Semi-continuous Sized Types and Termination

Andreas Abel[⋆]

Institut für Informatik
Ludwig-Maximilians-Universität München
Oettingenstr. 67, D-80538 München, Germany
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 semantical criterion for admissibility. To provide a syntactical criterion, a calculus for semi-continuous function 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 partial evaluation. In languages with a notion of computation on the type-level, such as dependently-typed languages or rich typed intermediate languages in compilers [11], termination of expressions that compute a type is required for type checking and type soundness. 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 programs to ensure consistency. Since the pioneering work of Mendler [15], termination analysis has been combined with typing, with much success for strongly-typed languages [14,6,13,19,7,9]. The resulting technique, *type-based termination 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) In design and justification, type-based termination rests on a technology extensively studied for several decades: types.

[⋆] Research supported by the coordination action *TYPES* (510996) and thematic network *Applied Semantics II* (IST-2001-38957) of the European Union and the project *Cover* of the Swedish Foundation of Strategic Research (SSF).

(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 without change to the termination module. 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:

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

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

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

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

$$\begin{aligned}&\mathsf{eqGRose}\ \mathit{eqF}\ \mathit{eqA}\ \mathsf{leaf}\ \mathsf{leaf} = \mathsf{true}\\ &\mathsf{eqGRose}\ \mathit{eqF}\ \mathit{eqA}\ (\mathsf{node}\ a\ \mathit{fr})\ (\mathsf{node}\ a'\ \mathit{fr}') = (\mathit{eqA}\ a\ a') \wedge\\ &\qquad\qquad (\mathit{eqF}\ (\mathsf{eqGRose}\ \mathit{eqF}\ \mathit{eqA})\ \mathit{fr}\ \mathit{fr}')\\ &\mathsf{eqGRose}\ \mathit{eqF}\ \mathit{eqA}\ \_\ \_ = \mathsf{false}\end{aligned}$$

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 $F A$. The equality test for generalized rose trees, $\mathsf{eqGRose}\ \mathit{eqF}\ \mathit{eqA}$, 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:

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

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:

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

When defining $\mathsf{eqGRose}$ for trees of height $< \imath + 1$, we may use $\mathsf{eqGRose}$ on trees of height $< \imath$. Hence, in the clause for two nodes, term $\mathsf{eqGRose}$ *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):

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

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* [14]). 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 inductive theorem provers, are the W-type, Brouwer ordinals and the accessibility predicate [17].

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* 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 types $A(\alpha)$ that are *upper semi-continuous* in α, meaning $\limsup_{\alpha \to \lambda} \mathcal{A}(\alpha) \subseteq \mathcal{A}(\lambda)$ for limit ordinals λ, are admissible. Function types $C(\alpha) = A(\alpha) \to B(\alpha)$ will be admissible if A is *lower semi-continuous* ($\mathcal{A}(\lambda) \subseteq \liminf_{\alpha \to \lambda} \mathcal{A}(\alpha)$) 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 really necessary 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. 4.2). 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 subtype 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:

$$\begin{array}{ll}
A(\imath) := (\mathsf{Nat}^\omega \to \mathsf{Nat}^\imath) \to \mathsf{Nat}^\omega & \\
\mathsf{shift} : \quad \forall\imath.\, (\mathsf{Nat}^\omega \to \mathsf{Nat}^{\imath+1}) & \\
\qquad\qquad \to \mathsf{Nat}^\omega \to \mathsf{Nat}^\imath & \\
\mathsf{shift} := \lambda g \lambda n.\, \mathsf{pred}\,(g\,(\mathsf{succ}\,n)) &
\end{array}
\qquad
\begin{array}{lll}
f & : & \forall\imath.\, A(\imath) \to A(\imath+1) \\
f & := & \lambda loop \lambda g.\ loop\,(\mathsf{shift}\,g) \\
 & & \\
\mathsf{loop} & : & \forall\imath.\, A(\imath) \\
\mathsf{loop} & := & \mathsf{fix}\, f
\end{array}$$

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), that preserve normalization but limit the definition of recursive functions to special patterns, namely instantiations of the recursion scheme dictated by the recursion operator. Taming the general recursion operator fix through typing, however, which allows the definition of recursive functions in the intuitive way known from functional programming, is not yet fully explored. Mendler [15] 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 [6] and Giménez [13] developed Mendler's approach further, until a presentation using ordinal-indexed (co)inductive types was found and proven sound by Barthe et al. [7]. 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 [2], 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 [1]. Other recent works on type-based termination [9,10,8] stick to the restriction of $\lambda^{\widehat{}}$. Xi [19] 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 which excludes infinite objects we are interested in.

Closest to the present work is the sized type system of Hughes, Pareto, and Sabry [14], *Synchronous Haskell* [16], 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 semantical criteria for ω-undershooting types are already present, but in a rather 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 also allow ordinals up to the ωth uncountable 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 [4].

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

In this section we introduce $\mathsf{F}_\omega^{\widehat{}}$, 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. Function kinds are equipped with polarities p [18], which are written before the domain or on top of the arrow. Polarity $+$ denotes covariant constructors, $-$ contravariant constructors and $\circ$ mixed-variant constructors [12]. It is well-known that in order to obtain a normalizing language, any constructor underlying an inductive type must be covariant [15], hence, we restrict formation of least fixed-points $\mu_\kappa^a F$ to covariant Fs. (Abel [3] and Matthes [5] provide more explanation on polarities.)

The first argument, a, to μ, which we usually write as superscript, denotes the upper bound for the height of elements in the inductive type. The index a is a constructor of kind ord and denotes an ordinal; the canonical inhabitants of ord are given by the grammar

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

with $\imath$ an ordinal variable. 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 $\mu^a F \leq \mu^b F$ for $a \leq b$, meaning that μ is covariant in the index argument. Finally, $F \leq F'$ implies $\mu^a F \leq \mu^a F'$, so we get the kinding

$$\mu_\kappa : \mathsf{ord} \xrightarrow{+} (\kappa \xrightarrow{+} \kappa) \xrightarrow{+} \kappa$$

for the least fixed-point constructor. The kind κ is required to be *pure*, i. e., a kind not mentioning ord, for cardinality reasons. Only then it is possible to

Polarities, kinds, constructors, kinding contexts.

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

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

1	$: *$	unit type
$+$	$: * \xrightarrow{+} * \xrightarrow{+} *$	disjoint sum
$\times$	$: * \xrightarrow{+} * \xrightarrow{+} *$	cartesian product
$\to$	$: * \xrightarrow{-} * \xrightarrow{+} *$	function space
$\forall_\kappa$	$: (\kappa \xrightarrow{\circ} *) \xrightarrow{+} *$	quantification
μ_{κ_*}	$: \mathsf{ord} \xrightarrow{+} (\kappa_* \xrightarrow{+} \kappa_*) \xrightarrow{+} \kappa_*$	inductive constructors
ν_{κ_*}	$: \mathsf{ord} \xrightarrow{-} (\kappa_* \xrightarrow{+} \kappa_*) \xrightarrow{+} \kappa_*$	coinductive constructors
s	$: \mathsf{ord} \xrightarrow{+} \mathsf{ord}$	successor of ordinal
∞	$: \mathsf{ord}$	infinity ordinal

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

r, s, t	$::= c \mid x \mid \lambda x t \mid r\, s$	term
c	$::= () \mid \mathsf{pair} \mid \mathsf{fst} \mid \mathsf{snd} \mid \mathsf{inl} \mid \mathsf{inr} \mid \mathsf{case} \mid \mathsf{in} \mid \mathsf{out} \mid \mathsf{fix}^\mu_n \mid \mathsf{fix}^\nu_n$	constant ($n \in \mathbb{N}$)
v	$::= \lambda x t \mid \mathsf{pair}\, t_1\, t_2 \mid \mathsf{inl}\, t \mid \mathsf{inr}\, t \mid \mathsf{in}\, t \mid c \mid \mathsf{pair}\, t \mid \mathsf{fix}^\nabla_n s\, t_{1..m}$	value ($m \leq n$)
$e(\_)$	$::= \_\; s \mid \mathsf{fst}\; \_ \mid \mathsf{snd}\; \_ \mid \mathsf{case}\; \_ \mid \mathsf{out}\; \_ \mid \mathsf{fix}^\mu_n\; s\; t_{1..n}\; \_$	evaluation frame
E	$::= \mathsf{Id} \mid E \circ e$	evaluation context
Γ	$::= \diamond \mid \Gamma, x\!:\!A \mid \Gamma, X\!:\!p\kappa$	typing context

Reduction $t \longrightarrow t'$.

$(\lambda x t)\, s$	$\longrightarrow [s/x]t$	$\mathsf{out}\,(\mathsf{in}\, r)$	$\longrightarrow r$
$\mathsf{fst}\,(r, s)$	$\longrightarrow r$	$\mathsf{fix}^\mu_n\; s\; t_{1..n}\,(\mathsf{in}\, t)$	$\longrightarrow s\,(\mathsf{fix}^\mu_n\; s)\, t_{1..n}\,(\mathsf{in}\, t)$
$\mathsf{snd}\,(r, s)$	$\longrightarrow s$	$\mathsf{out}\,(\mathsf{fix}^\nu_n\; s\, t_{1..n})$	$\longrightarrow \mathsf{out}\,(s\,(\mathsf{fix}^\nu_n\; s)\, t_{1..n})$
$\mathsf{case}\,(\mathsf{inl}\, r)$	$\longrightarrow \lambda x \lambda y.\, x\, r$		
$\mathsf{case}\,(\mathsf{inr}\, r)$	$\longrightarrow \lambda x \lambda y.\, y\, r$	+ closure under all term constructs	

Fig. 1. $\mathsf{F}^{\widehat{}}_\omega$: Syntax and operational semantics

estimate a single *closure ordinal* ∞ at which the fixed-point is reached for *all* inductive types. 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. If ord was allowed in the kind of a fixed-point, the closure ordinal of this 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

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

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

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

Example 1 (Some sized types).

$$\begin{array}{llll}
\mathsf{Nat} & : \;\; \mathsf{ord} \xrightarrow{+} * & \mathsf{GRose} : & \mathsf{ord} \xrightarrow{+} (* \xrightarrow{+} *) \xrightarrow{+} * \xrightarrow{+} * \\
\mathsf{Nat} & := \lambda\imath.\; \mu^{\imath}\lambda X.\, 1 + X & \mathsf{GRose} := & \lambda\imath\lambda F\lambda A.\; \mu^{\imath}\lambda X.\, 1 + A \times F\, X \\
\mathsf{List} & : \;\; \mathsf{ord} \xrightarrow{+} * \xrightarrow{+} * & \mathsf{Tree} \;\;\, : & \mathsf{ord} \xrightarrow{+} * \xrightarrow{-} * \xrightarrow{+} * \\
\mathsf{List} & := \lambda\imath\lambda A.\; \mu^{\imath}\lambda X.\, 1 + A \times X & \mathsf{Tree} \;\;\, := & \lambda\imath\lambda B\lambda A.\, \mathsf{GRose}^{\imath}\, (\lambda X.\, B \to X)\, A \\
\mathsf{Stream} & : \;\; \mathsf{ord} \xrightarrow{-} * \xrightarrow{+} * & & \\
\mathsf{Stream} & := \lambda\imath\lambda A.\, \nu^{\imath}\lambda X.\, A \times X & &
\end{array}$$

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 ($\times$) types. Further, there is folding, in, and unfolding, out, of (co)inductive types. Let $\kappa = \boldsymbol{p\kappa} \to *$ a pure kind, $F : +\kappa \to \kappa$, $G_i : \kappa_i$ for $1 \leq i \leq |\boldsymbol{\kappa}|$, $a : \mathsf{ord}$, and $\nabla \in \{\mu, \nu\}$, then we have the following (un)folding rules:

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

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

One-step reduction $t \longrightarrow t'$ is defined by the β-reduction axioms given in Figure 1 plus congruence rules. Interesting are the reduction rules for recursion and corecursion:

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

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 [15], Giménez [13], Barthe et al. [7], and the author [2]. Dually, corecursive functions are only unfolded on demand, i. e., in an evaluation context, the matching one being $\mathsf{out}\ \_$.

As pointed out in the introduction, recursion is introduced by the rule

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

Herein, ∇ stands for μ or ν, and the judgement $A\ \mathsf{fix}^{\nabla}_{n}\text{-}\mathsf{adm}$ makes sure type A is admissible for (co)recursion, as discussed in the introduction. In the following, we will find out which types are admissible.

3 Semantics

In this section, we provide an interpretation of types as saturated sets of strongly normalizing terms. Let $\mathcal{S}$ denote the set of strongly normalizing terms. We define *safe* (weak head) reduction by these axioms:

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

Additionally, we close safe reduction under evaluation contexts and transitivity:

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

The relation is defined such that $\mathcal{S}$ is closed under $\rhd$-expansion, meaning $t \rhd t' \in \mathcal{S}$ implies $t \in \mathcal{S}$. 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

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

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 terms without creating a new redex. A term set $\mathcal{A}$ is *saturated* if $\mathcal{A}$ is closed and $\mathcal{N} \subseteq \mathcal{A} \subseteq \mathcal{S}$.

Interpretation of kinds. 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. Function kinds $[\![\circ\kappa \to \kappa']\!] := [\![\kappa]\!] \to [\![\kappa']\!]$ are interpreted as set-theoretic function spaces; a covariant function kind denotes just the monotonic functions and a contravariant kind the antitonic ones. 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 $\sup^{p\kappa \to \kappa'}$.

Limits and iteration. 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:

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

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:

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

For monotone f, we obtain the usual approximants of least and greatest fixed-points as $\boldsymbol{\mu}^\alpha f = f^\alpha(\bot)$ and $\boldsymbol{\nu}^\alpha f = f^\alpha(\top)$.

Closure ordinal. 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, $|[\![*]\!]| \le |\mathcal{P}(\mathbb{N})| = \beth_1$, and by induction on κ, $|[\![\kappa]\!]| \le \beth_{|\kappa|+1}$. 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 $\top^{\mathsf{ord}} = \beth_\omega$.

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 $e^{-1}\mathcal{A} = \{r \mid e(r) \in \mathcal{A}\}$. For saturated sets $\mathcal{A}, \mathcal{B} \in [\![*]\!]$ we define the following saturated sets:

$$\begin{aligned} \mathcal{A} \boxplus \mathcal{B} &:= \overline{\mathsf{inl} \cdot \mathcal{A}} \cup \overline{\mathsf{inr} \cdot \mathcal{B}} & \boxed{1} &:= \overline{\{()\}} \\ \mathcal{A} \boxtimes \mathcal{B} &:= (\mathsf{fst}\ \_)^{-1}\mathcal{A} \cap (\mathsf{snd}\ \_)^{-1}\mathcal{B} & \mathcal{A}^\mu &:= \overline{\mathsf{in} \cdot \mathcal{A}} \\ \mathcal{A} \boxed{\to} \mathcal{B} &:= \textstyle\bigcap_{s\in\mathcal{A}} (\_\ s)^{-1}\mathcal{B} & \mathcal{A}^\nu &:= (\mathsf{out}\ \_)^{-1}\mathcal{A} \end{aligned}$$

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 $\nabla \in \{\mu, \nu\}$.

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

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

We extend this semantics to constructors F in the usual way, such that if $\Delta \vdash F : \kappa$ and $\theta(X) \in [\![\kappa']\!]$ for all $(X : p\kappa') \in \Delta$, then $[\![F]\!]_\theta \in [\![\kappa]\!]$.

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

Semantic admissibility and strong normalization. For the main theorem to follow, we assume semantical soundness of our yet to be defined syntactical criterion of 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: $\mathcal{A}(\alpha) = \bigcap_{k \in K} \mathcal{B}_1(k, \alpha) \boxed{\to} \dots \boxed{\to} \mathcal{B}_n(k, \alpha) \boxed{\to} \mathcal{B}(k, \alpha)$ for some K and some $\mathcal{B}_1, \dots, \mathcal{B}_n, \mathcal{B} \in K \times [\![\mathsf{ord}]\!] \to [\![*]\!]$. 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. Semi-continuity: $\limsup_{\alpha \to \lambda} \mathcal{A}(\alpha) \subseteq \mathcal{A}(\lambda)$ for all limit ordinals $\lambda \in [\![\mathsf{ord}]\!] \setminus \{0\}$.

Let $t\theta$ denote the simultaneous substitution of $\theta(x)$ for each $x \in \mathsf{FV}(t)$ in t.

Theorem 1 (Type soundness). *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 $t\theta \in [\![B]\!]\theta$.*

Corollary 1 (Strong normalization). *If $\Gamma \vdash t : B$ then t is strongly normalizing.*

4 Semi-continuity

As motivated in the introduction, only types $\mathcal{C} \in [\![\mathsf{ord}]\!] \to [\![*]\!]$ with $\inf_\lambda \mathcal{C} \sqsubseteq \mathcal{C}(\lambda)$ can be admissible for recursion. Under which conditions on $\mathcal{A}$ and $\mathcal{B}$ can a function type $\mathcal{A}(\alpha) \boxed{\to} \mathcal{B}(\alpha)$ be admissible? It shows that the first choice $\inf_\lambda \mathcal{B} \sqsubseteq \mathcal{B}(\lambda)$ is a requirement too strong: To show $\inf_{\alpha<\lambda}(\mathcal{A}(\alpha) \boxed{\to} \mathcal{B}(\alpha)) \sqsubseteq \mathcal{A}(\lambda) \boxed{\to} \mathcal{B}(\lambda)$ we would need $\mathcal{A}(\lambda) \sqsubseteq \inf_\lambda \mathcal{A}$, which is not even true for $\mathcal{A} = \mathcal{N}at$ at limit ω. However, each type $\mathcal{C}$ with $\limsup_\lambda \mathcal{C} \sqsubseteq \mathcal{C}(\lambda)$ also fulfills $\inf_\lambda \mathcal{C} \sqsubseteq \mathcal{C}(\lambda)$, and the modified condition distributes better over function spaces.

Lemma 1. *If $\mathcal{A}(\lambda) \sqsubseteq \liminf_\lambda \mathcal{A}$ and $\limsup_\lambda \mathcal{B} \sqsubseteq \mathcal{B}(\lambda)$ then $\limsup_\lambda(\mathcal{A}(\alpha) \boxed{\to} \mathcal{B}(\alpha)) \sqsubseteq \mathcal{A}(\lambda) \boxed{\to} \mathcal{B}(\lambda)$.*

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 $f(\lambda)$).

4.1 Positive Results

Basic semi-continuous types. 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 to show upper semi-continuity of function types such as $\mathsf{Eq}(\mathsf{GRose}^\imath\, F A)$ (see introduction) and, e. g.,

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

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 co-domain $\mathcal{C}'(\alpha)$ is upper semi-continuous, hence, Lemma 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

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

However, domain types where one indexed inductive type is inside another inductive type are useful in practice, see Example 3. 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 $f(\lambda) = \inf_\lambda f$. An antitone f guarantees $\inf_\lambda f = \limsup_\lambda f$, so f is upper semi-continuous. This establishes upper semi-continuity of a type involved in stream-zipping,

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

The domain types are antitonic, hence lower semi-continuous, and the coinductive codomain is upper semi-continuous. Upper semi-continuity of $\mathcal{S}tream^\alpha(\mathcal{N}at^\alpha)$ and similar types is not yet covered, but now we will develop concepts that allow us to look inside (co)inductive types.

Semi-continuity and (co)induction. 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 2 (Facts about limits).

1. $\limsup_{\alpha\to\lambda} f(\alpha,\alpha) \sqsubseteq \limsup_{\beta\to\lambda} \limsup_{\gamma\to\lambda} f(\beta,\gamma)$.
2. $\liminf_{\beta\to\lambda} \liminf_{\gamma\to\lambda} f(\beta,\gamma) \sqsubseteq \liminf_{\alpha\to\lambda} f(\alpha,\alpha)$.
3. $\limsup_{\alpha\to\lambda} \inf_{i\in I} f(\alpha,i) \sqsubseteq \inf_{i\in I} \limsup_{\alpha\to\lambda} f(\alpha,i)$.
4. $\sup_{i\in I} \liminf_{\alpha\to\lambda} f(\alpha,i) \sqsubseteq \liminf_{\alpha\to\lambda} \sup_{i\in I} f(\alpha,i)$.

Strictly positive contexts: $\Pi ::= \diamond \mid \Pi, X\!:\!+\kappa_*$.

Semi-continuity $\Delta; \Pi \vdash^{\imath q} F : \kappa$ for $q \in \{\oplus, \ominus\}$.

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

$$\textsc{cont-in}\ \frac{\Delta \vdash F : \kappa}{\Delta, \imath\!:\!p\mathsf{ord}; \Pi \vdash^{\imath q} F : \kappa} \qquad \textsc{cont-var}\ \frac{X\!:\!p\kappa \in \Delta, \Pi \qquad p \in \{+, \circ\}}{\Delta; \Pi \vdash^{\imath q} X : \kappa}$$

$$\textsc{cont-}\forall\ \frac{\Delta; \Pi \vdash^{\imath\oplus} F : \circ\kappa \to *}{\Delta; \Pi \vdash^{\imath\oplus} \forall_\kappa F : *} \qquad \textsc{cont-abs}\ \frac{\Delta, X\!:\!p\kappa; \Pi \vdash^{\imath q} F : \kappa'}{\Delta; \Pi \vdash^{\imath q} \lambda X F : p\kappa \to \kappa'}\ X \neq \imath$$

$$\textsc{cont-app}\ \frac{\Delta, \imath\!:\!p'\mathsf{ord}; \Pi \vdash^{\imath q} F : p\kappa \to \kappa' \qquad p^{-1}\Delta \vdash G : \kappa}{\Delta, \imath\!:\!p'\mathsf{ord}; \Pi \vdash^{\imath q} F\, G : \kappa'}$$

$$\textsc{cont-sum}\ \frac{\Delta; \Pi \vdash^{\imath q} A, B : *}{\Delta; \Pi \vdash^{\imath q} A + B : *} \qquad \textsc{cont-prod}\ \frac{\Delta; \Pi \vdash^{\imath q} A, B : *}{\Delta; \Pi \vdash^{\imath q} A \times B : *}$$

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

$$\textsc{cont-mu}\ \frac{\Delta; \Pi, X\!:\!+\kappa_* \vdash^{\imath\ominus} F : \kappa_* \qquad \Delta \vdash^{\imath\ominus} a : \mathsf{ord}}{\Delta; \Pi \vdash^{\imath\ominus} \mu^a_{\kappa_*} \lambda X F : \kappa_*}$$

$$\textsc{cont-nu}\ \frac{\Delta; \Pi, X\!:\!+\kappa_* \vdash^{\imath\oplus} F : \kappa_* \qquad a \in \{\infty, \mathsf{s}^n \jmath \mid (\jmath\!:\!p\mathsf{ord}) \in \Delta \text{ with } p \in \{+, \circ\}\}}{\Delta; \Pi \vdash^{\imath\oplus} \nu^a_{\kappa_*} \lambda X F : \kappa_*}$$

Fig. 2. $\mathsf{F}_{\widehat{\omega}}$: Semi-continuous constructors

Fact 3 states that lim sup pushes through infimum and, thus, justifies rule CONT-$\forall$ in Fig. 2 (see Sect. 5). The dual fact 4 expresses that lim inf can be pulled out of a supremum.

Lemma 3. *Binary sums* $\boxed{+}$ *and products* $\boxed{\times}$ *and the operations* $(-)^\mu$ *and* $(-)^\nu$ *are* lim sup*-pushable and* lim inf*-pullable.*

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

A generalization of Lemma 1 is:

Lemma 4 **(**lim sup **through function space).**
$\limsup_{\alpha\to\lambda} (\mathcal{A}(\alpha) \boxed{\to} \mathcal{B}(\alpha)) \sqsubseteq (\liminf_\lambda \mathcal{A}) \boxed{\to} \limsup_\lambda \mathcal{B}$.

Now, to (co)inductive types. Let $\phi \in \mathsf{O} \to \mathsf{O}$.

Lemma 5. $\boldsymbol{\mu}^{\liminf_\lambda \phi} = \liminf_{\alpha\to\lambda} \boldsymbol{\mu}^{\phi(\alpha)}$ *and* $\limsup_{\alpha\to\lambda} \boldsymbol{\nu}^{\phi(\alpha)} = \boldsymbol{\nu}^{\liminf_\lambda \phi}$.

Lemma 6. *For $\alpha \in \mathsf{O}$, let $\mathcal{F}_\alpha \in \mathfrak{L} \xrightarrow{+} \mathfrak{L}$ be* $\liminf$*-pullable and $\mathcal{G}_\alpha \in \mathfrak{L} \xrightarrow{+} \mathfrak{L}$ be* $\limsup$*-pushable. Then for all $\beta \in \mathsf{O}$, $\boldsymbol{\mu}^\beta(\liminf_\lambda \mathcal{F}) \sqsubseteq \liminf_{\alpha\to\lambda} \boldsymbol{\mu}^\beta \mathcal{F}_\alpha$ and $\limsup_{\alpha\to\lambda} \boldsymbol{\nu}^\beta \mathcal{G}_\alpha \sqsubseteq \boldsymbol{\nu}^\beta(\limsup_\lambda \mathcal{G})$.*

Proof. By transfinite induction on β.

Corollary 2 (Limits and (co)inductive types).

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

Proof. For instance, the second inclusion can be derived in three steps using Lemma 2.1, Lemma 5, and Lemma 6.

Now, since $\mathcal{G}_\alpha(\mathcal{X}) = (\mathcal{N}at^\alpha \boxed{\times} \mathcal{X})^\nu$ is $\limsup$-pushable, we have can infer upper semi-continuity of $\mathcal{S}tream^\alpha(\mathcal{N}at^\alpha) = \boldsymbol{\nu}^\alpha \mathcal{G}_\alpha$. Analogously, we establish lower semi-continuity of $\mathcal{L}ist^\alpha(\mathcal{N}at^\alpha)$.

4.2 Negative Results

Function space and lower semi-continuity. One may wonder whether Lemma 1 can be dualized, i. e., does upper semi-continuity of $\mathcal{A}$ and lower semi-continuity of $\mathcal{B}$ entail lower semi-continuity of $\mathcal{C}(\alpha) = \mathcal{A}(\alpha) \boxed{\Rightarrow} \mathcal{B}(\alpha)$? The answer is no, e. g., consider $\mathcal{C}(\alpha) = \mathcal{N}at^\omega \boxed{\Rightarrow} \mathcal{N}at^\alpha$. 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 Lemma 1 the type $(\mathcal{N}at^\omega \boxed{\Rightarrow} \mathcal{N}at^\alpha) \boxed{\Rightarrow} \mathcal{N}at^\omega$, which admits a looping function (see introduction), would be upper semi-continuous.

Inductive types and upper semi-continuity. Pareto [16] proves that inductive types are (in our terminology) $\limsup$-pushable. His inductive types denote only finitely branching trees, but we also consider infinite branching, arising from function space embedded in inductive types. In my thesis [4, Sect. 5.4.3] I show that infinitely branching inductive data types do not inherit upper semi-continuity from their defining body. But remember that inductive types can still be upper semi-continuous if they are covariant in their size index.

5 A Kinding System for Semi-continuity

We turn the results of the last section into a calculus and define a judgement $\Delta; \Pi \vdash^{\imath q} F : \kappa$, where $\imath$ is an ordinal variable $(\imath : \mathsf{ord}) \in \Delta$, 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

$X\!:\!+\kappa'$. The complete listing of rules can be found in Figure 2; in the following, we discuss a few.

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

If $\imath$ appears positively in F, then F is trivially upper semi-continuous. In the conclusion we may choose to set $p = \circ$, meaning that we forget that F is monotone in $\imath$.

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

This rule incarnates Lemma 1. 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.

$$\textsc{cont-nu}\ \frac{\Delta; \Pi, X\!:\!+\kappa_* \vdash^{\imath\oplus} F : \kappa_*}{\Delta; \Pi \vdash^{\imath\oplus} \nu^a \lambda X\!:\!\kappa_*.\, F : \kappa_*}$$

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

Theorem 2 (Soundness of Continuity Derivations). *Let θ a valuation of the variables in Δ and Π, $(X\!:\!+\kappa') \in \Pi$, $\mathcal{G} \in [\![\mathsf{ord}]\!] \to [\![\kappa']\!]$, and $\lambda \in [\![\mathsf{ord}]\!]$ a limit ordinal.*

1. *If $\Delta; \Pi \vdash^{\imath\ominus} F : \kappa$ then*
   (a) $[\![F]\!]_{\theta[\imath\mapsto\lambda]} \sqsubseteq \liminf_{\alpha\to\lambda} [\![F]\!]_{\theta[\imath\mapsto\alpha]}$, *and*
   (b) $[\![F]\!]_{\theta[X\mapsto \liminf_\lambda \mathcal{G}]} \sqsubseteq \liminf_{\alpha\to\lambda} [\![F]\!]_{\theta[X\mapsto\mathcal{G}(\alpha)]}$.
2. *If $\Delta; \Pi \vdash^{\imath\oplus} F : \kappa$ then*
   (a) $\limsup_{\alpha\to\lambda} [\![F]\!]_{\theta[\imath\mapsto\alpha]} \sqsubseteq [\![F]\!]_{\theta[\imath\mapsto\lambda]}$, *and*
   (b) $\limsup_{\alpha\to\lambda} [\![F]\!]_{\theta[X\mapsto\mathcal{G}(\alpha)]} \sqsubseteq [\![F]\!]_{\theta[X\mapsto\limsup_\lambda \mathcal{G}]}$

Proof. By induction on the derivation [4, Sect. 5.5]. The soundness of cont-nu hinges on the fact that strictly positive coinductive types close at ordinal ω.

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

$$\begin{array}{l} \Gamma \vdash (\lambda\imath.\, \forall \boldsymbol{X}\!:\!\boldsymbol{\kappa}.B_1 \to \cdots \to B_n \to \mu^\imath F \boldsymbol{H} \to C)\ \ \mathsf{fix}^\mu_n\text{-}\mathsf{adm} \\ \text{iff}\quad \Gamma, \imath\!:\!\circ\mathsf{ord}, \boldsymbol{X}\!:\!\boldsymbol{\kappa}; \diamond \vdash^{\imath\oplus} B_{1..n} \to \mu^\imath F \boldsymbol{H} \to C : * \end{array}$$

$$\begin{array}{l} \Gamma \vdash (\lambda\imath.\, \forall \boldsymbol{X}\!:\!\boldsymbol{\kappa}.B_1 \to \cdots \to B_n \to \nu^\imath F \boldsymbol{H})\ \ \mathsf{fix}^\nu_n\text{-}\mathsf{adm} \\ \text{iff}\quad \Gamma, \imath\!:\!\circ\mathsf{ord}, \boldsymbol{X}\!:\!\boldsymbol{\kappa}; \diamond \vdash^{\imath\oplus} B_{1..n} \to \nu^\imath F \boldsymbol{H} : * \end{array}$$

It is easy to check that admissible types fulfill the semantic criteria given at the end of Section 3.

Example 2 (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:

$$\begin{array}{lll} \mathsf{mapStream} : & \forall A \forall B.\,(A \to B) \to \forall \imath.\, \mathsf{Stream}^{\imath} A \to \mathsf{Stream}^{\imath} B \\ \mathsf{nats} & :\ \forall \imath.\, \mathsf{Stream}^{\imath}\, \mathsf{Nat}^{\imath} \\ \mathsf{nats} & := \mathsf{fix}^{\nu}_{0} \lambda nats.\, \langle \mathsf{zero},\ \mathsf{mapStream}\ \mathsf{succ}\ nats \rangle \end{array}$$

Example 3 (Inductive type inside inductive type). In the following, we describe breadth-first traversal of rose (finitely branching) trees whose termination is recognized by $\mathsf{F}^{\frown}_{\omega}$.

$$\begin{array}{l} \mathsf{Rose} :\quad \mathsf{ord} \xrightarrow{+} * \xrightarrow{+} * \\ \mathsf{Rose} := \lambda \imath \lambda A.\, \mathsf{GRose}^{\imath}\, \mathsf{List}^{\infty}\, A = \lambda \imath \lambda A.\ \mu^{\imath}_{*} \lambda X.\, A \times \mathsf{List}^{\infty} X \end{array}$$

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 $< \imath$).

$$\begin{array}{l} \mathsf{step} :\quad \forall \jmath \forall A \forall \imath.\, \mathsf{List}^{\jmath}(\mathsf{Rose}^{\imath+1}\, A) \to \mathsf{List}^{\jmath}\, A \times \mathsf{List}^{\infty}(\mathsf{Rose}^{\imath}\, A) \\ \mathsf{step} := \mathsf{fix}^{\mu}_{0} \lambda step \lambda l.\, \mathsf{match}\ l\ \mathsf{with} \\ \qquad\qquad \mathsf{nil} \mapsto \langle \mathsf{nil},\, \mathsf{nil} \rangle \\ \qquad\qquad \mathsf{cons}\, \langle a, rs' \rangle\, rs \mapsto \mathsf{match}\ step\ rs\ \mathsf{with} \\ \qquad\qquad\qquad\qquad \langle as,\ rs'' \rangle \mapsto \langle \mathsf{cons}\ a\ as,\ \mathsf{append}\, rs'\, rs'' \rangle \end{array}$$

Now, bf iterates step on a non-empty forest. It is defined by induction on $\imath$.

$$\begin{array}{l} \mathsf{bf} :\quad \forall \imath \forall A.\, \mathsf{Rose}^{\imath}\, A \to \mathsf{List}^{\infty}(\mathsf{Rose}^{\imath}\, A) \to \mathsf{List}^{\infty} A \\ \mathsf{bf} := \mathsf{fix}^{\mu}_{0} \lambda bf \lambda r \lambda rs.\ \mathsf{match}\ \mathsf{step}\, (\mathsf{cons}\ r\ rs)\ \mathsf{with} \\ \qquad\qquad \langle as,\ \mathsf{nil} \rangle \qquad\qquad \mapsto as \\ \qquad\qquad \langle as,\ \mathsf{cons}\ r'\ rs' \rangle \mapsto \mathsf{append}\ as\ (bf\ r'\ rs') \end{array}$$

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 Cor. 2 and rule CONT-MU.

6 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 semantical 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 [4, Ch. 6]. These examples cannot be handled by type-based termination à la Barthe et al. [7,8], but our developments could be directly incorporated into their calculus.

In previous work [1], 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 [14] 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 more fine-grained polarity system will be necessary.

Acknowledgments. I would like to thank my supervisor, Martin Hofmann, for discussions on $\mathsf{F}_{\omega}^{\frown}$. Thanks to John Hughes for lending his ear in difficult phases of this work, for instance, when I was trying to prove upper semi-continuity of inductive types but then found a counterexample. Thanks to the anonymous referees of previous versions of this paper who gave insightful and helpful comments.

References

1. Abel, A.: Termination and guardedness checking with continuous types. In: Hofmann, M., ed., Typed Lambda Calculi and Applications (TLCA 2003), Valencia, Spain, volume 2701 of Lecture Notes in Computer Science. Springer-Verlag (2003), 1–15
2. Abel, A.: Termination checking with types. RAIRO – Theoretical Informatics and Applications **38** (2004) 277–319. Special Issue: Fixed Points in Computer Science (FICS'03)
3. Abel, A.: Polarized subtyping for sized types. In: Grigoriev, D., Harrison, J., Hirsch, E. A., eds., Computer Science Symposium in Russia (CSR 2006), St. Petersburg, June 8-12, 2006, volume 3967 of Lecture Notes in Computer Science. Springer-Verlag (2006), 381–392
4. Abel, A.: A Polymorphic Lambda-Calculus with Sized Higher-Order Types. Ph.D. thesis, Ludwig-Maximilians-Universität München (2006)
5. Abel, A., Matthes, R.: Fixed points of type constructors and primitive recursion. In: Marcinkowski, J., Tarlecki, A., eds., Computer Science Logic, CSL'04, volume 3210 of Lecture Notes in Computer Science. Springer-Verlag (2004), 190–204
6. Amadio, R. M., Coupet-Grimal, S.: Analysis of a guard condition in type theory. In: Nivat, M., ed., Foundations of Software Science and Computation Structures, First International Conference, FoSSaCS'98, volume 1378 of Lecture Notes in Computer Science. Springer-Verlag (1998), 48–62
7. Barthe, G., Frade, M. J., Giménez, E., Pinto, L., Uustalu, T.: Type-based termination of recursive definitions. Mathematical Structures in Computer Science **14** (2004) 1–45

8. Barthe, G., Grégoire, B., Pastawski, F.: Practical inference for type-based termination in a polymorphic setting. In: Urzyczyn, P., ed., Typed Lambda Calculi and Applications (TLCA 2005), Nara, Japan, volume 3461 of Lecture Notes in Computer Science. Springer-Verlag (2005), 71–85
9. Blanqui, F.: A type-based termination criterion for dependently-typed higher-order rewrite systems. In: van Oostrom, V., ed., Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3 – 5, 2004, Proceedings, volume 3091 of Lecture Notes in Computer Science. Springer-Verlag (2004), 24–39
10. Blanqui, F.: Decidability of type-checking in the Calculus of Algebraic Constructions with size annotations. In: Ong, C.-H. L., ed., 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. Springer-Verlag (2005), 135–150
11. Crary, K., Weirich, S.: Flexible type analysis. In: Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, France, volume 34 of SIGPLAN Notices. ACM Press (1999), 233–248
12. Duggan, D., Compagnoni, A.: Subtyping for object type constructors (1999). Presented at FOOL 6
13. Giménez, E.: Structural recursive definitions in type theory. In: Larsen, K. G., Skyum, S., Winskel, G., eds., Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13-17, 1998, Proceedings, volume 1443 of Lecture Notes in Computer Science. Springer-Verlag (1998), 397–408
14. Hughes, J., Pareto, L., Sabry, A.: Proving the correctness of reactive systems using sized types. In: 23rd Symposium on Principles of Programming Languages, POPL'96 (1996), 410–423
15. Mendler, N. P.: 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. IEEE Computer Society Press (1987), 30–36
16. Pareto, L.: Types for Crash Prevention. Ph.D. thesis, Chalmers University of Technology (2000)
17. Paulin-Mohring, C.: Inductive definitions in the system Coq—rules and properties. Technical report, Laboratoire de l'Informatique du Parallélisme (1992)
18. Steffen, M.: Polarized Higher-Order Subtyping. Ph.D. thesis, Technische Fakultät, Universität Erlangen (1998)
19. Xi, H.: Dependent types for program termination verification. In: Proceedings of 16th IEEE Symposium on Logic in Computer Science. Boston, USA (2001)