in}\ f) \longrightarrow^+ \text{in}\ (\text{p} \circ f \circ \text{succ})$$

(Note that these definitions are perfectly acceptable and not to blame.) Another innocent function is the following. Its type looks funny, since it produces something in the empty

type, but let us mind that Hungry, being an inductive type "with nothing to start induction," is also empty.

$$\mathsf{tr} \quad : \quad \forall \imath.\, \mathsf{Hungry}^\imath(\mathsf{Nat}^\infty) \to \forall A.A$$
$$\mathsf{tr} \quad := \quad \mathsf{fix}_0^\mu \lambda tr \lambda h.\; tr\, ((\mathsf{p} \circ (\mathsf{out}\, h) \circ \mathsf{succ})\, \mathsf{zero})$$

$$\mathsf{tr}\, (\mathsf{in}\, f) \longrightarrow^+ \mathsf{tr}\, ((\mathsf{p} \circ f \circ \mathsf{succ})\, \mathsf{zero})$$

Some calculation now shows that $\mathsf{tr}\,(\mathsf{h}\,\mathsf{zero})$, the application of tr to the "bad guy" h, diverges:

$$
\begin{array}{lll}
\mathsf{tr}\,(\mathsf{h}\,\mathsf{zero}) & \longrightarrow^+ & \mathsf{tr}\,(\mathsf{in}\,(\mathsf{s} \circ \mathsf{h} \circ \mathsf{pred})) \\
& \longrightarrow^+ & \mathsf{tr}\,((\mathsf{p} \circ \mathsf{s} \circ \mathsf{h} \circ \mathsf{pred} \circ \mathsf{succ})\,\mathsf{zero}) \qquad \longrightarrow^+ \\
\mathsf{tr}\,(\mathsf{p}\,(\mathsf{s}\,(\mathsf{h}\,\mathsf{zero}))) & \longrightarrow^+ & \mathsf{tr}\,(\mathsf{p}\,(\mathsf{s}\,(\mathsf{in}\,(\mathsf{s} \circ \mathsf{h} \circ \mathsf{pred}))))) \\
& \longrightarrow^+ & \mathsf{tr}\,(\mathsf{p}\,(\mathsf{in}\,(\mathsf{s}^2 \circ \mathsf{h} \circ \mathsf{pred}^2))) \\
& \longrightarrow^+ & \mathsf{tr}\,(\mathsf{in}\,(\mathsf{p} \circ \mathsf{s}^2 \circ \mathsf{h} \circ \mathsf{pred}^2 \circ \mathsf{succ})) \\
& \longrightarrow^+ & \mathsf{tr}\,((\mathsf{p}^2 \circ \mathsf{s}^2 \circ \mathsf{h} \circ \mathsf{pred}^2 \circ \mathsf{succ}^2)\,\mathsf{zero}) \quad \longrightarrow^+ \\
\mathsf{tr}\,(\mathsf{p}^2\,(\mathsf{s}^2\,(\mathsf{h}\,\mathsf{zero}))) & \longrightarrow^+ & \ldots
\end{array}
$$

6. A Kinding System for Semi-Continuity

We turn the results of Section 4 into a calculus and define a judgement $\Delta; \Pi \vdash^{\imath q} F : \kappa$, where $\imath$ is an ordinal variable with $(\imath : p\mathsf{ord}) \in \Delta$ for some p, the bit $q \in \{\ominus, \oplus\}$ states whether the constructor F under consideration is lower ($\ominus$) or upper ($\oplus$) semi-continuous, and Π is a context of *strictly positive* constructor variables $X : +\kappa'$. We will prove (Thm. 6.2) that the family $F(\imath)$ is lim sup-pushable in all $X \in \Pi$ if $q = \oplus$ and lim inf-pullable if $q = \ominus$.

The complete listing of rules can be found in Figure 3; in the following, we discuss a few.

$$\textsc{cont-co} \quad \frac{\Delta, \imath : +\mathsf{ord} \vdash F : \kappa \qquad p \in \{+, \circ\}}{\Delta, \imath : p\mathsf{ord}; \Pi \vdash^{\imath\oplus} F : \kappa}$$

If $\imath$ appears only positively in F, then F is trivially upper semi-continuous. However, monotonicity does not imply lim sup-pushability, so no variables from Π may occur in F. In the conclusion we may choose to set $p = \circ$, meaning that we forget that F is monotone in $\imath$. Rule CONT-CONTRA is analogous and rule CONT-IN states that a constant F is (trivially) continuous.

$$\textsc{cont-var} \quad \frac{X : p\kappa \in \Delta, \Pi \qquad p \le +}{\Delta; \Pi \vdash^{\imath q} X : \kappa}$$

Rule CONT-VAR can be used also for $X = \iota$. It states that the identity is continuous and both lim sup-pushable and lim inf-pullable.

Using the four rules discussed so far, we can derive semi-continuity properties of ordinal expressions. Expressions like ∞ and $\mathsf{s}^n\jmath$ (with $\jmath \ne \imath$) which are constant in ι are trivially continuous; so is the identity ι. Expressions of the form $\mathsf{s}^n\imath$ with $n \ge 1$ are only upper semi-continuous (rule CONT-CO), but not lower semi-continuous.

Now we discuss some rules to construct semi-continuous types and type constructors.

$$\textsc{cont-arr} \quad \frac{-\Delta; \diamond \vdash^{\imath\ominus} A : * \qquad \Delta; \Pi \vdash^{\imath\oplus} B : *}{\Delta; \Pi \vdash^{\imath\oplus} A \to B : *}$$

Strictly positive contexts.
$$\Pi \quad ::= \quad \diamond \mid \Pi, X\!:\!+\kappa_*$$
Semi-continuity $\Delta; \Pi \vdash^{iq} F : \kappa$ for $q \in \{\oplus, \ominus\}$.

$$\text{CONT-CO} \ \frac{\Delta, i\!:\!+\text{ord} \vdash F : \kappa \qquad p \le +}{\Delta, i\!:\!p\text{ord}; \Pi \vdash^{i\oplus} F : \kappa} \qquad\qquad \text{CONT-CONTRA} \ \frac{\Delta, i\!:\!-\text{ord} \vdash F : \kappa \qquad p \le -}{\Delta, i\!:\!p\text{ord}; \Pi \vdash^{i\ominus} F : \kappa}$$

$$\text{CONT-IN} \ \frac{\Delta \vdash F : \kappa}{\Delta, i\!:\!p\text{ord}; \Pi \vdash^{iq} F : \kappa} \qquad\qquad \text{CONT-VAR} \ \frac{X\!:\!p\kappa \in \Delta, \Pi \qquad p \le +}{\Delta; \Pi \vdash^{iq} X : \kappa}$$

$$\text{CONT-ABS} \ \frac{\Delta, X\!:\!p\kappa; \Pi \vdash^{iq} F : \kappa'}{\Delta; \Pi \vdash^{iq} \lambda X\!:\!\kappa.\, F : p\kappa \to \kappa'} \ X \neq i$$

$$\text{CONT-APP} \ \frac{\Delta, i\!:\!p'\text{ord}; \Pi \vdash^{iq} F : p\kappa \to \kappa' \qquad p^{-1}\Delta \vdash G : \kappa}{\Delta, i\!:\!p'\text{ord}; \Pi \vdash^{iq} F\, G : \kappa'}$$

$$\text{CONT-SUM} \ \frac{\Delta; \Pi \vdash^{iq} A : * \qquad \Delta; \Pi \vdash^{iq} B : *}{\Delta; \Pi \vdash^{iq} A + B : *}$$

$$\text{CONT-PROD} \ \frac{\Delta; \Pi \vdash^{iq} A : * \qquad \Delta; \Pi \vdash^{iq} B : *}{\Delta; \Pi \vdash^{iq} A \times B : *}$$

$$\text{CONT-ARR} \ \frac{-\Delta; \diamond \vdash^{i\ominus} A : * \qquad \Delta; \Pi \vdash^{i\oplus} B : *}{\Delta; \Pi \vdash^{i\oplus} A \to B : *} \qquad \text{CONT-}\forall \ \frac{\Delta; \Pi \vdash^{i\oplus} F : \circ\kappa \to *}{\Delta; \Pi \vdash^{i\oplus} \forall_\kappa F : *}$$

$$\text{CONT-MU} \ \frac{\Delta; \Pi, X\!:\!+\kappa_* \vdash^{i\ominus} F : \kappa_* \qquad \Delta \vdash a : \text{ord}}{\Delta; \Pi \vdash^{i\ominus} \mu^a(\lambda X\!:\!\kappa_*.\, F) : \kappa_*} \ a = i \text{ or } i \notin \mathsf{FV}(a)$$

$$\text{CONT-NU} \ \frac{\Delta; \Pi, X\!:\!+\kappa_* \vdash^{i\oplus} F : \kappa_* \qquad \Delta \vdash a\ \text{ord}}{\Delta; \Pi \vdash^{i\oplus} \nu^a(\lambda X\!:\!\kappa_*.\, F) : \kappa_*}$$

Pure ordinal expressions $\Delta \vdash a\ ord$.

$$\text{ORD-}\infty \ \frac{}{\Delta \vdash \infty\ ord} \qquad \text{ORD-VAR} \ \frac{(i\!:\!p\text{ord}) \in \Delta \qquad p \le +}{\Delta \vdash i\ ord} \qquad \text{ORD-s} \ \frac{\Delta \vdash a\ ord}{\Delta \vdash \mathsf{s}\, a\ ord}$$

Figure 3: $\mathsf{F}_{\widehat{\omega}}$: Semi-continuous constructors and recursion types.

This rule incarnates Thm. 4.7. Note that, because A is to the left of the arrow, the polarity of all ordinary variables in A is reversed, and A may not contain strictly positive variables.

$$\text{CONT-NU} \ \frac{\Delta; \Pi, X\!:\!+\kappa_* \vdash^{i\oplus} F : \kappa_* \qquad \Delta \vdash a\ ord}{\Delta; \Pi \vdash^{i\oplus} \nu^a(\lambda X\!:\!\kappa_*.\, F) : \kappa_*}$$

Rule CONT-NU states that strictly positive coinductive types are upper semi-continuous. The ordinal a must be ∞ or $\mathsf{s}^n j$ for some $j\!:\!\text{ord} \in \Delta$ (which may also be identical to i).

Lemma 6.1. *Assume $\Delta \vdash a$ ord and let $\theta \in [\![\Delta]\!]$, $\imath$ an ordinal variables, and*

$$\phi := (\alpha \mapsto [\![a]\!]_{\theta[\imath \mapsto \alpha]}) \in [\![\text{ord}]\!] \to [\![\text{ord}]\!].$$

Then for all limit ordinals $\lambda \geq \omega$, either $\liminf_\lambda \phi = \phi(\lambda)$, or both sides are $\geq \omega$.

Proof. ϕ is either constant or affine ($\phi(\alpha) = \min \alpha + n, \top^{\text{ord}}$ for some $n \in \mathbb{N}$), and such ϕ have the required property. $\qquad\square$

Theorem 6.2 (Soundness of Continuity Derivations). *Assume $\Delta; \vec{X}: +\vec{\kappa} \vdash^{\imath q} F : \kappa$. Let θ a valuation of the variables in Δ and set $\mathcal{F}_\alpha(\vec{\mathcal{G}}) = [\![F]\!]_{\theta[\imath \mapsto \alpha][\vec{X} \mapsto \vec{\mathcal{G}}]}$.*

    (1) *If $q = \ominus$ then the family $\mathcal{F}$ is $\liminf$-pullable.*
    (2) *If $q = \oplus$ then the family $\mathcal{F}$ is $\limsup$-pushable.*

Proof. By induction on the continuity derivation. Some cases:

$$\text{CONT-NU} \quad \frac{\Delta; \vec{X}:\vec{\kappa}, X:+\kappa_* \vdash^{\imath\oplus} F : \kappa_* \qquad \Delta \vdash a \text{ ord}}{\Delta; \vec{X}:\vec{\kappa} \vdash^{\imath\oplus} \nu^a(\lambda X:\kappa_*. F) : \kappa_*}$$

Let $\mathcal{F}_\alpha(\vec{\mathcal{G}})(\mathcal{H}) = [\![F]\!]_{\theta[\imath \mapsto \alpha][\vec{X} \mapsto \vec{\mathcal{G}}][X \mapsto \mathcal{H}]}$ and $\phi(\alpha) = [\![a]\!]_{\theta[\imath \mapsto \alpha]}$. By Lemma 6.1, ϕ is either constant or affine, and by induction hypothesis, $\mathcal{F}$ is $\limsup$-pushable. Thus, we can apply Thm. 4.15 to infer the goal.

$$\text{CONT-MU} \quad \frac{\Delta; \Pi, X:+\kappa_* \vdash^{\imath\ominus} F : \kappa_* \qquad \Delta \vdash a : \text{ord}}{\Delta; \Pi \vdash^{\imath\ominus} \mu^a(\lambda X:\kappa_*. F) : \kappa_*} \quad a = \imath \text{ or } \imath \notin \mathsf{FV}(a)$$

$\phi(\alpha) := [\![a]\!]_{\theta[\imath \mapsto \alpha]}$ is either constant or the identity, hence, it is monotone and continuous. The goal follows from Thm. 4.18. $\qquad\square$

Now we are able to formulate the syntactical admissibility criterion for types of (co)recursive functions.

Definition 6.3 (Syntactic admissibility).

$\Gamma \vdash A \; \mathsf{fix}_n^\mu\text{-adm}$ iff $\Gamma, \imath:\circ\text{ord} \vdash A \, \imath = \forall \vec{X}:\vec{\kappa}.B_1 \to \cdots \to B_n \to \mu^\imath F\vec{H} \to C : *$
                      and $\Gamma, \imath:\circ\text{ord}; \diamond \vdash^{\imath\oplus} \forall \vec{X}:\vec{\kappa}. B_{1..n} \to \mu^\imath F\vec{H} \to C : *$

$\Gamma \vdash A \; \mathsf{fix}_n^\nu\text{-adm}$ iff $\Gamma, \imath:\circ\text{ord} \vdash A \, \imath = \forall \vec{X}:\vec{\kappa}.B_1 \to \cdots \to B_n \to \nu^\imath F\vec{H} : *$
                      and $\Gamma, \imath:\circ\text{ord}; \diamond \vdash^{\imath\oplus} \forall \vec{X}:\vec{\kappa}. B_{1..n} \to \nu^\imath F\vec{H} : *$

It is easy to check that admissible types fulfill the semantic criteria given at the end of Section 3. We prove Assumption 3.5, restated as the following theorem.

Theorem 6.4 (Soundness of admissibility). *If $\Gamma \vdash A \; \mathsf{fix}_n^\nabla\text{-adm}$ and $\theta(X) \in [\![\kappa]\!]$ for all $(X:\kappa) \in [\![\Gamma]\!]$ then $\mathcal{A} := [\![A]\!]_\theta \in [\![\text{ord}]\!] \to [\![*]\!]$ has the following properties:*

    (1) *Shape: $\mathcal{A}(\alpha) = \bigcap_{k \in K} \mathcal{B}_1(k, \alpha) \boxed{\to} \ldots \boxed{\to} \mathcal{B}_n(k, \alpha) \boxed{\to} \mathcal{B}(k, \alpha)$ for some K and some $\mathcal{B}_1, \ldots, \mathcal{B}_n, \mathcal{B} \in K \times [\![\text{ord}]\!] \to [\![*]\!]$. In case $\nabla = \mu$, $\mathcal{B}(k, \alpha) = \mathcal{I}(k, \alpha)^\mu \boxed{\to} \mathcal{C}(k, \alpha)$ for some $\mathcal{I}, \mathcal{C}$. Otherwise, $\mathcal{B}(k, \alpha) = \mathcal{C}(k, \alpha)^\nu$ for some $\mathcal{C}$.*
    (2) *Bottom-check: $\mathcal{I}(k, 0)^\mu = \bot^*$ in case $\nabla = \mu$ and $\mathcal{C}(k, 0)^\nu = \top^*$ in case $\nabla = \nu$.*
    (3) *Limit-check: $\inf_{\alpha < \lambda} \mathcal{A}(\alpha) \subseteq \mathcal{A}(\lambda)$ for all limit ordinals $\lambda \in [\![\text{ord}]\!] \setminus \{0\}$.*

Proof. Set $K := [\![\kappa_1]\!] \times \ldots [\![\kappa_m]\!]$ with $m := |\vec{\kappa}|$ and $\mathcal{B}_i(k, \alpha) := [\![B_i]\!]_{\theta[\vec{X} \mapsto k][\imath \mapsto \alpha]}$ for $i = 1..n$. Further, let $\mathcal{F}(k)(\alpha) := [\![F]\!]_{\theta[\vec{X} \mapsto k][\imath \mapsto \alpha]}$ and $\mathcal{H}_j(k)(\alpha) := [\![H_j]\!]_{\theta[\vec{X} \mapsto k][\imath \mapsto \alpha]}$.

In case $\nabla = \mu$, first let $\mathcal{C}(k,\alpha) = [\![C]\!]_{\theta[\vec{X}\mapsto k][\imath\mapsto\alpha]}$. Define $\emptyset^{\vec{p\kappa}\to *}(\vec{\mathcal{G}}) = \emptyset$ and observe that for any $f \in [\![\kappa]\!] \overset{+}{\to} [\![\kappa]\!]$, $\boldsymbol{\mu}^\alpha f^\mu = (f^\mu)^\alpha(\bot^\kappa) = ((f \circ (-)^\mu)^\alpha(\emptyset^\kappa))^\mu$. (Induction on α, using $\emptyset^\mu = \bot$ and sup-continuity of $(-)^\mu$.) Thus, we can set

$$\mathcal{I}(k,\alpha) = (((\mathcal{F}(k)(\alpha) \circ (-)^\mu)^\alpha(\emptyset))(\vec{\mathcal{H}})$$

and have $\mathcal{I}(k,\alpha)^\mu = [\![\mu^\imath F \vec{H}]\!]_{\theta[\vec{X}\mapsto k][\imath\mapsto\alpha]}$. Properties (1) and (2) are hence satisfied. By Thm. 6.2, the type $\mathcal{A}$ is upper semi-continuous which implies (3).

For case $\nabla = \nu$, define $(\mathsf{out} \cdot \top^{\vec{p\kappa}\to *})(\vec{\mathcal{G}}) = \mathsf{out} \cdot \top^*$. Then $(\mathsf{out} \cdot \top)^\nu = \top$. Observe that $\boldsymbol{\nu}^\alpha f^\nu = (f^\nu)^\alpha(\top) = ((f \circ (-)^\nu)^\alpha(\mathsf{out} \cdot \top))^\nu$ and set

$$\mathcal{C}(k,\alpha) = (((\mathcal{F}(k)(\alpha) \circ (-)^\nu)^\alpha(\mathsf{out} \cdot \top))(\vec{\mathcal{H}}).$$

Then $\mathcal{C}(k,\alpha)^\nu = [\![\nu^\imath F \vec{H}]\!]_{\theta[\vec{X}\mapsto k][\imath\mapsto\alpha]}$. Using Thm. 6.2, all three properties hold. $\qquad\square$

Example 6.5 (Inductive type inside coinductive type). Rule CONT-NU allows the type system to accept the following definition, which assigns an informative type to the stream nats of all natural numbers in ascending order:

$$
\begin{aligned}
\mathsf{nats} \quad &: \quad \forall\imath.\, \mathsf{Stream}^\imath\,\mathsf{Nat}^\imath \\
\mathsf{nats} \quad &:= \quad \mathsf{fix}_0^\nu \lambda nats.\, \langle \mathsf{zero},\ \mathsf{mapStream\ succ}\ nats \rangle \\[4pt]
\mathsf{mapStream} \quad &: \quad \forall A\forall B.\,(A \to B) \to \forall\imath.\, \mathsf{Stream}^\imath A \to \mathsf{Stream}^\imath B \\
\mathsf{mapStream} \quad &:= \quad \lambda f \mathsf{fix}_1^\nu \lambda maps \lambda s.\, \mathsf{in}\langle f\,(\mathsf{fst}\,(\mathsf{out}\,s)),\ maps\,(\mathsf{snd}\,(\mathsf{out}\,s)) \rangle
\end{aligned}
$$

The type of nats expresses that if you read the first n elements of the stream, these are numbers $< n$. In particular, the ith element of nats is at most $i - 1$. This is the most information a type of nats can carry in our type system.

Example 6.6 (Inductive type inside inductive type). In the following, we describe breadth-first traversal of rose (finitely branching) trees whose termination is recognized by $\widehat{\mathsf{F}_\omega}$.

$$
\begin{aligned}
\mathsf{Rose} \quad &: \quad \mathsf{ord} \overset{+}{\to} * \overset{+}{\to} * \\
\mathsf{Rose} \quad &:= \quad \lambda\imath\lambda A.\, \mathsf{GRose}^\imath\,\mathsf{List}^\infty A = \lambda\imath\lambda A.\, \mu_*^\imath\lambda X.\, A \times \mathsf{List}^\infty X
\end{aligned}
$$

The step function, defined by induction on $\jmath$, traverses a list of rose trees of height $< \imath + 1$ and produces a list of the roots and a list of the branches (height $< \imath$).

$$
\begin{aligned}
\mathsf{step} \quad &: \quad \forall\jmath\forall A\forall\imath.\, \mathsf{List}^\jmath(\mathsf{Rose}^{\imath+1} A) \to \mathsf{List}^\jmath A \times \mathsf{List}^\infty(\mathsf{Rose}^\imath A) \\
\mathsf{step} \quad &:= \quad \mathsf{fix}_0^\mu \lambda step \lambda l.\, \mathsf{match}\ l\ \mathsf{with} \\
&\qquad\qquad \mathsf{nil} \mapsto \langle \mathsf{nil},\,\mathsf{nil} \rangle \\
&\qquad\qquad \mathsf{cons}\,\langle a, rs'\rangle\,rs \mapsto \mathsf{match}\ step\ rs\ \mathsf{with} \\
&\qquad\qquad\qquad\qquad \langle as,\, rs''\rangle \mapsto \langle \mathsf{cons}\,a\,as,\ \mathsf{append}\,rs'\,rs''\rangle
\end{aligned}
$$

Now, bf iterates step on a non-empty forest, which is represented by a single rose tree r and a possibly empty list of rose trees rs. It is defined by induction on $\imath$, the strict upper bound of the tree heights.

$$
\begin{aligned}
\mathsf{bf} \quad &: \quad \forall\imath\forall A.\, \mathsf{Rose}^\imath A \to \mathsf{List}^\infty(\mathsf{Rose}^\imath A) \to \mathsf{List}^\infty A \\
\mathsf{bf} \quad &:= \quad \mathsf{fix}_0^\mu \lambda bf \lambda r \lambda rs.\, \mathsf{match\ step}\,(\mathsf{cons}\,r\,rs)\ \mathsf{with} \\
&\qquad\qquad \langle as,\,\mathsf{nil}\rangle \qquad\quad\; \mapsto \quad as \\
&\qquad\qquad \langle as,\,\mathsf{cons}\,r'\,rs'\rangle \;\; \mapsto \quad \mathsf{append}\ as\ (bf\ r'\ rs')
\end{aligned}
$$

Function bf terminates because the recursive-call trees in forest $\mathsf{cons}\,r'\,rs'$ are smaller than the input trees in forest $\mathsf{cons}\,r\,rs$. This information is available to the type system through

the type of step. The type of bf is admissible for recursion since $\mathsf{List}^\infty\,(\mathsf{Rose}^\imath\,A)$ is lower semi-continuous in $\imath$—thanks to Thm. 4.18 and rule CONT-MU.

It is clear that admissibility is no way complete. One can find trivial examples of terminating programs which are refused by the type system because they fail the admissibility check. For instance, take the recursive identity function of type $\forall\imath.\,\mathsf{Nat}^\imath \to \mathsf{Nat}^\imath$ and add an unused argument of type $\mathsf{Nat}^\infty \to \mathsf{Nat}^\imath$:

$$\mathsf{loopnot} : \forall\imath.\,\mathsf{Nat}^\imath \to (\mathsf{Nat}^\infty \to \mathsf{Nat}^\imath) \to \mathsf{Nat}^\imath$$
$$\mathsf{loopnot}\;0 \qquad g \;\; = \;\; 0$$
$$\mathsf{loopnot}\;(n+1)\;g \;\; = \;\; 1 + \mathsf{loopnot}\,n\,(\mathsf{shift}\,g)$$

Its type is not upper semi-continuous, but of course loopnot is terminating.

7. CONCLUSIONS

We have motivated the importance of semi-continuity for the soundness of type-based termination checking, explored the realm of semi-continuous functions from ordinals to semantic types, and developed a calculus for semi-continuous types. We have seen a few interesting examples involving semi-continuous types, many more can be found in the author's thesis [Abe06b, Ch. 6]. These examples cannot be handled by type-based termination à la Barthe et al. [BFG$^+$04, BGP05], but our developments could be directly incorporated into their calculus.

In previous work [Abe03], I have already presented a calculus for admissible recursion types. But the language had neither polymorphism, higher-kinded types, nor semi-continuous types inside each other ($\mathsf{Stream}^\imath\,\mathsf{Nat}^\imath$). Hughes, Pareto, and Sabry [HPS96] have also given criteria for admissible types similar to ours, but rather ad-hoc ones, not based on the mathematical concept of semi-continuity. Also, a crucial difference is that we also treat *infinitely* branching data structures. To be fair, I should say that their work has been a major source of inspiration for me.

As a further direction of research, I propose to develop a kinding system where semi-continuity is first class, i.e., one can abstract over semi-continuous constructors, and kind arrows can carry the corresponding polarities $\ominus$ or $\oplus$. First attempts suggest that such a calculus is not straightforward, and a more fine-grained polarity system will be necessary. Important is also the study of semi-continuity properties of dependent types, in order to apply these results to type-based termination in type-theoretic proof assistants.

REFERENCES

[Abe03] Andreas Abel. Termination and guardedness checking with continuous types. In M. Hofmann, editor, *Typed Lambda Calculi and Applications (TLCA 2003), Valencia, Spain*, volume 2701 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, June 2003.

[Abe04] Andreas Abel. Termination checking with types. *RAIRO – Theoretical Informatics and Applications*, 38(4):277–319, 2004. Special Issue: Fixed Points in Computer Science (FICS'03).

[Abe06a]   Andreas Abel. Polarized subtyping for sized types. In Dima Grigoriev, John Harrison, and Edward A. Hirsch, editors, *Computer Science - Theory and Applications, First International Computer Science Symposium in Russia, CSR 2006, St. Petersburg, Russia, June 8-12, 2006, Proceedings*, volume 3967 of *Lecture Notes in Computer Science*, pages 381–392. Springer-Verlag, 2006.

[Abe06b]   Andreas Abel. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types*. PhD thesis, Ludwig-Maximilians-Universität München, 2006.

[Abe06c]   Andreas Abel. Semi-continuous sized types and termination. In Zoltán Ésik, editor, *Computer Science Logic, 20th International Workshop, CSL 2006, 15th Annual Conference of the EACSL, Szeged, Hungary, September 21-24, 2006, Proceedings*, volume 4207 of *Lecture Notes in Computer Science*, pages 72–88. Springer-Verlag, 2006.

[ACG98]   Roberto M. Amadio and Solange Coupet-Grimal. Analysis of a guard condition in type theory (extended abstract). In Maurice Nivat, editor, *Foundations of Software Science and Computation Structure, First International Conference, FoSSaCS'98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1378 of *Lecture Notes in Computer Science*, pages 48–62. Springer-Verlag, 1998.

[Alt01]   Thorsten Altenkirch. Representations of first order function types as terminal coalgebras. In *Typed Lambda Calculi and Applications, TLCA 2001*, number 2044 in Lecture Notes in Computer Science, pages 8 – 21. Springer-Verlag, 2001.

[AM04]   Andreas Abel and Ralph Matthes. Fixed points of type constructors and primitive recursion. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic, 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Karpacz, Poland, September 20-24, 2004, Proceedings*, volume 3210 of *Lecture Notes in Computer Science*, pages 190–204. Springer-Verlag, 2004.

[AR99]   Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In Jörg Flum and Mario Rodríguez-Artalejo, editors, *Computer Science Logic, 13th International Workshop, CSL '99, 8th Annual Conference of the EACSL, Madrid, Spain, September 20-25, 1999, Proceedings*, volume 1683 of *Lecture Notes in Computer Science*, pages 453–468. Springer-Verlag, 1999.

[BFG⁺04]   Gilles Barthe, Maria J. Frade, Eduardo Giménez, Luis Pinto, and Tarmo Uustalu. Type-based termination of recursive definitions. *Mathematical Structures in Computer Science*, 14(1):1–45, 2004.

[BGP05]   Gilles Barthe, Benjamin Grégoire, and Fernando Pastawski. Practical inference for type-based termination in a polymorphic setting. In Pawel Urzyczyn, editor, *Typed Lambda Calculi and Applications (TLCA 2005), Nara, Japan*, volume 3461 of *Lecture Notes in Computer Science*, pages 71–85. Springer-Verlag, 2005.

[Bla04]   Frédéric Blanqui. A type-based termination criterion for dependently-typed higher-order rewrite systems. In Vincent van Oostrom, editor, *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3 – 5, 2004, Proceedings*, volume 3091 of *Lecture Notes in Computer Science*, pages 24–39. Springer-Verlag, 2004.

[Bla05]   Frédéric Blanqui. Decidability of type-checking in the Calculus of Algebraic Constructions with size annotations. In C.-H. Luke Ong, editor, *Computer Science Logic, 19th International Workshop, CSL 2005, 14th Annual Conference of the EACSL, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3634 of *Lecture Notes in Computer Science*, pages 135–150. Springer-Verlag, 2005.

[BP99]   Richard S. Bird and Ross Paterson. De Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–91, 1999.

[CW99]   Karl Crary and Stephanie Weirich. Flexible type analysis. In *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, France*, volume 34 of *SIGPLAN Notices*, pages 233–248. ACM Press, 1999.

[DC99]   Dominic Duggan and Adriana Compagnoni. Subtyping for object type constructors, January 1999. Presented at FOOL 6.

[Gim98]   Eduardo Giménez. Structural recursive definitions in type theory. In K. G. Larsen, S. Skyum, and G. Winskel, editors, *Automata, Languages and Programming, 25th International Colloquium,*

ICALP'98, Aalborg, Denmark, July 13-17, 1998, Proceedings, volume 1443 of *Lecture Notes in Computer Science*, pages 397–408. Springer-Verlag, 1998.

[Han02]   Peter Hancock. The step to the next number class. http://www.dcs.ed.ac.uk/home/pgh/number-classes.html, 2002.

[Hin00a] Ralf Hinze. Efficient generalized folds. In Johan Jeuring, editor, *Proceedings of the Second Workshop on Generic Programming, WGP 2000*, Ponte de Lima, Portugal, July 2000.

[Hin00b] Ralf Hinze. Generalizing generalized tries. *Journal of Functional Programming*, 10(4):327–351, July 2000.

[HPS96]   John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'96*, pages 410–423, 1996.

[Men87]   Nax P. Mendler. Recursive types and type constraints in second-order lambda calculus. In *Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science, Ithaca, N.Y.*, pages 30–36. IEEE Computer Society Press, 1987.

[Par00]   Lars Pareto. *Types for Crash Prevention*. PhD thesis, Chalmers University of Technology, 2000.

[Pie02]   Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[PM92]   Christine Paulin-Mohring. Inductive definitions in the system Coq—rules and properties. Technical report, Laboratoire de l'Informatique du Parallélisme, December 1992.

[Ste98]   Martin Steffen. *Polarized Higher-Order Subtyping*. PhD thesis, Technische Fakultät, Universität Erlangen, 1998.

[Vou04]   Jérôme Vouillon. Subtyping union types. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic, 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Karpacz, Poland, September 20-24, 2004, Proceedings*, volume 3210 of *Lecture Notes in Computer Science*, pages 415–429. Springer-Verlag, 2004.

[Xi01]   Hongwei Xi. Dependent types for program termination verification. In *Proceedings of 16th IEEE Symposium on Logic in Computer Science*, Boston, USA, June 2001.

APPENDIX A. COMPLETE SPECIFICATION OF $\widehat{\mathsf{F}_\omega}$

The following figures display all constructs and rules of $\widehat{\mathsf{F}_\omega}$.

Syntactic categories.

$$
\begin{array}{llll}
p & ::= & + \mid - \mid \circ & \text{polarity} \\
\kappa & ::= & * \mid \mathsf{ord} \mid p\kappa \to \kappa' & \text{kind} \\
\kappa_* & ::= & * \mid p\kappa_* \to \kappa'_* & \text{pure kind} \\
a,b,A,B,F,G & ::= & C \mid X \mid \lambda X{:}\kappa.\, F \mid F\,G & \text{(type) constructor} \\
C & ::= & 1 \mid + \mid \times \mid \to \mid \forall_\kappa \mid \mu_{\kappa_*} \mid \nu_{\kappa_*} \mid \mathsf{s} \mid \infty & \text{constructor constant} \\
\Delta & ::= & \diamond \mid \Delta, X{:}p\kappa & \text{polarized context}
\end{array}
$$

The signature Σ assigns kinds to constants ($\kappa \xrightarrow{p} \kappa'$ means $p\kappa \to \kappa'$).

$$
\begin{array}{lll}
1 & : & * \qquad\qquad\qquad\qquad\qquad\quad \text{unit type} \\
+ & : & * \xrightarrow{+} * \xrightarrow{+} * \qquad\qquad\qquad \text{disjoint sum} \\
\times & : & * \xrightarrow{+} * \xrightarrow{+} * \qquad\qquad\qquad \text{cartesian product} \\
\to & : & * \xrightarrow{-} * \xrightarrow{+} * \qquad\qquad\qquad \text{function space} \\
\forall_\kappa & : & (\kappa \xrightarrow{\circ} *) \xrightarrow{+} * \qquad\qquad \text{quantification} \\
\mu_{\kappa_*} & : & \mathsf{ord} \xrightarrow{+} (\kappa_* \xrightarrow{+} \kappa_*) \xrightarrow{+} \kappa_* \quad \text{inductive constructors} \\
\nu_{\kappa_*} & : & \mathsf{ord} \xrightarrow{-} (\kappa_* \xrightarrow{+} \kappa_*) \xrightarrow{+} \kappa_* \quad \text{coinductive constructors} \\
\mathsf{s} & : & \mathsf{ord} \xrightarrow{+} \mathsf{ord} \qquad\qquad\qquad \text{successor of ordinal} \\
\infty & : & \mathsf{ord} \qquad\qquad\qquad\qquad\quad\; \text{infinity ordinal}
\end{array}
$$

Notation.

$$\nabla \;\text{ for }\; \mu \text{ or } \nu \qquad \nabla^a \;\text{ for }\; \nabla a$$

$$\forall X{:}\kappa.\, A \;\text{ for }\; \forall_\kappa(\lambda X{:}\kappa.\, A) \qquad \forall X A \;\text{ for }\; \forall X{:}\kappa.\, A \qquad \lambda X F \;\text{ for }\; \lambda X{:}\kappa.\, F$$

$$A + B \;\text{ for }\; + A\,B \qquad A \times B \;\text{ for }\; \times A\,B \qquad A \to B \;\text{ for }\; \to A\,B$$

Ordering and composition of polarities.

$$p \le p \qquad \circ \le p \qquad +p = p \qquad -- = + \qquad \circ p = \circ \qquad pp' = p'p$$

Inverse application of a polarity to a context.

$$
\begin{array}{llllll}
p^{-1}\diamond & = & \diamond & \quad \circ^{-1}(\Delta, X{:}\circ\kappa) & = & (\circ^{-1}\Delta), X{:}\circ\kappa \\
+^{-1}\Delta & = & \Delta & \quad \circ^{-1}(\Delta, X{:}+\kappa) & = & \circ^{-1}\Delta \\
-^{-1}(\Delta, X{:}p\kappa) & = & (-^{-1}\Delta), X{:}(-p)\kappa & \quad \circ^{-1}(\Delta, X{:}-\kappa) & = & \circ^{-1}\Delta
\end{array}
$$

Kinding $\Delta \vdash F : \kappa$.

$$\textsc{kind-c}\; \frac{C{:}\kappa \in \Sigma}{\Delta \vdash C : \kappa} \qquad \textsc{kind-var}\; \frac{X{:}p\kappa \in \Delta \qquad p \le +}{\Delta \vdash X : \kappa}$$

$$\textsc{kind-abs}\; \frac{\Delta, X{:}p\kappa \vdash F : \kappa'}{\Delta \vdash \lambda X{:}\kappa.\, F : p\kappa \to \kappa'} \qquad \textsc{kind-app}\; \frac{\Delta \vdash F : p\kappa \to \kappa' \qquad p^{-1}\Delta \vdash G : \kappa}{\Delta \vdash F\,G : \kappa'}$$

Figure 4: $\mathsf{F}\widehat{\omega}$: Kinds and constructors.

Constructor equality $\Delta \vdash F = F' : \kappa$.

$$\text{EQ-}\infty \ \frac{}{\Delta \vdash \mathsf{s}\,\infty = \infty : \mathsf{ord}}$$

$$\text{EQ-}\beta \ \frac{\Delta, X\!:\!p\kappa \vdash F : \kappa' \qquad p^{-1}\Delta \vdash G : \kappa}{\Delta \vdash (\lambda X\!:\!\kappa.\,F)\,G = [G/X]F : \kappa'} \qquad \text{EQ-}\eta \ \frac{\Delta \vdash F : p\kappa \to \kappa'}{\Delta \vdash (\lambda X\!:\!\kappa.\,F\,X) = F : p\kappa \to \kappa'}$$

$$\text{EQ-VAR} \ \frac{X\!:\!p\kappa \in \Delta \qquad p \le +}{\Delta \vdash X = X : \kappa} \qquad \text{EQ-}\lambda \ \frac{\Delta, X\!:\!p\kappa \vdash F = F' : \kappa'}{\Delta \vdash \lambda X\!:\!\kappa.\,F = \lambda X\!:\!\kappa.\,F' : p\kappa \to \kappa'}$$

$$\text{EQ-C} \ \frac{C\!:\!\kappa \in \Sigma}{\Delta \vdash C = C : \kappa} \qquad \text{EQ-APP} \ \frac{\Delta \vdash F = F' : p\kappa \to \kappa' \qquad p^{-1}\Delta \vdash G = G' : \kappa}{\Delta \vdash F\,G = F'\,G' : \kappa'}$$

$$\text{EQ-SYM} \ \frac{\Delta \vdash F = F' : \kappa}{\Delta \vdash F' = F : \kappa} \qquad \text{EQ-TRANS} \ \frac{\Delta \vdash F_1 = F_2 : \kappa \qquad \Delta \vdash F_2 = F_3 : \kappa}{\Delta \vdash F_1 = F_3 : \kappa}$$

Constructor subtyping $\Delta \vdash F \le F' : \kappa$.

$$\text{LEQ-S-R} \ \frac{\Delta \vdash a : \mathsf{ord}}{\Delta \vdash a \le \mathsf{s}\,a : \mathsf{ord}} \qquad \text{LEQ-}\infty \ \frac{\Delta \vdash a : \mathsf{ord}}{\Delta \vdash a \le \infty : \mathsf{ord}}$$

$$\text{LEQ-}\lambda \ \frac{\Delta, X\!:\!p\kappa \vdash F \le F' : \kappa'}{\Delta \vdash \lambda X\!:\!\kappa.\,F \le \lambda X\!:\!\kappa.\,F' : p\kappa \to \kappa'}$$

$$\text{LEQ-APP} \ \frac{\Delta \vdash F \le F' : p\kappa \to \kappa' \qquad p^{-1}\Delta \vdash G : \kappa}{\Delta \vdash F\,G \le F'\,G : \kappa'}$$

$$\text{LEQ-APP+} \ \frac{\Delta \vdash F : +\kappa \to \kappa' \qquad \Delta \vdash G \le G' : \kappa}{\Delta \vdash F\,G \le F\,G' : \kappa'}$$

$$\text{LEQ-APP-} \ \frac{\Delta \vdash F : -\kappa \to \kappa' \qquad -^{-1}\Delta \vdash G' \le G : \kappa}{\Delta \vdash F\,G \le F\,G' : \kappa'}$$

$$\text{LEQ-REFL} \ \frac{\Delta \vdash F = F' : \kappa}{\Delta \vdash F \le F' : \kappa} \qquad \text{LEQ-TRANS} \ \frac{\Delta \vdash F_1 \le F_2 : \kappa \qquad \Delta \vdash F_2 \le F_3 : \kappa}{\Delta \vdash F_1 \le F_3 : \kappa}$$

$$\text{LEQ-ANTISYM} \ \frac{\Delta \vdash F \le F' : \kappa \qquad \Delta \vdash F' \le F : \kappa}{\Delta \vdash F = F' : \kappa}$$

Figure 5: $\mathsf{F}\widehat{_\omega}$: Constructor equality and subtyping.

Syntactic categories.

$$
\begin{array}{llll}
r,s,t & ::= & c \mid x \mid \lambda x t \mid r\,s & \text{term} \\
c & ::= & \langle\rangle \mid \mathsf{pair} \mid \mathsf{fst} \mid \mathsf{snd} \mid \mathsf{inl} \mid \mathsf{inr} \mid \mathsf{case} \mid \mathsf{in} \mid \mathsf{out} \mid \mathsf{fix}_n^\mu \mid \mathsf{fix}_n^\nu & \text{constant } (n \in \mathbb{N}) \\
v & ::= & \lambda x t \mid \langle\rangle \mid \mathsf{pair}\,t_1\,t_2 \mid \mathsf{inl}\,t \mid \mathsf{inr}\,t \mid \mathsf{in}\,t \mid c \mid \mathsf{pair}\,t \mid \mathsf{fix}_n^\nabla s\,t_{1..m} & \text{value } (m < n) \\
e(\_) & ::= & \_\,s \mid \mathsf{fst}\,\_ \mid \mathsf{snd}\,\_ \mid \mathsf{case}\,\_ \mid \mathsf{out}\,\_ \mid \mathsf{fix}_n^\mu s\,t_{1..n\,\_} & \text{evaluation frame} \\
E(\_) & ::= & e_1(\ldots e_n(\_)\ldots) & \text{eval. cxt.}(n \ge 0) \\
\Gamma & ::= & \diamond \mid \Gamma, x{:}A \mid \Gamma, X{:}p\kappa & \text{typing context}
\end{array}
$$

Notation:
$$
\langle r,s\rangle \ \text{for} \ \mathsf{pair}\,r\,s \qquad t_{1..n} \ \text{for} \ t_1\,t_2\ldots t_n.
$$

Reduction $t \longrightarrow t'$.

$$
\begin{array}{llll}
(\lambda x t)\,s & \longrightarrow & [s/x]t & \\
\mathsf{fst}\,\langle r,s\rangle & \longrightarrow & r & \\
\mathsf{snd}\,\langle r,s\rangle & \longrightarrow & s & \\
\mathsf{case}\,(\mathsf{inl}\,r) & \longrightarrow & \lambda x \lambda y.\,x\,r & \\
\mathsf{case}\,(\mathsf{inr}\,r) & \longrightarrow & \lambda x \lambda y.\,y\,r & \\
\end{array}
$$

$$
\begin{array}{llll}
\mathsf{out}\,(\mathsf{in}\,r) & \longrightarrow & r \\
\mathsf{fix}_n^\mu s\,t_{1..n}\,(\mathsf{in}\,t) & \longrightarrow & s\,(\mathsf{fix}_n^\mu s)\,t_{1..n}\,(\mathsf{in}\,t) \\
\mathsf{out}\,(\mathsf{fix}_n^\nu s\,t_{1..n}) & \longrightarrow & \mathsf{out}\,(s\,(\mathsf{fix}_n^\nu s)\,t_{1..n}) \\
{[s/x]t} & \longrightarrow & [s'/x]t \quad \text{if } s \longrightarrow s'
\end{array}
$$

The signature Σ contains types for some constants:

$$
\begin{array}{lcl}
\mathsf{pair} & : & \forall A \forall B.\ A \to B \to A \times B \\
\mathsf{fst} & : & \forall A \forall B.\ A \times B \to A \\
\mathsf{snd} & : & \forall A \forall B.\ A \times B \to B
\end{array}
\qquad
\begin{array}{lcl}
\langle\rangle & : & 1 \\
\mathsf{inl} & : & \forall A \forall B.\ A \to A + B \\
\mathsf{inr} & : & \forall A \forall B.\ B \to A + B
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{case} & : & \forall A \forall B \forall C.\ A + B \to (A \to C) \to (B \to C) \to C \\[4pt]
\mathsf{in} & : & \forall F{:}\kappa \xrightarrow{+} \kappa.\ \forall G^1{:}\kappa_*^1 \ldots \forall G^n{:}\kappa_*^n.\ \forall \imath{:}\mathsf{ord}.\ F\,(\nabla_\kappa^\imath F)\,\vec{G} \to \nabla_\kappa^{\imath+1} F\,\vec{G} \\[4pt]
\mathsf{out} & : & \forall F{:}\kappa \xrightarrow{+} \kappa.\ \forall G^1{:}\kappa_*^1 \ldots \forall G^n{:}\kappa_*^n.\ \forall \imath{:}\mathsf{ord}.\ \nabla_\kappa^{\imath+1} F\,\vec{G} \to F\,(\nabla_\kappa^\imath F)\,\vec{G}
\end{array}
$$

$$
(\nabla \in \{\mu,\nu\},\ \kappa = \vec{\kappa_*} \xrightarrow{\vec{p}} *)
$$

Well-formed typing contexts.

$$
\text{CXT-EMPTY}\ \frac{}{\diamond\ \mathsf{cxt}} \qquad
\text{CXT-TYVAR}\ \frac{\Gamma\ \mathsf{cxt}}{\Gamma, X{:}\circ\kappa\ \mathsf{cxt}} \qquad
\text{CXT-VAR}\ \frac{\Gamma\ \mathsf{cxt} \qquad \Gamma \vdash A : *}{\Gamma, x{:}A\ \mathsf{cxt}}
$$

Typing $\Gamma \vdash t : A$.

$$
\text{TY-C}\ \frac{(c{:}A) \in \Sigma}{\Gamma \vdash c : A} \qquad
\text{TY-VAR}\ \frac{(x{:}A) \in \Gamma \qquad \Gamma\ \mathsf{cxt}}{\Gamma \vdash x : A} \qquad
\text{TY-ABS}\ \frac{\Gamma, x{:}A \vdash t : B}{\Gamma \vdash \lambda x t : A \to B}
$$

$$
\text{TY-APP}\ \frac{\Gamma \vdash r : A \to B \qquad \Gamma \vdash s : A}{\Gamma \vdash r\,s : B} \qquad
\text{TY-SUB}\ \frac{\Gamma \vdash t : A \qquad \Gamma \vdash A \le B : *}{\Gamma \vdash t : B}
$$

$$
\text{TY-GEN}\ \frac{\Gamma, X{:}\circ\kappa \vdash t : F\,X}{\Gamma \vdash t : \forall_\kappa F} \qquad
\text{TY-INST}\ \frac{\Gamma \vdash t : \forall_\kappa F \qquad \Gamma \vdash G : \kappa}{\Gamma \vdash t : F\,G}
$$

$$
\text{TY-REC}\ \frac{\Gamma \vdash A\ \mathsf{fix}_n^\nabla\text{-adm} \qquad \Gamma \vdash a : \mathsf{ord}}{\Gamma \vdash \mathsf{fix}_n^\nabla : (\forall \imath{:}\mathsf{ord}.\ A\,\imath \to A\,(\imath+1)) \to A\,a}
$$

Figure 6: $\mathsf{F}_{\widehat{\omega}}$: Terms, reduction and typing.