Wellfounded Recursion with Copatterns

A Unified Approach to Termination and Productivity

Andreas Abel

Department of Computer Science,
Ludwig-Maximilians-University Munich, Germany
andreas.abel@ifi.lmu.de

Brigitte Pientka

School of Computer Science,
McGill University, Montreal, Canada
bpientka@cs.mcgill.ca

Abstract

In this paper, we study strong normalization of a core language based on System F_ω which supports programming with finite and infinite structures. Building on our prior work, finite data such as finite lists and trees are defined via constructors and manipulated via pattern matching, while infinite data such as streams and infinite trees is defined by observations and synthesized via copattern matching. In this work, we take a type-based approach to strong normalization by tracking size information about finite and infinite data in the type. This guarantees compositionality. More importantly, the duality of pattern and copatterns provide a unifying semantic concept which allows us for the first time to elegantly and uniformly support both well-founded induction and coinduction by mere rewriting. The strong normalization proof is structured around Girard's reducibility candidates. As such our system allows for non-determinism and does not rely on coverage. Since System F_ω is general enough that it can be the target of compilation for the Calculus of Constructions, this work is a significant step towards representing observation-centric infinite data in proof assistants such as Coq and Agda.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Data types and structures, Patterns, Recursion; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Program and recursion schemes, Type structure; F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—Lambda calculus

General Terms Languages, Theory

Keywords Recursion, Coinduction, Pattern matching, Productivity, Strong normalization, Type-based termination

1. Introduction

Integrating infinite data and coinduction with dependent types is tricky. For example, in the Calculus of (Co)Inductive Constructions, the core theory underlying Coq (INRIA 2012), coinduction is broken, since computation does not preserve types (Giménez 1996). In Agda (Norell 2007), a dependently typed proof and programming environment based on Martin-Löf Type Theory, inductive and coinductive types cannot be mixed in a compositional way.[1] In previous work (Abel et al. 2013) we have introduced *copatterns* as a novel perspective on defining infinite structures that might serve as a new foundation for coinduction in dependently-typed languages, overcoming the problems in the present solutions.

In the copattern approach, finite data such as finite lists and trees are defined as usual via constructors and manipulated via pattern matching, while infinite data such as streams and infinite trees are defined by observations and synthesized via copattern matching. For example, instead of conceiving streams as built by the constructor `cons`, we consider the observations `head` and `tail` about streams as primitive. Programs about streams are defined in terms of the observations `head` and `tail`.

Our previous work left the question of termination of recursive function and the productivity of infinite objects open. Both issues are crucial since we want to program inductive proofs as recursive functions and coinductive proofs as infinite objects or corecursive functions producing infinite objects. In this article, we adapt type-based termination (Hughes et al. 1996; Amadio and Coupet-Grimal 1998; Barthe et al. 2004; Blanqui 2004; Abel 2008a; Sacchini 2013) to definitions by copatterns.

A syntactic termination check would ensure that recursive calls occur only with arguments smaller than the ones of the original call. In type-based termination, inductive types are tagged with a size expression that denotes the (ordinal) maximal height of the trees inhabiting it, i. e., an upper bound on the number of constructors in the longest path of the tree. To prove termination of a recursive function means to show that it can safely handle arguments of arbitrary size. This can be established by well-founded induction: to show that a function can handle arguments up to a fixed size a, we may assume it already safely processes arguments of any smaller size $b < a$. This induction principle can be turned into a typing rule for recursive functions, using sized types and size quantification. How can this be dualized to coinduction? A stream is *productive* if we can make arbitrarily deep observations, i. e., if we can take its tail arbitrarily many times. To show that a stream definition is productive, we also proceed by well-founded induction. To show that it can safely handle a observations, we may assume that b observations are fine for any $b < a$. The number of observations we can safely make is called the *depth* of the stream, or more general, of the coinductive structure. One should not be mislead and think of the depth as "size"; streams do not have a size since they are not tree-structures in memory—they only exist as processes that con-

[1] In Agda, one can encode the property "infinitely often" from temporal logic, but not its dual "eventually forever" (Altenkirch and Danielsson 2010).

ICFP'13, September, 2013, Boston, USA.

tinuously yield elements on demand. But it is fruitful to transfer the concept of *depth* to (co)recursive functions. The depth of a function is the maximal size of arguments it can safely handle. As we are only interested in streams of infinite depth in the end, we care only about functions of infinite depth. Yet to establish productivity and termination, we need to induct on depth.

The type-based termination approach is in contrast to common approaches taken in systems such as Coq (INRIA 2012) and Agda (Norell 2007) which employ a syntactic guardedness check to ensure corecursive programs are productive: all corecursive calls must occur under a constructor. This ensures that the next unit of information can be computed in a finite amount of time (Sijtsma 1989). However, this approach has also known limitations: it is difficult to handle higher-order programs such as $g\,f = \mathsf{cons}\,0\,(f\,(g\,f))$ where the productivity of g depends on the behavior of the function f. It is also not compositional, i. e., we cannot easily abstract over a constructor cons in a productive program and replace it with a function f. Both limitations are due to the lack of information we have about f in the syntactic guardedness check. Types on the other hand already track information about each argument to a definition and its output. Type-based termination piggy-backs on the typing analysis and avoids a separate formal system to traverse the definitions. By indexing types with sizes, we are able to carry more precise information about input and output arguments and their relation which is then verified simultaneously while type checking the definitions.

The contributions of our work are:

- We present $\mathsf{F}_\omega^{\mathsf{cop}}$, an extension of System F_ω by inductive and coinductive types, sizes and bounded size quantification, pattern and copattern matching and lexicographic termination measures.
- In contrast to previous approaches on type-based termination, we use well-founded induction on ordinals instead of conventional induction that distinguishes between zero, successor and limit ordinals. Disposing of this case distinction, we operate within constructive foundations of mathematics (Taylor 1996).
- Well-founded induction leads to a construction of inductive types by inflationary iteration, which has been utilized to justify cyclic proofs in the sequent calculus (Sprenger and Dam 2003). We are the first to utilize inflationary iteration in a type system.
- Well-founded induction alleviates the need for a semi-continuity check for sized types of recursive functions (Hughes et al. 1996; Abel 2008a) which sometimes disguises itself as a monotonicity check (Barthe et al. 2004; Blanqui 2004; Barthe et al. 2008; Sacchini 2013). Thus, we put type-based termination on leaner and better understandable foundations.
- Since we construct infinite objects by copattern matching, standard rewriting becomes strongly normalizing even for corecursive definitions, and productivity becomes an instance of termination. Thus, we achieve a unified treatment of recursion and corecursion that is central to type-based termination.
- Our typing rules are formulated as a bidirectional type-checking algorithm that can be implemented as such. See, e. g., MiniAgda (Abel 2012).
- We prove soundness of $\mathsf{F}_\omega^{\mathsf{cop}}$ by an untyped term model based on Girard's reducibility candidates. The proof exhibits semantic counterparts of pattern and copattern typing and accounts for incomplete and overlapping rewrite rules.

Due to lack of space, we leave out $\mathsf{F}_\omega^{\mathsf{cop}}$'s inference rules concerning the kind and type level, the description of program typing, and most details of the soundness proof. The full development can be found in the extended version (Abel and Pientka 2013).

2. Copatterns and Termination

Let us illustrate how to program with copatterns using a simple example of generating a stream of zeros. A streams s over an element type A is given by the two *observations* head and tail: We can inspect the head of s by applying the projection $s\ .\mathsf{head}$ and obtain an element of A. To obtain the tail of s, we use the projection $s\ .\mathsf{tail}$. We can then define the stream of zeros recursively by the following two clauses:

$$\begin{aligned} \mathsf{zeros}\ .\mathsf{head} &= 0 \\ \mathsf{zeros}\ .\mathsf{tail} &= \mathsf{zeros} \end{aligned}$$

More generally, zeros can be coded as $\mathsf{repeat}\ 0$ with

$$\begin{aligned} \mathsf{repeat}\ a\ .\mathsf{head} &= a \\ \mathsf{repeat}\ a\ .\mathsf{tail} &= \mathsf{repeat}\ a \end{aligned}$$

The left hand side of each clause is considering the definiendum, here repeat, in a *copattern*, here $\cdot\ a\ .\mathsf{head}$ and $\cdot\ a\ .\mathsf{tail}$, resp. A copattern consists of a hole, $\cdot$, applied to a sequence of patterns and/or projections. The hole is filled, e. g., by the definiendum. In this case, we have first a variable pattern, a, and then a projection head/tail.

The definition of repeat is *complete* because the given copatterns are covering all possible cases (Abel et al. 2013). Rewriting with the equations for repeat terminates in all situations, since one projection is consumed in each rewriting step. For example, projecting the $(n+1)$st element (counting from 0) of $\mathsf{repeat}\ a$, i. e., $\mathsf{repeat}\ a\ .\mathsf{tail}^{n+1}\ .\mathsf{head}$ reduces in one step to $\mathsf{repeat}\ a\ .\mathsf{tail}^{n}\ .\mathsf{head}$ and after n more steps to $\mathsf{repeat}\ a\ .\mathsf{head}$.

2.1 Example: Fibonacci

Let us look at programming with copatterns and type-based termination for a more interesting example, the stream of Fibonacci numbers. It can be elegantly implemented in terms of $\mathsf{zipWith}\ f\ s\ t$ which pointwise applies the binary function f to the elements of streams s and t.

$$\begin{aligned} \mathsf{zipWith}\ f\ s\ t\ .\mathsf{head} &= f\ (s\ .\mathsf{head})\ (t\ .\mathsf{head}) \\ \mathsf{zipWith}\ f\ s\ t\ .\mathsf{tail} &= \mathsf{zipWith}\ f\ (s\ .\mathsf{tail})\ (t\ .\mathsf{tail}) \end{aligned}$$

$$\begin{aligned} \mathsf{fib}\ .\mathsf{head} &= 0 \\ \mathsf{fib}\ .\mathsf{tail}\ .\mathsf{head} &= 1 \\ \mathsf{fib}\ .\mathsf{tail}\ .\mathsf{tail} &= \mathsf{zipWith}\ (+)\ \mathsf{fib}\ (\mathsf{fib}\ .\mathsf{tail}) \end{aligned}$$

The last equation states in terms of streams that the $(n+2)$nd element of the Fibonacci stream is the sum of the nth and the $(n+1)$st. It looks like fib is a terminating definition since $\mathsf{fib}\ .\mathsf{tail}\ .\mathsf{tail}$ only refers to fib and $\mathsf{fib}\ .\mathsf{tail}$, thus, one projection is removed in each recursive call. However, termination of fib is also dependent on good properties of $\mathsf{zipWith}$. For instance, the following faulty clause for $\mathsf{zipWith}$ would make $\mathsf{fib}\ .\mathsf{tail}\ .\mathsf{tail}\ .\mathsf{head}$ loop:

$$\mathsf{zipWith}\ f\ s\ t\ .\mathsf{head} = f\ (s\ .\mathsf{tail}\ .\mathsf{head})\ (t\ .\mathsf{tail}\ .\mathsf{head})$$

$$\begin{aligned} &\mathsf{fib}\ .\mathsf{tail}\ .\mathsf{tail}\ .\mathsf{head} \\ &\quad = \mathsf{zipWith}\ (+)\ \mathsf{fib}\ (\mathsf{fib}\ .\mathsf{tail})\ .\mathsf{head} \\ &\quad = (\mathsf{fib}\ .\mathsf{tail}\ .\mathsf{head}) + (\mathsf{fib}\ .\mathsf{tail}\ .\mathsf{tail}\ .\mathsf{head}) \\ &\quad = (\mathsf{fib}\ .\mathsf{tail}\ .\mathsf{head}) + (\mathsf{fib}\ .\mathsf{tail}\ .\mathsf{head}) + (\mathsf{fib}\ .\mathsf{tail}\ .\mathsf{tail}\ .\mathsf{head}) \\ &\quad = \dots \end{aligned}$$

The problem is that the faulty $\mathsf{zipWith}$ adds again one tail projection that has been removed in going from the original call $\mathsf{fib}\ .\mathsf{tail}\ .\mathsf{tail}$ to the recursive call $\mathsf{fib}\ .\mathsf{tail}$, thus, we are left with the same number of projections, leading to an infinite call cycle.

What we learn from this counterexample is that in order to reason about termination of stream expressions, we need to trade the naive image of streams as infinite sequences for a notion of streams that can safely be subjected to α many projections, where

$\alpha \leq \omega$ can be a natural number or (the smallest) infinity ω. We refer to such streams as *sized streams*, or streams having *depth* α. Clearly, if a stream of depth α is required, we can safely supply a stream of depth $\beta \geq \alpha$, thus, sized streams are subject to contravariant subtyping.

The original $\mathsf{zipWith}$ delivers, if called with input streams of depth α, an output stream of the same depth. This allows us to reason about the termination of fib as follows. We show that fib is a stream of arbitrary depth α by induction on $\alpha \leq \omega$. Cases $\alpha < 2$ are easy. The interesting case is $\alpha = n + 2$ when we take two tail projections and then another n projections, thus, $n + 2$ projections in total. Then we may assume (by induction hypothesis) that on the rhs taking up to $n + 1$ projections of fib is fine, thus, fib and $\mathsf{fib}\,.\mathsf{tail}$ behave well under another n projections—they both can be assigned depth n using subtyping. Passing them to $\mathsf{zipWith}\,(+)$ returns in turn a stream of the same depth n, hence the lhs $\mathsf{fib}\,.\mathsf{tail}\,.\mathsf{tail}$ can be assigned depth n and, consequently, fib depth $n + 2$, which was our goal.

The faulty $\mathsf{zipWith}$, however, needs streams of depth $n + 1$ to deliver a stream of depth n. Since $\mathsf{fib}\,.\mathsf{tail}$ can only safely be assumed to have depth n, not depth $n + 1$, the termination proof attempt fails, and rightfully so.

In this model proof we assumed that taking a projection will decrease the depth by exactly one. In the following, we will loosen this assumption and let projections take us to any strictly smaller depth.

2.2 Type-based termination for copatterns

In this section, we present the key ideas behind $\mathsf{F}_\omega^{\mathsf{cop}}$, our polymorphic core language for type-based termination checking of recursive definitions involving inductive and coinductive types. We illustrate how the integration of size expressions into the type system captures and mechanizes the informal reasoning about termination employed in the previous section.

Size quantification for inductive and coinductive types. Besides quantification over types $\forall A{:}{*}.\, B$ we have quantification over sizes $\forall i{<}a.\, B$. To unify these two forms of quantification we add to the base kind $*$ of types the base kinds $<a$ denoting sets of ordinals below a and conceive $\forall i{<}a.\, B$ as shorthand for $\forall i{:}({<}a).\, B$. Thus, size expressions fall in the same syntactic class as type expressions. We introduce a special ordinal ∞, the closure ordinal for all (co)inductive types we consider. As far as streams are concerned, ∞ can be thought of as ω. In general, valid size expressions are of the form $a ::= i + n \mid \infty + n$ where i is a size variable and n a concrete number (we drop $+0$).

The type of streams of depth a over element type A will be denoted by $\mathsf{Stream}^a A$, and we consider the following typing rules for the projections:

$$\frac{s : \mathsf{Stream}^a A}{s\,.\mathsf{head} : \forall i{<}a^\uparrow.\, A} \qquad \frac{s : \mathsf{Stream}^a A}{s\,.\mathsf{tail} : \forall i{<}a^\uparrow.\, \mathsf{Stream}^i A} \qquad (1)$$

These rules state that if you want to project a stream of depth a, you will need to provide a *witness* that you are able to do so, i. e., an ordinal $i < a^\uparrow$. In case of tail, this witness serves also as the depth of the projected stream. For instance, if $s : \mathsf{Stream}^{i+2} A$, then $s\,.\mathsf{tail}\,(i+1)\,.\mathsf{head}\,i : A$. *Bound normalization* $a^\uparrow$, defined by $(i+n)^\uparrow = i+n$ and $(\infty+n)^\uparrow = \infty+1$, allows us to turn bounds $a \geq \infty$ into $\infty + 1$ and project from the fixpoint $\mathsf{Stream}^\infty A$ without information loss. For $s : \mathsf{Stream}^\infty A$ we have $s\,.\mathsf{tail}\,\infty : \mathsf{Stream}^\infty A$ since $\infty < \infty^\uparrow = \infty + 1$, reflecting that the tail of a fully defined stream has infinite depth as well.

In practice, we often use the following derived rule which eliminates the universal quantifier and directly compares sizes.

$$\frac{s : \mathsf{Stream}^a A}{s\,.\mathsf{head}\,b : A}\; b < a^\uparrow \qquad \frac{s : \mathsf{Stream}^a A}{s\,.\mathsf{tail}\,b : \mathsf{Stream}^b A}\; b < a^\uparrow$$

More generally, following previous work (Abel et al. 2013), we represent coinductive types as recursive records νR, with $R = \{d_1 : F_1; \ldots; d_n : F_n\}$ giving (sized) types to the projections $d_{1..n}$ as follows:

$$\frac{r : \nu^a R}{r\,.d_k : \forall i{<}a^\uparrow.\, F_k(\nu^i R)}$$

For instance, with $\mathsf{Stream}^i A = \nu^i\{\mathsf{head} : \lambda X.\, A; \mathsf{tail} : \lambda X.\, X\}$ we obtain the typing of head and tail presented above (1). Considering R as a finite map from projections to type constructors, we write R_{d_k} for F_k.

Dually, inductive types are recursive variants μS with $S = \langle c_1 : F_1; \ldots; c_n : F_n\rangle$ and constructor typing

$$\frac{t : \exists i{<}a^\uparrow.\, F_k(\mu^i S)}{c_k\, t : \mu^a S}.$$

For instance, finite lists can be defined as follows: $\mathsf{List}^i A = \mu^i\langle \mathsf{nil} : \lambda X.\, 1;\ \mathsf{cons} : \lambda X.\, A \times X\rangle$. Integrating the quantifier rules, we derive the following inferences for constructors and destructors:

$$\frac{s : S_c(\mu^b S)}{c^b s : \mu^a S}\; b < a^\uparrow \qquad \frac{r : \nu^a R}{r\,.d\,b : R_d(\nu^b R)}\; b < a^\uparrow.$$

Specifying termination measures. The polymorphically typed version of $\mathsf{zipWith}$ officially looks as follows, where we write $\forall i{\leq}a$ as abbreviation for $\forall i{<}(a+1)$:

$$\begin{array}{l} \mathsf{zipWith} : \forall i{\leq}\infty.\, |i| \Rightarrow \forall A{:}{*}.\, \forall B{:}{*}.\, \forall C{:}{*}.\\ \qquad (A \to B \to C) \to \\ \qquad \mathsf{Stream}^i A \to \mathsf{Stream}^i B \to \mathsf{Stream}^i C \end{array}$$

$$\begin{array}{ll} \mathsf{zipWith}\ i\ A\ B\ C\ f\ s\ t\ .\mathsf{head}\ j & = f\ (s\ .\mathsf{head}\ j)\ (t\ .\mathsf{head}\ j)\\ \mathsf{zipWith}\ i\ A\ B\ C\ f\ s\ t\ .\mathsf{tail}\ j & = \mathsf{zipWith}\ j\ A\ B\ C\ f\\ & \qquad (s\ .\mathsf{tail}\ j)\ (t\ .\mathsf{tail}\ j) \end{array}$$

The first equation has type C and the second one type $\mathsf{Stream}^j C$. The kind of j is $<i$ due to the typing of head and tail, thus, zipWith is well-defined (and terminating) by induction on its first argument, the size argument. The associated termination measure is located after the size variable(s) and, in general, a tuple $|a, b, c|$ of size expressions under the lexicographic order.[2] In this case, it is just the unary tuple $|i|$, meaning that the termination measure is just the value of size variable i. The measure is not officially part of the type; it is rather an annotation that allows us to termination check the clauses without having to infer a termination order.

High-level idea of size-based termination checking. When we check a corecursive definition such as the second clause of zipWith we start with traversing the left hand side (lhs). We first introduce assumption $i{\leq}\infty$ into the context and now hit the measure annotation $|i|$ in the type. At this point we introduce the assumption $\mathsf{zipWith} : \forall j{\leq}\infty.\, |j|{<}|i| \Rightarrow \forall A{:}{*}.\, \forall B{:}{*}.\, \forall C{:}{*}.\, (A \to B \to C) \to \mathsf{Stream}^j A \to \mathsf{Stream}^j B \to \mathsf{Stream}^j C$ which will be used to check the recursive call on the right hand side (rhs). It has a constraint $|j| < |i|$, a lexicographic comparison of size expression tuples (which here just means $j < i$), that is checked before applying $\mathsf{zipWith}\ j$ to A. Continued checking of the lhs introduces further assumptions $A, B, C : *$, $f : A \to B \to C$, $s : \mathsf{Stream}^i A$, $t : \mathsf{Stream}^i B$, and $j < i$. Checking the rhs succeeds since the

[2] The notation for termination measures is taken from Xi (2002)

constraint $|j| < |i|$ is satisfied and s .tail j : $\mathsf{Stream}^j A$ and t .tail j : $\mathsf{Stream}^j B$.

In the following, we abbreviate $\forall A{:}*$ to just $\forall A$ and $\forall i{\le}\infty$ to just $\forall i$. With all size and type-arguments, the definition of the Fibonacci stream becomes:

$$\begin{array}{ll} \mathsf{fib} \ : \ \forall i.\, |i| \Rightarrow \mathsf{Stream}^i \mathbb{N} & \\ \mathsf{fib}\ i\ .\mathsf{head}\ j & = 0 \\ \mathsf{fib}\ i\ .\mathsf{tail}\ j\ .\mathsf{head}\ k & = 1 \\ \mathsf{fib}\ i\ .\mathsf{tail}\ j\ .\mathsf{tail}\ k & = \mathsf{zipWith}\ k\ \mathbb{N}\ \mathbb{N}\ \mathbb{N}\ (+)\ (\mathsf{fib}\ k)\ (\mathsf{fib}\ j\ .\mathsf{tail}\ k) \end{array}$$

In the last line, the lhs introduces size variables i and $j < i$ and $k < j$ and an assumption $\mathsf{fib} : \forall i'.\, |i'| < |i| \Rightarrow \mathsf{Stream}^{i'} \mathbb{N}$ and expects a rhs of type $\mathsf{Stream}^k \mathbb{N}$. Since $k < j < i$, both recursive calls are valid, and the expressions $\mathsf{fib}\ k$ and $\mathsf{fib}\ j$ .tail k both have type $\mathsf{Stream}^k \mathbb{N}$. With $\mathsf{zipWith}\ k\ \mathbb{N}\ \mathbb{N}\ \mathbb{N} : \mathsf{Stream}^k \mathbb{N} \to \mathsf{Stream}^k \mathbb{N} \to \mathsf{Stream}^k \mathbb{N}$, the rhs is well-typed, and fib is terminating.

2.3 Example: Stream processor

Ghani et al. (2009) describe programs for continuous stream functions $\mathsf{Stream}\ A \to \mathsf{Stream}\ B$ in terms of a mixed coinductive-inductive data type SP with two constructors $\mathsf{get} : (A \to \mathsf{SP}) \to \mathsf{SP}$ and $\mathsf{put} : (B \times \mathsf{SP}) \to \mathsf{SP}$. We use this example to illustrate how our foundation supports size-based reasoning on such mixed datatypes and lexicographic termination measures for mutually recursive functions. A stream processor can either get an element $v : A$ from the input stream and enter a new state, depending on the read value, or it can put an element $w : B$ on the output stream and enter a new state. To be productive, it can only read finitely many values from the input stream before writing a value on the output stream, thus, SP is actually a nesting of a least fixed-point into a greatest one: $\mathsf{SP} = \nu X.\, \mu Y.\, (A \to Y) + (B \times X)$. We express this nesting by the definition of two data types, an inductive variant SP_μ and a coinductive record type SP_ν.

$$\begin{array}{lcl} \mathsf{SP}^i_\mu\ X & = & \mu^i \langle \mathsf{get} : \lambda Y.\, A \to Y;\ \mathsf{put} : \lambda Y.\, B \times X \rangle \\ \mathsf{SP}^i_\nu & = & \nu^i \{\mathsf{out} : \lambda X.\, \mathsf{SP}^\infty_\mu X\} \end{array}$$

Inside the coinductive type, we use the inductive type SP_μ at size ∞ since we want to allow an arbitrary (finite) number of gets between two puts. We get the following derived rules for typing constructors and destructors:

$$\frac{f : A \to \mathsf{SP}^b_\mu X}{\mathsf{get}^b f : \mathsf{SP}^a_\mu X}\ b < a^\uparrow \qquad \frac{w : B \qquad sp : X}{\mathsf{put}^b (w, sp) : \mathsf{SP}^a_\mu X}\ b < a^\uparrow$$

$$\frac{sp : \mathsf{SP}^a_\nu}{sp\ .\mathsf{out}\ b : \mathsf{SP}^\infty_\mu\ \mathsf{SP}^b_\nu}\ b < a^\uparrow$$

In the context of stream processors it is convenient to consider streams as given by a single destructor force which returns head and tail in a pair, thus, $\mathsf{Str}^i A = \nu^i \{\mathsf{force} : \lambda X.\, A \times X\}$. Dedicated projections hd and tl can be defined by

$$\begin{array}{lcl} \mathsf{hd} & : & \forall i.\, \mathsf{Str}^{i+1} A \to A \\ \mathsf{hd}\ i\ s & = & \mathsf{fst}\ (s\ .\mathsf{force}\ i) \\[4pt] \mathsf{tl} & : & \forall i.\, \mathsf{Str}^{i+1} A \to \mathsf{Str}^i A \\ \mathsf{tl}\ i\ s & = & \mathsf{snd}\ (s\ .\mathsf{force}\ i) \end{array}$$

with fst and snd the obvious first and second projections from pairs. Via bound normalization, facilitating $\mathsf{Str}^\infty = \mathsf{Str}^{\infty+1}$, we obtain instances $\mathsf{hd}\ \infty : \mathsf{Str}^\infty A \to A$ and $\mathsf{tl}\ \infty : \mathsf{Str}^\infty A \to \mathsf{Str}^\infty A$.

Running a stream processor on an input stream produces an output stream as follows (informally coded in a Haskell-like language):

$$\begin{array}{lcl} \mathsf{run}\ (\mathsf{get}\ f)\ (v, vs) & = & \mathsf{run}\ (f\ v)\ vs \\ \mathsf{run}\ (\mathsf{put}(w, sp))\ vs & = & (w,\ \mathsf{run}\ sp\ vs) \end{array}$$

We represent this function via two mutually recursive functions, one handling SP_μ and one SP_ν:

$$\begin{array}{lcl} \multicolumn{3}{l}{\mathsf{run}_\mu : \forall i \forall j.\, |i, j+1| \Rightarrow \mathsf{SP}^j_\mu(\mathsf{SP}^i_\nu) \to \mathsf{Str}^\infty A \to B \times \mathsf{Str}^i B} \\ \mathsf{run}_\mu\ i\ j\ (\mathsf{get}^{j'} f) \quad\ vs & = & \mathsf{run}_\mu\ i\ j'\ (f\ (\mathsf{hd}\ \infty\ vs))\ (\mathsf{tl}\ \infty\ vs) \\ \mathsf{run}_\mu\ i\ j\ (\mathsf{put}^{j'}(w, sp))\ vs & = & (w,\ \mathsf{run}_\nu\ i\ sp\ vs) \\[4pt] \multicolumn{3}{l}{\mathsf{run}_\nu : \forall i.\, |i, 0| \Rightarrow \mathsf{SP}^i_\nu \to \mathsf{Str}^\infty A \to \mathsf{Str}^i B} \\ \mathsf{run}_\nu\ i\ sp\ vs\ .\mathsf{force}\ i' & = & \mathsf{run}_\mu\ i'\ \infty\ (sp\ .\mathsf{out}\ i')\ vs \end{array}$$

The recursive run_μ handles a sequence of gets terminated by put and emits the head of a forced stream $B \times \mathsf{Str}^i B$. The tail is produced by the corecursive run_ν which, upon forcing, calls run_μ again. The termination is guaranteed by the lexicographic measures, which decrease in each recursive call:

$$\begin{array}{lll} \mathsf{run}_\mu \to \mathsf{run}_\mu : & |i, j+1| > |i, j'+1| & \text{since } j > j' \\ \mathsf{run}_\mu \to \mathsf{run}_\nu : & |i, j+1| > |i, 0| & \\ \mathsf{run}_\nu \to \mathsf{run}_\mu : & |i, 0| \quad\ > |i', \infty+1| & \text{since } i > i' \end{array}$$

Note that since we are not doing induction on SP^i_ν, but coinduction into Str^i, we could use SP^∞_ν instead of SP^i_ν in the types of run_μ and run_ν. However, the given types are more precise: instead of a stream processor of infinite depth, they only require a stream processor of depth i to produce a stream of depth i.

3. Syntax

In this section, we formally define $\mathsf{F}^{\mathsf{cop}}_\omega$, our higher-order polymorphic lambda-calculus with sized inductive and coinductive types, polarized higher-order subtyping, and definitions by pattern and copattern matching. As in previous work (Abel 2008a) we choose System F_ω rather than System F as basis since the notion of a *type constructor* is required (at least, semantically) if one wants to talk its fixed-points, i. e., about (co)inductive types.

$$\begin{array}{lll} \mathsf{SizeVar} & \ni i, j & \\ \mathsf{SizeExp} & \ni a, b & ::= i + n \mid \infty + n \quad (n \ge 0) \\ \mathsf{SizeExp}^+ & \ni a^+, b^+ & ::= a \mid n \\ \mathsf{Measure} & \ni \mathfrak{m} & ::= \cdot \mid a^+, \mathfrak{m} \\[4pt] \mathsf{Pol} & \ni \pi & ::= \circ \mid + \mid - \mid \top \\ \mathsf{SizeCxt} & \ni \Psi & ::= \cdot \mid \Psi, i{:}\pi({<}a) \end{array}$$

Figure 1. Sizes and measures.

3.1 Sizes

Fig. 1 gives a grammar for sizes, measures, and size contexts. A *size expression* a consists of a base, which is either a size variable i or ∞, and an offset, a natural number n.

$$a ::= i + n \mid \infty + n$$

We omit the offset when 0. Each size variable i comes with a bound $i < a$, which is recorded in a *size context*

$$\Psi ::= \cdot \mid \Psi, i{:}\pi({<}a).$$

A size context is considered as finite map from size variables i to their *polarity* π (see below) and their *kind* $<a$. We write $\le a$ for $<(a+1)$ and size for $\le\infty$. *Extended size expressions* a^+ allow as a third base, n, i.e. just a natural number. *Measures* $\mathfrak{m}$ are tuples of extended size expressions. There are a number of trivial judgements concerning well-formedness and partial ordering of (extended) size expressions and measures (see Table 1). These judgements may use the bounds stored in size context Ψ and are all defined as expected; their inference rules can be found in the extended version.

$\Psi \vdash a$	size a is well-formed
$\Psi \vdash a < b$	strict size comparison
$\Psi \vdash a \leq b$	size comparison
$\Psi \vdash a^+$	extended size a^+ is well-formed
$\Psi \vdash a^+ < b^+$	strict comparison
$\Psi \vdash a^+ \leq b^+$	comparison
$\Psi \vdash_n \mathfrak{m}$	measure $\mathfrak{m}$ is a well-formed n-tuple
$\Psi \vdash \mathfrak{m} < \mathfrak{m}'$	strict lexicographic measure comparison
$\Psi \vdash \mathfrak{m} \leq \mathfrak{m}'$	lexicographic measure comparison
$\Psi \vdash \exists \Psi'$	Ψ' is consistent for each valuation of Ψ

Table 1. Size-related judgements.

In constraint-based systems, strong normalization is usually lost in inconsistent contexts.[3] While our size contexts Ψ are always consistent, i. e., enjoy a valuation[4] η of the declared size variables (by natural numbers even), we need sometimes a stronger property that a size context extension Ψ' is consistent with a fixed valuation η of Ψ, i. e., Ψ' must be consistent even when we apply η to its declared bounds. For instance, $i{\leq}\infty, j{<}i$ is consistent, but $j{<}i$ is not a consistent extension of $i{\leq}\infty$ under valuation $\eta(i) = 0$, since there is no solution for j. We write $\boxed{\Psi \vdash \exists \Psi'}$ if Ψ' consistently extends Ψ in this sense. This judgement is inspired by Blanqui and Riba (2006).

$$
\begin{array}{lllll}
\mathsf{SKind} & \ni \iota & ::= & * \mid o \mid \iota \to \iota' & \\
\mathsf{Kind} & \ni \kappa & ::= & * \mid {<}a \mid \pi\kappa \to \kappa' & \\
\mathsf{TyCxt} & \ni \Delta & ::= & \cdot \mid \Delta, X{:}\pi\kappa & \\
\mathsf{Cxt} & \ni \Gamma & ::= & \cdot \mid \Gamma, x : A \mid \Gamma, x : {}^{?}\!A & \\
\\
\mathsf{TyVar} & \ni X, Y, Z, i, j & & & \\
\mathsf{TyAtom} & \ni K & ::= & a \mid X \mid 1 \mid \times \mid \to \mid \forall_\kappa \mid \exists_\kappa & \\
\mathsf{Type} & \ni F, G, A, B, C & ::= & K \mid \lambda X{:}\iota.\, F \mid F\, G & \\
 & & \mid & \mu^a S \mid \nu^a R & \\
\\
\mathsf{Var} & \ni x, y, z & & & \\
\mathsf{Cons} & \ni c & & & \\
\mathsf{Proj} & \ni d & & & \\
\mathsf{Variant} & \ni S & ::= & \langle c_1{:}F_1; \ldots; c_n{:}F_n \rangle & n \geq 0 \\
\mathsf{Record} & \ni R & ::= & \{d_1{:}F_1; \ldots; d_n{:}F_n\} & n \geq 0 \\
\\
\mathsf{MType} & \ni {}'\!A, {}'\!B & ::= & \forall \Psi.\, \mathfrak{m} \Rightarrow A & \\
\mathsf{CType} & \ni {}^{?}\!A, {}^{?}\!B & ::= & \forall \Psi.\, \mathfrak{c} \Rightarrow A & \\
\mathsf{Cond} & \ni \mathfrak{c} & ::= & \mathfrak{m} {<} \mathfrak{m}' &
\end{array}
$$

Figure 2. Kinds and type constructors.

3.2 Kinds and type constructors

The type constructors of F_ω are assigned kinds $\iota ::= * \mid \iota \to \iota'$, with base kind $*$ classifying all proper types and function kinds $\iota \to \iota'$ the (higher-order) type operators. We add a second base kind $\iota ::= \cdots \mid o$ that classifies size expressions, which we locate at the type level, since they are computationally irrelevant and can be erased during compilation, just as the types are.

[3] For instance, in extensional type theory, $X : \mathsf{Type},\ p : X = (X \to X) \vdash (\lambda x{:}X.\, x\, x)(\lambda x{:}X.\, x\, x) : X$. The blame is on the false equality assumption $X = X \to X$ which is used for type conversion.

[4] A valuation η of size context Ψ is a map from size variables i to sizes $\eta(i)$ that fulfills the constraints for the size variables given by Ψ. Formally, $\eta(i) < [\![a]\!]_\eta$ must hold in case $i{:}\pi({<}a) \in \Psi$, where $[\![a]\!]_\eta$ is the value of size expression a in environment η.

These *simple kinds* ι form with the type constructor a simply-"typed" type-level lambda calculus. We *refine* these kinds into $\mathsf{F}^{\mathsf{cop}}_\omega$-kinds

$$\kappa ::= * \mid {<}a \mid \kappa \xrightarrow{\pi} \kappa'$$

where ${<}a$ refines o into the kind of size expressions $b < a$. The *polarized function kind* $\kappa \xrightarrow{\pi} \kappa'$, also written $\pi\kappa \to \kappa'$, allows us to express that the classified type constructor is co-variant ($\pi = +$), contravariant ($\pi = -$), constant ($\pi = \top$) or mixed-variant or of unknown variance ($\pi = \circ$). The polarities π are partially ordered $\circ \leq +, - \leq \top$ according to their information content. This and the order on size expressions induce a subkinding relation $\Psi \vdash \kappa \leq \kappa'$ on kinds of the same structure, i. e., the same underlying simple kind $|\kappa| = |\kappa'|$. Here, when comparing two o-kinds $({<}a) \leq ({<}b)$, we resort to size comparison $a \leq b$. The default variance is $\circ$ (no information) and we may omit it, writing simply $\kappa \to \kappa'$ or $\Psi, i{:}({<}a)$, which is further abbreviated by $\Psi, i{<}a$.

Kinding or *type variable contexts* $\Delta ::= \cdot \mid \Delta, X{:}\pi\kappa$, which provide scoping and kinding information for type constructors, generalize size contexts from bounds $({<}a)$ to arbitrary kinds κ. We may use a Δ where a Ψ is formally required, silently erasing all non-size variables from Δ. More generally, context *restriction* $\Delta \restriction \vec{X}$ of context Δ to a set of variables $\vec{X}$ deletes the bindings for all $Y \notin \vec{X}$ from Δ.

$\Psi \vdash \kappa$	kind κ is well-formed in Ψ
$\Psi \vdash \kappa \leq \kappa'$	κ is a subkind of κ'
$\Delta \vdash \Delta'$	kinding context Δ' is well-formed in Δ
$\Delta \vdash \exists \Delta'$	Δ' is consistent for each valuation of Δ

Table 2. Kind-related judgements.

The judgement $\Delta \vdash \exists \Delta'$ (see Table 2) states that Δ' is consistent for each valuation of Δ. Only the size declarations matter here, so it is a straightforward extension of $\Psi \vdash \exists \Psi'$.

Figure 2 contains a grammar for the type constructors of $\mathsf{F}^{\mathsf{cop}}_\omega$. Its core is a simply-kinded lambda-calculus $X \mid \lambda X{:}\iota.\, F \mid F\, G$ with constants $1, \times, \to, \forall_\kappa$, and $\exists_\kappa$ to form unit, product, function, universal, and existential types. Size expressions a are considered type constructors so that sizes can be abstracted over and applied. We use the following short-hands:

$\lambda X F$	for	$\lambda X{:}\iota.\, F$	if ι inferable
$A \times B$	for	$(\times)\, A\, B$	product type
$A \to B$	for	$(\to)\, A\, B$	function type
$\forall X{:}\kappa.\, A$	for	$\forall_\kappa (\lambda X{:}\lvert\kappa\rvert.\, A)$	universal type
$\exists X{:}\kappa.\, A$	for	$\exists_\kappa (\lambda X{:}\lvert\kappa\rvert.\, A)$	existential type
$\forall i{<}a.\, A$	for	$\forall_{<a}\, (\lambda i{:}o.\, A)$	bounded universal
$\exists i{<}a.\, A$	for	$\exists_{<a}\, (\lambda i{:}o.\, A)$	bounded existential
$\forall i.\, A$	for	$\forall i{:}\mathsf{size}.\, A$	"unbounded" universal
$\exists i.\, A$	for	$\forall i{:}\mathsf{size}.\, A$	"unbounded" existential.

We also write $\forall \Delta.\, A$ for the universal abstraction of all type variables of Δ in type A.

The simple kind annotation ι in $\lambda X{:}\iota.\, F$ allows us to infer a unique simple kind for closed type constructors. The simple kind of an open type constructor depends only on the simple kinds of its free type variables. This property simplifies the interpretation $[\![F]\!]$ of type constructors as set-theoretic functions on semantic types we will give later.

For the purpose of type checking, we are only interested in β-normal type constructors. We write $F\ @^\iota\ G$ for the normalizing application of F to an argument G of simple kind ι. We may write $@^\kappa$ instead of $@^{|\kappa|}$, or even just $@$.

Sized inductive $\mu^a S$ and coinductive types $\nu^a R$ are given in terms of *variant rows* S and *record rows* R. A variant row $S =$

$\langle c_1{:}F_1; \dots; c_n{:}F_n\rangle$ is a finite map from variant labels c_i, called *constructors*, to type constructors $S_{c_i} = F_i$. Dually, a record row R maps record labels d, called *destructors* or *projections*, to type constructors R_d. Instead of presenting, for instance, streams as $\nu^a X.\,\{\mathsf{head} : A;\ \mathsf{tail} : X\}$, we move the abstraction over X into the record row as $\nu^a\{\mathsf{head} : \lambda X.A;\ \mathsf{tail} : \lambda X.X\}$, in order to formulate the typing rules more conveniently.

Finally, we have *constrained types* $\forall\Psi.\,\mathfrak{m}{<}\mathfrak{m}' \Rightarrow A$ that allow its inhabitants to be used only if the condition $\mathfrak{m} < \mathfrak{m}'$ is fulfilled. We use them to restrict recursive calls to situations where the termination measure has decreased. Recursive function definitions come with *measured types* $'A ::= \forall\Delta.\ \mathfrak{m} \Rightarrow A$. These are not proper types but rather blueprints for constrained types. The idea is that kinding context Δ declares some size variables that are used in measure $\mathfrak{m}$ (and type A). When we analyze the body of a recursive function of measure type $'A$ and the variables of Δ are in scope (thus, the measure $\mathfrak{m}$ is well-formed), we make a copy $'B = \forall\Delta'.\,\mathfrak{m}' \Rightarrow A'$ of $'A$ by renaming the variables of Δ to Δ'. Then, by *measure replacement* $'B^{<\mathfrak{m}}$ we create the constrained type $\forall\Delta'.\ \mathfrak{m}'{<}\mathfrak{m} \Rightarrow A'$ which is used to type the recursive occurrences of the function in its body.

$\Delta \vdash A$	type A is well-formed
$\Delta \vdash F \rightrightarrows \kappa$	F has kind κ (inference)
$\Delta \vdash F \leftleftarrows \kappa$	F has kind κ (checking)
$\Delta \vdash \Gamma$	typing context Γ is well-formed
$\Delta \vdash A \leq A'$	A is subtype of A'
$\Delta \vdash F \leq^\pi F' \rightrightarrows \kappa$	F is higher-ord. subtype of F' (κ inferred)
$\Delta \vdash F \leq^\pi F' \leftleftarrows \kappa$	F is higher-ord. subtype of F' (κ given)

Table 3. Type-related judgements.

Table 3 lists judgements for well-kindedness and partial ordering of types and type constructors. The judgements for types A only invoke the judgments for type constructors F in checking mode at base kind ($\leftleftarrows *$). The judgements for constructors are *bidirectional* with inference mode that computes the kind κ and checking mode that starts with a given κ. Bidirectional checking is complete since we are only interested in normal type constructors.

The rules for these judgements are given in an extended version of this article. A thorough discussion of polarized higher-order subtyping, i. e., subtyping for type constructors that take variance into account, is available in Abel (2008b) and Steffen (1998), we just recapitulate the basic principle here: A constructor F with $X_1{:}\pi_1\kappa_1, \dots, X_n{:}\pi_n\kappa_n \vdash F \leftleftarrows \kappa$ is interpreted as an operator

$$\lambda X_1 \dots \lambda X_n.F\ :\ \kappa_1 \stackrel{\pi_1}{\to} \dots \kappa_n \stackrel{\pi_n}{\to} \kappa$$

with variance given as noted in its kinding context. This induces the kinding rules, for instance $X{:}-*, Y{:}+* \vdash X \to Y : *$ is valid since function space is contravariant in its domain and covariant in its codomain. In particular, the hypothesis rule $X{:}\pi\kappa \vdash X : \kappa$ is only valid if $\pi \leq +$, i. e., $\pi = \circ$ which just states that $\lambda X.X : \kappa \to \kappa$ is a well-formed operator, or $\pi = +$ which additionally states that $\lambda X.X$ is monotone. Using the hypothesis rule on $\pi = -$ or $\pi = \top$ is invalid since $\lambda X.X$ is neither an antitone nor a constant operator.

Given a partial order $G \leq G'$, its π-parameterized version $G \leq^\pi G'$ can be defined as follows:

$$\begin{array}{lcl} G \leq^+ G' & = & G \leq G' \\ G \leq^- G' & = & G' \leq G \\ G \leq^\circ G' & = & G \leq G' \text{ and } G' \leq G \\ G \leq^\top G' & = & \text{true} \end{array}$$

π-variance of a constructor $F \rightrightarrows \pi\kappa \to \kappa'$ means that $F\,G \leq F\,G' \rightrightarrows \kappa$ whenever $G \leq^\pi G' \leftleftarrows \kappa$. (The reader is advised to play through the four cases for π in his mind.) Theoretically, the π-parameterized versions $\Delta \vdash F \leq^\pi F' \dots$ of higher-order subtyping could be defined from a non-parameterized version $\Delta \vdash F \leq F' \dots$, but to avoid the potential exponential blowup due to duplication of work in case of $\leq^\circ$, the π-parameterized versions are taken as primitive.

Exp	$\ni r, s, t$	$::= u \mid v \mid \lambda\vec{D}$	term
Intro	$\ni v$	$::= () \mid (t_1, t_2) \mid c\,t \mid {}^G t$	introduction
App	$\ni u$	$::= x \mid f \mid r\,e$	applicative
Fun	$\ni f, g$		function name
Elim	$\ni e$	$::= t \mid G \mid .d$	elimination
Pat	$\ni p$	$::= x \mid () \mid (p_1, p_2) \mid c\,p \mid {}^X p$	pattern
Copat	$\ni q$	$::= p \mid X \mid .d$	copattern
PatSp	$\ni \mathbf{q}$	$::= \vec{q}$	pattern spine
DCl	$\ni D$	$::= \{\mathbf{q} \to t\}$	def. clause
Def	$\ni \vec{D}$	$::= \{D_1; \dots; D_n\}$	def. clauses

Figure 3. Terms, (co)patterns, and clauses.

3.3 Terms and (co)patterns

Figure 3 presents the abstract syntax of $\mathsf{F}_\omega^{\mathsf{cop}}$ *terms* t, which are categorized into *introductions* v, *applicative terms* u, and *anonymous objects* $\lambda\vec{D}$. Introductions $()$, (t_1, t_2), $c\,t$ and ${}^G t$ construct tuples and inductive and existential types. Applicative terms x, f, and $r\,e$ are identifiers and generalized applications of a term r to an *elimination* e, which can be a term s for function elimination, a type G for instantiation of a polymorphic function, or a destructor $.d$ for projection from a coinductive type.

For each introduction form v we have the corresponding form of pattern p, and for each elimination form e there is a copattern q. Application copatterns are just patterns p to match the argument, type application copatterns Q are either type variables X or the special size pattern ∞, which matches anything, and projection copatterns are simply destructors d that match the same destructor in an elimination. A sequence of $\vec{q}$ of copatterns is called a pattern spine $\mathbf{q}$, in correspondence to an *elimination spine* $\vec{e}$.

Generalized lambda abstraction $\lambda\vec{D}$ introduces an object whose behavior is given by the clauses $\vec{D}$, each of which consists of a lhs, a (possibly empty) copattern sequence $\vec{q}$, and a rhs, a term t. Objects subsume both record and λ expressions of traditional functional languages. Here are a few simple examples:

$\lambda\{x \to t\}$	ordinary λ-abstraction $\lambda x t$
$\lambda\{X \to t\}$	type abstraction $\Lambda X t$
$\lambda\{(x, y) \to x\}$	first projection from pair
$\lambda\{{}^X x\,y \to y\,X\,x\}$	elimination of existential
$\lambda\{A\,x\,y\,.\mathsf{head}\,\infty \to x$	
$;A\,x\,y\,.\mathsf{tail}\ \ \infty \to y\}$	cons for $\mathsf{Stream}^\infty A$
$\lambda\{\cdot \to s;\ \cdot \to t\}$	non-deterministic choice $s \oplus t$

The meaning, given by the operational semantics, is that whenever $\lambda\vec{D}$ is applied to a sequence of eliminations $\vec{e}$ that match the copatterns $\vec{q}$ of a clause with rhs t under a substitution σ and a type substitution τ, then $(\lambda\vec{D})\,\vec{e}$ reduces to $t\sigma\tau$, the rhs instantiated by the substitutions computed from pattern matching. Using $\boxed{\vec{e} \,/\, \vec{q} \searrow \sigma;\tau}$ for *pattern matching*, the basic rule for *contraction* $\boxed{r \mapsto r'}$ becomes:

$$\frac{\vec{e} \,/\, \mathbf{q}_k \searrow \sigma;\tau}{\lambda\{\overrightarrow{\mathbf{q} \to t}\}\ \vec{e}\,\vec{e}\,' \mapsto t_k\sigma\tau\,\vec{e}\,'}$$

As usual, r is called a *redex* and r' its *reduct* if $r \mapsto r'$. We allow overlapping lhss, a spine $\vec{e}$ may match different pattern spines $\mathbf{q}$, resulting in different contractions of the same redex. Also, if no lhs in the clauses $\vec{D}$ matches $\vec{e}$, the expression $\lambda\vec{D}\,\vec{e}$ is *stuck*. While a coverage checker as described in previous work (Abel et al. 2013) could exclude overlapping and incomplete clauses in well-typed programs, we do not require coverage in this paper and confine ourselves to show *strong normalization*, i. e., the absence of infinite reduction sequences.

Not all stuck terms are pathological; since we are matching the whole pattern spine in one go, partially applied functions such as $\lambda\{xy \to t\}s$ are stuck, but can become unstuck if more arguments are supplied. The existence of partially applied functions will require careful treatment in the normalization proof, because non-contractibility of a non-introduction term is not preserved under application (as would be in the case of λ-calculus).

$\mathsf{Decl} \ni \delta$	$::= f : A = \vec{D}$	declaration
$\mathsf{MDecl} \ni {}'\delta$	$::= f : {}'A = \vec{D}$	declaration with measure
$\mathsf{Block} \ni \beta$	$::= \mathsf{mutual}_m\, {}'\vec{\delta}$	mutual block
$\mathsf{Prg} \ni P$	$::= \vec{\beta}; u$	program
$\mathsf{Sig} \ni \Sigma$	$::= \vec{\delta}$	signature

Figure 4. Declarations, blocks, and programs.

3.4 Declarations and programs

An $\mathsf{F}_\omega^{\mathsf{cop}}$ program consists of a sequence $\vec{\beta}$ of mutual blocks and an applicative term u, the *entry point* (this could be the name of the main function or a call to the main function with some initial arguments). Each *mutual block* $\mathsf{mutual}_m {}'\vec{\delta}$ is a sequence ${}'\vec{\delta}$ of mutually recursive declarations with a lexicographic termination measure of length m. Each *declaration* $f : {}'A = \vec{D}$ assigns to a function symbol f its measured type ${}'A$ and its clauses $\vec{D}$. Measures serve their purpose during checking of the mutual block and are discarded afterwards. Erasure of measure $(\!|{}'\delta|\!)$ yields a (unmeasured) declaration $f : A = \vec{D}$; after checking a mutual block and erasing the measures, the individual declarations of the block become part of the signature Σ which is used for type-checking and evaluation of the remainder of the program. An applied function $f\,\vec{e}$ reduces if one of its clauses does:

$$\frac{(\lambda\vec{D})\,\vec{e} \mapsto t}{f\,\vec{e} \mapsto t}(f : A = \vec{D}) \in \Sigma$$

The one-step reduction relation $\boxed{t \longrightarrow t'}$ is the compatible closure of the contraction relation $t \mapsto t'$, i. e., $t \longrightarrow t'$ if t' is the result of contracting exactly one redex (in an arbitrary subterm of) t. Strong normalization of reduction will be shown to hold for well-typed programs.

$\Delta;\Gamma \vdash r \rightrightarrows C$	Infer type C for term r
$\Delta;\Gamma \vdash t \leftleftarrows C$	Term t checks against type C
$\Delta;\Gamma \vdash \{\mathbf{q} \to t\} \leftleftarrows A$	Clause $\{\mathbf{q} \to t\}$ checks against type A
$\Delta;\Gamma \vdash \vec{D} \leftleftarrows A$	Clauses D check against type A
$\Delta;\Gamma \vdash_{\Delta_0} p \leftleftarrows A$	Pattern p checks against type A
$\Delta;\Gamma \mid A \vdash_{\Delta_0} \mathbf{q} \rightrightarrows C$	Pattern spine $\mathbf{q}$ eliminates A into C

Table 4. Type checking.

3.5 Type checking

Table 4 lists the judgements involved in type checking $\mathsf{F}_\omega^{\mathsf{cop}}$ programs. Type-checking terms is bidirectional and a straightforward adaption of Abel et al. (2013) to polymorphism, bounded quantification, and constraints. The rules are given in figures 5 and 6, and we briefly explain them.

Inference $\boxed{\Delta;\Gamma \vdash r \rightrightarrows C}$. A function symbol f's type $\Sigma(f)$ is looked up in the signature, and a variable x's type $\Gamma(x)$ in the typing context. If $\Gamma(x)$ is a constrained type $\forall\Psi.\ \mathfrak{c} \Rightarrow A$, the variable x must be immediately applied to size arguments $\vec{a}$ satisfying both Ψ and the condition $\mathfrak{c}$; after all, a constrained type is, for consistency reasons, not a proper type for an expression. An application $r\,s$ of a function r of inferred type $A \to B$ has type B if the argument s checks against type A. Instantiation $r\,G$ of a polymorphic term r of inferred type $\forall_\kappa F$ has type $F\ @^\kappa\ G$ if G has kind κ. In particular, r could be of type $\forall i{<}a.\ A$, then G must be a size expression $<\ a$ to succeed. If r is of coinductive type $\nu^a R$, then $r\,.d$ has type $\forall j{<}a^\uparrow.R_d\,(\nu^j R)$, see Section 2.3.

There are two rules to switch direction. Checking r against type C succeeds if r's type is inferred as A and A is a subtype of C. Also, we can add *type ascription* $(t : A)$ to the term language; then inference of $(t : A)$ succeeds and yields A if A is a well-formed type and t checks against A. While type ascription is needed to bidirectionally type check redexes or stuck terms, it is dispensable if one confines to checking normal terms (in the sense that no elimination is applied to a λ in the source program). We will consider type ascriptions be removed before execution of the program, so they do not pop up in the operational and denotational semantics.

Checking $\boxed{\Delta;\Gamma \vdash t \leftleftarrows C}$. Introductions and λs are checked against a given type. Checking a pair ${}^G t$ of a type expression G and a term t against an existential type $\exists_k F$ succeeds if G has kind κ and t is of the correct instance $F\ @^\kappa\ G$. Checking a constructor term $c\,t$ against an inductive type $\mu^a S$ succeeds if t checks against $\exists j < a^\uparrow.\, S_c\,(\mu^j S)$. This means that t should be essentially a pair ${}^b t'$ of a size $b < a^\uparrow$ and t' be a correct argument to constructor c, i. e., having variant S_c instantiated to $\mu^j S$. If $a \geq \infty$, by bound normalization $b = \infty$ is a valid size index, which implies that in a value v in the fixpoint $\mu^\infty S$ all size witnesses can uniformly be ∞. To check $\lambda\vec{D}$ we check all clauses D_k.

Clause checking $\boxed{\Delta;\Gamma \vdash \{\mathbf{q} \to t\} \leftleftarrows A}$. We first check that pattern spine $\mathbf{q}$ eliminates indeed type A. As a result, we obtain a kinding context Δ' which binds the type variables X contained in $\mathbf{q}$ and a typing context Γ' which binds the pattern variables x contained in $\mathbf{q}$'s patterns, and a remaining type C of lhs and rhs. We now need to make sure that $\Delta \vdash \exists\Delta'$ such that any valuation of Δ can be extended to a valuation of Δ'. Complementing the original contexts $\Delta;\Gamma$ by the pattern contexts $\Delta';\Gamma'$ we check the rhs t against C.

Pattern spine checking $\boxed{\Delta;\Gamma \mid A \vdash_{\Delta_0} \mathbf{q} \rightrightarrows C}$. We eliminate type A which is well-formed in Δ_0. If there are no copatterns in $\mathbf{q}$, thus, the clause has an empty lhs, we simply return A which must be the type of the rhs. If we encounter an application pattern p, the eliminated type must be a function type $A \to B$. We check p against A and obtain pattern contexts $\Delta_1;\Gamma_1$. We continue to check the remaining copatterns, obtaining more pattern contexts $\Delta_2;\Gamma_2$ and a result type C, which we return together with the concatenated pattern contexts. Concatenation, and thus, pattern spine checking fails if the contexts do not have disjoint domains. A common variable would mean a non-linear lhs, which we exclude.

If we encounter a projection pattern $.d$, the eliminated type must be a coinductive type $\nu^a R$. Taking projection $.d$ yields type $\forall j{<}a^\uparrow.\ R_d(\nu^j R)$, thus, we continue to eliminate this type by ap-

$\boxed{\Delta;\Gamma \vdash r \rightrightarrows C}$ Expression typing (inference mode). In: Δ, Γ, r with $\Delta \vdash \Gamma$. Out: C with $\Delta \vdash C$

$$\frac{}{\Delta;\Gamma \vdash f \rightrightarrows \Sigma(f)} \qquad \frac{(x{:}A) \in \Gamma}{\Delta;\Gamma \vdash x \rightrightarrows A} \qquad \frac{(x : \forall\Psi.\, \mathfrak{c} \Rightarrow A) \in \Gamma \qquad \Delta \vdash \vec{a} \leftleftarrows \Psi \qquad \tau = \vec{a}/\hat{\Psi} \qquad \Delta \vdash \mathfrak{c}\tau}{\Delta;\Gamma \vdash x\,\vec{a} \rightrightarrows A\tau}$$

$$\frac{\Delta;\Gamma \vdash r \rightrightarrows A \to B \qquad \Delta;\Gamma \vdash s \leftleftarrows A}{\Delta;\Gamma \vdash r\,s \rightrightarrows B} \qquad \frac{\Delta;\Gamma \vdash r \rightrightarrows \nu^a R}{\Delta;\Gamma \vdash r.d \rightrightarrows \forall j{<}a^{\uparrow}.\, R_d\,(\nu^j R)} \qquad \frac{\Delta;\Gamma \vdash r \rightrightarrows \forall_\kappa F \qquad \Delta \vdash G \leftleftarrows \kappa}{\Delta;\Gamma \vdash r\,G \rightrightarrows F \mathbin{@^\kappa} G}$$

Switching.

$$\frac{\Delta \vdash A \qquad \Delta;\Gamma \vdash t \leftleftarrows A}{\Delta;\Gamma \vdash (t : A) \rightrightarrows A} \qquad \frac{\Delta;\Gamma \vdash r \rightrightarrows A \qquad \Delta \vdash A \leq C}{\Delta;\Gamma \vdash r \leftleftarrows C}$$

$\boxed{\Delta;\Gamma \vdash t \leftleftarrows C}$ Expression typing (checking mode). In: $\Delta;\Gamma, t, C$ with $\Delta \vdash \Gamma$ and $\Delta \vdash C$. Out: success/failure.

$$\frac{}{\Delta;\Gamma \vdash () \leftleftarrows 1} \qquad \frac{\Delta;\Gamma \vdash t_1 \leftleftarrows A_1 \qquad \Delta;\Gamma \vdash t_2 \leftleftarrows A_2}{\Delta;\Gamma \vdash (t_1, t_2) \leftleftarrows A_1 \times A_2} \qquad \frac{\Delta;\Gamma \vdash t \leftleftarrows \exists j{<}a^{\uparrow}.\, S_c\,(\mu^j S)}{\Delta;\Gamma \vdash c\,t \leftleftarrows \mu^a S}$$

$$\frac{\Delta \vdash G \leftleftarrows \kappa \qquad \Delta;\Gamma \vdash t \leftleftarrows F \mathbin{@^\kappa} G}{\Delta;\Gamma \vdash {}^{G}t \leftleftarrows \exists_\kappa F} \qquad \frac{\Delta;\Gamma \vdash \vec{D} \leftleftarrows A}{\Delta;\Gamma \vdash \lambda\vec{D} \leftleftarrows A}$$

$\boxed{\Delta;\Gamma \vdash D \leftleftarrows A}$ and $\boxed{\Delta;\Gamma \vdash \vec{D} \leftleftarrows A}$ definition typing. In: Δ, Γ, A, D or $\vec{D}$ with $\Delta \vdash \Gamma$ and $\Gamma \vdash A$. Out: success/failure.

$$\frac{\Delta';\Gamma' \mid A \vdash_\Delta \vec{q} \rightrightarrows C \qquad \Delta \vdash \exists\Delta' \qquad \Delta, \Delta';\Gamma, \Gamma' \vdash t \leftleftarrows C}{\Delta;\Gamma \vdash \{\vec{q} \to t\} \leftleftarrows A} \qquad \frac{\Delta;\Gamma \vdash D_k \leftleftarrows A \text{ for all } k}{\Delta;\Gamma \vdash \vec{D} \leftleftarrows A}$$

Figure 5. Type checking rules.

$\boxed{\Delta;\Gamma \vdash_{\Delta_0} p \leftleftarrows A}$ Pattern typing (linear). In: Δ_0, p, A with $\Delta_0 \vdash A$. Out: Δ, Γ with $\Delta_0, \Delta;\Gamma \vdash p \leftleftarrows A$.

$$\frac{}{\cdot; x{:}A \vdash_{\Delta_0} x \leftleftarrows A} \qquad \frac{}{\cdot;\cdot \vdash_{\Delta_0} () \leftleftarrows 1} \qquad \frac{\Delta_1;\Gamma_1 \vdash_{\Delta_0} p_1 \leftleftarrows A_1 \qquad \Delta_2;\Gamma_2 \vdash_{\Delta_0} p_2 \leftleftarrows A_2}{\Delta_1, \Delta_2;\Gamma_1, \Gamma_2 \vdash_{\Delta_0} (p_1, p_2) \leftleftarrows A_1 \times A_2}$$

$$\frac{\Delta;\Gamma \vdash_{\Delta_0} p \leftleftarrows \exists j{<}a^{\uparrow}.\, S_c\,(\mu^j S)}{\Delta;\Gamma \vdash_{\Delta_0} c\,p \leftleftarrows \mu^a S} \qquad \frac{\Delta;\Gamma \vdash_{\Delta_0, X{:}\kappa} p \leftleftarrows F \mathbin{@^\kappa} X}{X{:}\kappa, \Delta;\Gamma \vdash_{\Delta_0} {}^{X}p \leftleftarrows \exists_\kappa F}$$

$\boxed{\Delta;\Gamma \mid A \vdash_{\Delta_0} \vec{q} \rightrightarrows C}$ Pattern spine typing. In: $\Delta_0, A, \vec{q}$ with $\Delta_0 \vdash A$. Out: Δ, Γ, C with $\Delta_0, \Delta;\Gamma \vdash C$ and $\Delta_0, \Delta;\Gamma, z{:}A \vdash z\,\vec{q} \rightrightarrows C$.

$$\frac{}{\cdot;\cdot \mid A \vdash_{\Delta_0} \cdot \rightrightarrows A} \qquad \frac{\Delta_1;\Gamma_1 \vdash_{\Delta_0} p \leftleftarrows A \qquad \Delta_2;\Gamma_2 \mid B \vdash_{\Delta_0} \vec{q} \rightrightarrows C}{\Delta_1, \Delta_2;\Gamma_1, \Gamma_2 \mid A \to B \vdash_{\Delta_0} p\,\vec{q} \rightrightarrows C}$$

$$\frac{\Delta;\Gamma \mid \forall j{<}a^{\uparrow}.\, R_d\,(\nu^j R) \vdash_{\Delta_0} \vec{q} \rightrightarrows C}{\Delta;\Gamma \mid \nu^a R \vdash_{\Delta_0} .d\,\vec{q} \rightrightarrows C} \qquad \frac{\Delta;\Gamma \mid F \mathbin{@^\kappa} X \vdash_{\Delta_0, X{:}\kappa} \vec{q} \rightrightarrows C}{X{:}\kappa, \Delta;\Gamma \mid \forall_\kappa F \vdash_{\Delta_0} X\,\vec{q} \rightrightarrows C}$$

Figure 6. Pattern Typing.

plying it to a fresh size variable. The general form of a universal type $\forall_\kappa F$ is eliminated by a type variable pattern X; we record $X{:}\kappa$ in the type variable pattern context and continue eliminating $F \mathbin{@^\kappa} X$.

Pattern typing $\boxed{\Delta;\Gamma \vdash_{\Delta_0} p \leftleftarrows A}$. This judgement checks pattern p against type A which is valid in kinding context Δ_0, and returns pattern contexts $\Delta;\Gamma$. Pattern x succeeds against any type, returning singleton context $x{:}A$. The empty tuple $()$ succeeds against the unit type 1, binding no variables. The pair pattern (p_1, p_2) succeeds against the product type $A_1 \times A_2$ if each component p_i checks against its type A_i. The resulting pattern contexts are concatenated, checking for disjointness. A constructor pattern $c\,p$ checks against an inductive type $\mu^a S$ if p checks against $\exists j < a^{\uparrow}.\, S_c\,(\mu^j S)$. The latter succeeds if $p = {}^{j}p'$, then we add size variable $j{<}a$ to the pattern context and continue checking p' against $S_c\,(\mu^j S)$. This is an instance of checking against the general existential type $\exists_\kappa F$.

In the next section, we will validate all the typing rules by exhibiting a semantics of strongly normalizing terms based on Girard's reducibility candidates (Girard et al. 1989).

4. Semantics

In this section we show strong normalization of $\mathsf{F}^{\mathsf{cop}}_\omega$ by a term model. Types are interpreted as reducibility candidates à la Girard adapted to our needs. Our semantic constructions rely only on the terms and the operational semantics of $\mathsf{F}^{\mathsf{cop}}_\omega$, not to the types, kinds, or inference rules. Based on the operational semantics, semantic types and kinds are constructed that interpret the syntactic types, yet syntactic types are never used for semantic constructions.[5] We consider this conceptual hygiene important from a philosophic perspective: we use types just as a vehicle to assign properties to our programs; clearly, they have no run-time significance. While in the end we managed to keep syntactic types out of the semantic constructions, it was hard to get the semantic counterpart (Lemma 9) of pattern spine typing (Figure 6) right.

One clarification: Since $\mathsf{F}_\omega^{\mathsf{cop}}$ has Church-style polymorphism with explicit type abstraction and application, we can of course not talk about terms and operational semantics without mentioning syntactic types. However, we never refer to the structure of syntactic types, they remain abstract, and we could remove everything but type variables from our type language without altering the construction of semantic types and semantic typing "judgements". In particular, in the construction of the semantic universal type $\forall_{\mathcal{K}}\mathcal{F} = \{r \in \mathsf{SN} \mid r\, G \in \mathcal{F}(\mathcal{G}) \text{ for all } G \in \mathsf{Type}, \mathcal{G} \in \mathcal{K}\}$ there is no connection between the syntactic type constructor G and the semantic type constructor $\mathcal{G}$ (of semantic kind $\mathcal{K}$). Type applications serve only to make type-checking decidable, they do not play any role in evaluation.

Preliminaries. We use partially applied relations to denote sets. For instance, we write $(t \longrightarrow \_)$ or simply $t\!\longrightarrow$ for the set $\{t' \mid t \longrightarrow t'\}$ of reducts of t. Similarly, $<\!\alpha = \{\beta \mid \beta < \alpha\}$. The identity substitution is denoted by σ_{id}.

Strong normalization. Classically, a term t is strongly normalizing if it admits no infinite reduction sequences $t \longrightarrow t_1 \longrightarrow t_2$ starting with t. Inductively, we define $t \in \mathsf{SN}$ if all of its reducts are already in SN:

$$\frac{(t \longrightarrow \_) \subseteq \mathsf{SN}}{t \in \mathsf{SN}}$$

Naturally, if $t \in \mathsf{SN}$ then all its reducts and subterms are also strongly normalizing.

We extend the notion SN to other syntactic categories: An elimination e is strongly normalizing, $e \in \mathsf{SN}$, if it either is not a term (but a type G or a projection $.d$), or if it is a strongly normalizing term. A definition clause $D = \{\vec{q} \to t\}$ is strongly normalizing if $t \in \mathsf{SN}$.

Simulation. Our typing rules (see Figure 5) state that a definition $\lambda\vec{D} : A$ or $(f : A = \vec{D})$ is well-typed if each of the clauses D_k is of type A, individually. In the absence of a coverage check, there is no concept of "the clauses make sense *together*". We would like to see this independence of clauses reflected in our semantics. In particular, we would like to have *compositionality*, i. e., if each clause of a definition is semantically meaningful (in particular, does not lead to non-termination), then the clauses are meaningful together. For functions, our type-checker works exactly like that: each clause is checked individually, using the termination measure; an interaction between clauses need not be taken into account.

One idea is to say that a defined function $f : A = \vec{D}$ reduces non-deterministically to one of its clauses D_k, however, this immediately destroys strong normalization, because D_k might mention f. We need to defer unfolding of f until the pattern of one of its clauses matches. Thus, instead we say that $f\,\vec{e}$ reduces if $(\lambda\vec{D})\vec{e}$ reduces; f is simulated by its clauses $\vec{D}$. In general, a term r is *simulated* by terms $\vec{r}$, written $\boxed{r \rhd \vec{r}}$, iff each of its contractions under some eliminations is accounted for by one of the terms $\vec{r}$, formally $\forall\vec{e}, t.\ r\,\vec{e} \mapsto t \implies \exists k.\ r_k\,\vec{e} \mapsto t$. Closing reducibility candidates by simulation is one of the new ideas of our proof.

Lemma 1 (Simulation)**.**

1. *$\lambda\{D_1; \dots; D_n\} \rhd \lambda D_1, \dots, \lambda D_n$.*
2. *If $(f : A = \vec{D}) \in \Sigma$ then $f \rhd \lambda\vec{D}$.*
3. *If $r \rhd r_1, \dots, r_n$ then $r\,e \rhd r_1\,e, \dots, r_n\,e$.*

[5] Humbly following the masters (Vouillon and Melliès 2004).

4.1 Semantic Types

In order to show strong normalization we model types as sets of strongly normalizing terms, more precisely, as reducibility candidates à la Girard. We choose reducibility candidates over Tait's saturated sets, since they allow us to show strong normalization in the absence of standardization and confluence. As a consequence, we can model definitions with incomplete and overlapping patterns.

A set of terms $\mathcal{A}$ is a *reducibility candidate* (Girard et al. 1989), written $\mathcal{A} \in \mathsf{CR}$, if the following conditions hold.

CR1 $\mathcal{A} \subseteq \mathsf{SN}$: "each term in $\mathcal{A}$ is strongly normalizing".

CR2 if $t \in \mathcal{A}$ then $(t \longrightarrow \_) \subseteq \mathcal{A}$: "$\mathcal{A}$ is closed under reduction".

CR3 if $t \in \mathsf{Ne}$ and $(t \longrightarrow \_) \subseteq \mathcal{A}$ then $t \in \mathcal{A}$: "$\mathcal{A}$ contains a neutral already if all its redexes are in $\mathcal{A}$".

CR4 if $t \notin \mathsf{Intro}$ and $(t \longrightarrow \_) \subseteq \mathcal{A}$ and $t \rhd \vec{t} \in \mathcal{A}$ then $t \in \mathcal{A}$: "$\mathcal{A}$ is closed under simulation".

Condition **CR4**, is new; it introduces multi-clause objects $\lambda\vec{D}$ and function symbols f into a semantic type (candidate).

Lemma 2 (Multi-clause objects)**.**

1. *If $\lambda D_1, \dots, \lambda D_n \in \mathcal{A}$ then $\lambda\vec{D} \in \mathcal{A}$.*
2. *If $(f : A = \vec{D}) \in \Sigma$ and $\lambda\vec{D} \in \mathcal{A}$, then $f \in \mathcal{A}$.*

In **CR3**, Ne is a suitable set of so-called *neutral* terms. These are "good", i. e., inhabit a candidate, as soon as all their reducts are good. For Girard's technique to work, neutral terms need to include redexes such as $(\lambda x.t)\,s\,\vec{e}$ and variables x, and need to be closed under application, i. e., r neutral implies $r\,s$ neutral. In case of pure lambda calculus, any term which is not a lambda-abstraction can be considered neutral.

In our setting of matching the whole pattern spine $\vec{q}$ against the eliminations $\vec{e}$, things are more subtle. For instance, the partial application $\lambda\{x\,y \to x\,x\}\,\delta$ with $\delta = \lambda\{x \to x\,x\}$ is stuck (and even in normal form). However, it cannot be neutral and inhabit every candidate (following **CR3**), in particular semantic function types, since it reduces to the diverging term $\delta\,\delta$ if applied to one more argument. Thus, we can only accept stuck terms as neutral which cannot become unstuck by extra eliminations. This leads to the following definition:

Definition 3 (Neutral term, terminally stuck)**.** *A applicative term $u \in \mathsf{App}$ is* terminally stuck *if $u\,\vec{e}$ is not a redex for all eliminations $\vec{e}$. A term r is* neutral*, written $r \in \mathsf{Ne}$, if it is a redex or terminally stuck.*

As Girard's, our refined notion of neutrality includes redexes, variables, and is closed under eliminations. Further, if $r \in \mathsf{Ne}$ then any reduction in $r\,e$ is either a reduction in r or in e. A reducibility candidate $\mathcal{A}$ is never empty since $\mathsf{Var} \subseteq \mathcal{A}$ by virtue of **CR3**.

Closure. For a set $\mathcal{A} \subseteq \mathsf{SN}$ which is closed under reduction let $\overline{\mathcal{A}}$ be the least reducibility candidate $\supseteq \mathcal{A}$. Inductively, $\overline{\mathcal{A}}$ is defined as the closure under neutrals and simulation:

$$\frac{t \in \mathcal{A}}{t \in \overline{\mathcal{A}}} \qquad \frac{t \in \mathsf{Ne} \qquad (t \longrightarrow \_) \subseteq \overline{\mathcal{A}}}{t \in \overline{\mathcal{A}}}$$

$$\frac{t \notin \mathsf{Intro} \qquad (t \longrightarrow \_) \subseteq \overline{\mathcal{A}} \qquad t \rhd \vec{t} \in \overline{\mathcal{A}}}{t \in \overline{\mathcal{A}}}$$

$\mathcal{A} \mapsto \overline{\mathcal{A}}$ is a *closure operation*, i. e., it is *monotone* ($\mathcal{A} \subseteq \mathcal{B}$ implies $\overline{\mathcal{A}} \subseteq \overline{\mathcal{B}}$), *extensive* ($\mathcal{A} \subseteq \overline{\mathcal{A}}$), and *idempotent* ($\overline{\overline{\mathcal{A}}} \subseteq \overline{\mathcal{A}}$). Note that the closure operator never adds introduction terms such as $()$, (t_1, t_2), $c\,t$, or ${}^G t$ to a term set $\mathcal{A}$. Thus, for introductions $v \in \overline{\mathcal{A}}$ we have $v \in \mathcal{A}$ already.

CR is closed under arbitrary intersections and forms, under the inclusion $\subseteq$ order, a complete lattice with greatest element SN and least element $\overline{\emptyset}$.

Semantic types. In the following, let $\mathcal{A}, \mathcal{B} \in \mathsf{CR}$ be candidates, P a proposition, $\mathcal{K}$ some index set and $\mathcal{F} \in \mathcal{K} \to \mathsf{CR}$ a family of reducibility candidates. The following operations, except the conditional $P \Rightarrow \mathcal{A}$, construct new candidates from existing ones.

$$
\begin{array}{lcl}
\mathcal{A} \rightarrow \mathcal{B} & = & \{r \in \mathsf{SN} \mid \forall s \in \mathcal{A}.\, r\, s \in \mathcal{B}\} \\
\forall_{\mathcal{K}} \mathcal{F} & = & \{r \in \mathsf{SN} \mid \forall G \in \mathsf{Type}, \mathcal{G} \in \mathcal{K}.\, r\, G \in \mathcal{F}(\mathcal{G})\} \\
P \Rightarrow \mathcal{A} & = & \{r \in \mathsf{Exp} \mid r \in \mathcal{A} \text{ if } P\} \\
\mathbf{1} & = & \overline{\{()\}} \\
\mathcal{A}_1 \times \mathcal{A}_2 & = & \overline{\{(t_1, t_2) \mid t_1 \in \mathcal{A}_1 \text{ and } t_2 \in \mathcal{A}_2\}} \\
\exists_{\mathcal{K}} \mathcal{F} & = & \overline{\{{}^G t \mid G \in \mathsf{Type}, \exists \mathcal{G} \in \mathcal{K}, t \in \mathcal{F}(\mathcal{G})\}}
\end{array}
$$

Note that the condition $r \in \mathsf{SN}$ in the definition of $\mathcal{A} \rightarrow \mathcal{B}$ is redundant, since $x \in \mathcal{A}$ by **CR3** and $r\,x \in \mathsf{SN}$ implies $r \in \mathsf{SN}$. However, in the definition of $\forall_{\mathcal{K}} \mathcal{F}$ it is important since $\mathcal{K}$ could be empty, e. g., $\mathcal{K} = {<}0$. Conditional types are not first-class; $P \Rightarrow \mathcal{A}$ only forms a candidate if P is true, otherwise, it is just a set of expressions.

Lemma 4 (Semantic typing rules)**.** *The following inferences are trivial consequences of the construction of semantic types:*

$$
\frac{r \in \mathcal{A} \rightarrow \mathcal{B} \qquad s \in \mathcal{A}}{r\, s \in \mathcal{B}} \qquad \frac{r \in \forall_{\mathcal{K}} \mathcal{F} \qquad \mathcal{G} \in \mathcal{K}}{r\, G \in \mathcal{F}(\mathcal{G})}
$$

$$
\frac{}{() \in \mathbf{1}} \qquad \frac{t_1 \in \mathcal{A}_1 \qquad t_2 \in \mathcal{A}_2}{(t_1, t_2) \in \mathcal{A}_1 \times \mathcal{A}_2} \qquad \frac{\mathcal{G} \in \mathcal{K} \qquad t \in \mathcal{F}(\mathcal{G})}{{}^G t \in \exists_{\mathcal{K}} \mathcal{F}}
$$

Besides definitions (which we will treat in Section 4.5), rules for constructors and destructors are missing. We will describe semantic (co)inductive types in the next section.

4.2 Ordinals and Fixed-Points

Previous approaches to type-based termination (Hughes et al. 1996; Amadio and Coupet-Grimal 1998; Barthe et al. 2004; Blanqui 2004; Sacchini 2013) have defined approximants of least $\boldsymbol{\mu}^\alpha \mathcal{F}$ and greatest fixed-points $\boldsymbol{\nu}^\alpha \mathcal{F}$ of monotone type constructors $\mathcal{F} \in \mathsf{CR} \xrightarrow{+} \mathsf{CR}$ by conventional induction on ordinal α, distinguishing zero (0), successor $(\alpha + 1)$, and limit ordinals (λ).

$$
\begin{array}{lcl@{\qquad}lcl}
\boldsymbol{\mu}^0 \ \mathcal{F} & = & \overline{\emptyset} & \boldsymbol{\nu}^0 \ \mathcal{F} & = & \mathsf{SN} \\
\boldsymbol{\mu}^{\alpha+1} \mathcal{F} & = & \mathcal{F}(\boldsymbol{\mu}^\alpha \mathcal{F}) & \boldsymbol{\nu}^{\alpha+1} \mathcal{F} & = & \mathcal{F}(\boldsymbol{\nu}^\alpha \mathcal{F}) \\
\boldsymbol{\mu}^\lambda \ \mathcal{F} & = & \overline{\bigcup_{\alpha<\lambda} \boldsymbol{\mu}^\alpha \mathcal{F}} & \boldsymbol{\nu}^\lambda \ \mathcal{F} & = & \bigcap_{\alpha<\lambda} \boldsymbol{\nu}^\alpha \mathcal{F}
\end{array}
$$

In this work, we adopt the approach of Sprenger and Dam (2003) for approximations in μ-calculus and use *well-founded induction* instead, which amounts to construct $\boldsymbol{\mu}^\alpha \mathcal{F}$ by *inflationary iteration* and $\boldsymbol{\nu}^\alpha \mathcal{F}$ by *deflationary iteration*.

$$
\boldsymbol{\mu}^\alpha \mathcal{F} = \overline{\bigcup_{\beta<\alpha} \mathcal{F}(\boldsymbol{\mu}^\beta \mathcal{F})} \qquad \boldsymbol{\nu}^\alpha \mathcal{F} = \bigcap_{\beta<\alpha} \mathcal{F}(\boldsymbol{\nu}^\beta \mathcal{F})
$$

In this definition, $\mathcal{F}$ does not have to be monotone to obtain an ascending chain of approximants in case of $\boldsymbol{\mu}$ and a descending chain for $\boldsymbol{\nu}$. However, if $\mathcal{F}$ is monotone, one can derive above equations as special cases for α being zero, successor, or limit ordinal, *if such a distinction on ordinals exists*. Intuitionistically, this distinction is not valid (Taylor 1996); by building on well-founded induction, we remain within constructive foundations.

Let α, β, γ range over ordinals. We write $\forall_{\beta<\alpha} \mathcal{F}(\beta)$ for $\forall_{<\alpha} \mathcal{F}$ and analogously for $\exists$. Let $\mathcal{S} \in \mathsf{Cons} \rightharpoonup \mathsf{CR} \to \mathsf{CR}$ and $\mathcal{R} \in \mathsf{Proj} \rightharpoonup \mathsf{CR} \to \mathsf{CR}$ where we write the first argument, the constructor c, or the destructor d, resp., as index, thus, $\mathcal{S}_c$ and $\mathcal{R}_d$ resp. We define the αth approximants $\boldsymbol{\mu}^\alpha \mathcal{S}, \boldsymbol{\nu}^\alpha \mathcal{R} \in \mathsf{CR}$ of recursive variant and record type as follows.

$$
\begin{array}{lcl}
\boldsymbol{\mu}^\alpha \mathcal{S} & = & \overline{\{c\,t \mid c \in \mathsf{dom}(\mathcal{S}) \text{ and } t \in \exists_{\beta<\alpha} \mathcal{S}_c(\boldsymbol{\mu}^\beta \mathcal{S})\}} \\
\boldsymbol{\nu}^\alpha \mathcal{R} & = & \{r \in \mathsf{SN} \mid \forall d \in \mathsf{dom}(\mathcal{R}).\, r.d \in \forall_{\beta<\alpha} \mathcal{R}_d(\boldsymbol{\nu}^\beta \mathcal{R})\}
\end{array}
$$

Since $\exists_{<\alpha} \mathcal{F}$ is monotonic in α for any $\mathcal{F}$, so is $\boldsymbol{\mu}^\alpha \mathcal{S}$. Dually $\forall_{<\alpha} \mathcal{F}$ and $\boldsymbol{\nu}^\alpha \mathcal{R}$ are antitonic in α. We obtain chains:

$$
\begin{array}{l}
\overline{\emptyset} \ = \boldsymbol{\mu}^0 \mathcal{S} \subseteq \boldsymbol{\mu}^1 \mathcal{S} \subseteq \ldots \subseteq \boldsymbol{\mu}^\gamma \mathcal{S} \subseteq \boldsymbol{\mu}^{\gamma+1} \mathcal{S} \subseteq \ldots \\
\mathsf{SN} = \boldsymbol{\nu}^0 \mathcal{R} \supseteq \boldsymbol{\nu}^1 \mathcal{R} \supseteq \ldots \supseteq \boldsymbol{\nu}^\gamma \mathcal{R} \supseteq \boldsymbol{\nu}^{\gamma+1} \mathcal{R} \supseteq \ldots
\end{array}
$$

If $\boldsymbol{\mu}^\alpha \mathcal{S} = \boldsymbol{\mu}^\gamma \mathcal{S}$ for some $\alpha > \gamma$ then $\boldsymbol{\mu}^\beta \mathcal{S} = \boldsymbol{\mu}^\gamma \mathcal{S}$ for all $\beta > \gamma$ and we say that the chain has become *stationary* at γ. Since the set Exp of expressions is countable and all elements of these chains are subsets of Exp, the chains must become stationary latest at the first uncountable ordinal Ω. We call the ordinal at which all such chains of our language are stationary the *closure ordinal* and denote it by $\boldsymbol{\infty}$.

Since it does not make sense to inspect chains beyond the closure ordinal, we introduce *bound normalization*

$$
\alpha^\uparrow = \begin{cases} \boldsymbol{\infty} + 1 & \text{if } \alpha \geq \boldsymbol{\infty}, \\ \alpha & \text{otherwise.} \end{cases}
$$

Note that $\boldsymbol{\mu}^\alpha \mathcal{S} = \boldsymbol{\mu}^{\alpha^\uparrow} \mathcal{S}$ and $\boldsymbol{\nu}^\alpha \mathcal{R} = \boldsymbol{\nu}^{\alpha^\uparrow} \mathcal{R}$. In the following we will talk about ordinals that are as big as $\boldsymbol{\infty} + n$ for finite n, but not bigger ones, so all ordinals will be in $\mathsf{O} = \{\alpha \mid \alpha < \boldsymbol{\infty} + \omega\}$, a set closed under successor. As size index to a least or greatest fixed point, only the ordinals in $\mathsf{Size} = \{\alpha \mid \alpha \leq \boldsymbol{\infty}\}$ are interesting. Thus, if no bound for an ordinal β is given, we assume $\beta \in \mathsf{Size}$, for instance, we write $\exists_\beta \mathcal{F}(\beta)$ instead of $\exists_{\beta \in \mathsf{Size}} \mathcal{F}(\beta)$ or $\exists_{\mathsf{Size}} \mathcal{F}$.

The stationary point $\boldsymbol{\mu}^{\boldsymbol{\infty}} \mathcal{S}$ is a pre-fixed point in the sense that $t \in \mathcal{S}_c(\boldsymbol{\mu}^{\boldsymbol{\infty}} \mathcal{S})$ implies $c^{\,\infty} t \in \boldsymbol{\mu}^{\boldsymbol{\infty}+1} \mathcal{S} = \boldsymbol{\mu}^{\boldsymbol{\infty}} \mathcal{S}$. Dually, $\boldsymbol{\nu}^{\boldsymbol{\infty}} \mathcal{R}$ is a post-fixed point as $r \in \boldsymbol{\nu}^{\boldsymbol{\infty}} \mathcal{R} = \boldsymbol{\nu}^{\boldsymbol{\infty}+1} \mathcal{R}$ implies $r.d\,\infty \in \mathcal{R}_d(\boldsymbol{\nu}^{\boldsymbol{\infty}} \mathcal{R})$. Note that we do not require $\mathcal{R}$ or $\mathcal{S}$ to be monotone for the implications to hold in these directions. Yet we do if we want $\boldsymbol{\mu}^{\boldsymbol{\infty}} \mathcal{S}$ and $\boldsymbol{\nu}^{\boldsymbol{\infty}} \mathcal{R}$ to be fixed-points.

Lemma 5 (Fixed-points)**.** *If $\mathcal{S}_c, \mathcal{R}_d$ be monotone for all $c \in \mathsf{dom}(\mathcal{S})$ and $d \in \mathsf{dom}(\mathcal{R})$, then*

1. $\boldsymbol{\mu}^{\boldsymbol{\infty}} \mathcal{S} = \overline{\{c^{\,b} t \mid c \in \mathsf{dom}(\mathcal{S}), b \in \mathsf{Type}, t \in \mathcal{S}_c(\boldsymbol{\mu}^{\boldsymbol{\infty}} \mathcal{S})\}}$*, and*
2. $\boldsymbol{\nu}^{\boldsymbol{\infty}} \mathcal{R} = \{r \mid \forall d \in \mathsf{dom}(\mathcal{R}), b \in \mathsf{Type}.\ r.d\ b \in \mathcal{R}_d(\boldsymbol{\nu}^{\boldsymbol{\infty}} \mathcal{R})\}$.

Proof. For 1, it is sufficient to show $\subseteq$, meaning that $\boldsymbol{\mu}^{\boldsymbol{\infty}} \mathcal{S}$ is a post-fixed point. Note that by definition

$$
\boldsymbol{\mu}^{\boldsymbol{\infty}} \mathcal{S} = \overline{\bigcup_{\beta<\boldsymbol{\infty}} \{c^{\,b} t \mid c \in \mathsf{dom}(\mathcal{S}), b \in \mathsf{Type}, t \in \mathcal{S}_c(\boldsymbol{\mu}^\beta \mathcal{S})\}},
$$

so we conclude by monotonicity of $\mathcal{S}_c$ and the closure operator, using $\boldsymbol{\mu}^\beta \mathcal{S} \subseteq \boldsymbol{\mu}^{\boldsymbol{\infty}} \mathcal{S}$. For 2, it is sufficient to show that $\boldsymbol{\nu}^{\boldsymbol{\infty}} \mathcal{R}$ is a pre-fixed point. So, if $r.d\ b \in \mathcal{R}_d(\boldsymbol{\nu}^{\boldsymbol{\infty}} \mathcal{R})$ for all $d \in \mathsf{dom}(\mathcal{R})$ and $b \in \mathsf{Type}$, then $r \in \boldsymbol{\nu}^{\boldsymbol{\infty}} \mathcal{R}$. It is sufficient to show $r.d\ b \in \mathcal{R}_d(\boldsymbol{\nu}^\beta \mathcal{R})$ for all $\beta < \boldsymbol{\infty}$, and this follows from $\boldsymbol{\nu}^{\boldsymbol{\infty}} \mathcal{R} \subseteq \boldsymbol{\nu}^\beta \mathcal{R}$ by monotonicity of $\mathcal{R}_d$. ∎

4.3 Kinds

Higher kinds are interpreted as π-variant set-theoretical function spaces $\mathcal{K}_1 \xrightarrow{\pi} \mathcal{K}_2$ over the base kinds CR and ${<}\alpha$. For ρ a *size valuation* mapping size variables to ordinals, *kind interpretation* $[\![\kappa]\!]_\rho$ is defined in the obvious way.

A *semantic kinding context* $\mathcal{D}$ maps type variables X to semantic kinds $\mathcal{K}$. Semantic kinding contexts classify *type environments* ρ mapping type variables to semantic types or type constructors; we have $\rho \in \mathcal{D}$ if $\rho(X) \in \mathcal{D}(X)$ for all $X \in \mathsf{dom}(\mathcal{D})$. Since kinds in $\mathcal{D}$ can depend on size variables declared earlier in $\mathcal{D}$, semantic kinding contexts are dependent. Given $\mathcal{D}$ and a family $\mathcal{D}'(\rho \in \mathcal{D})$, the dependent concatenation of $\mathcal{D}$ and $\mathcal{D}'$ is written $\Sigma_{\mathcal{D}}\mathcal{D}'$.

4.4 Type constructors

Type constructors of higher kind are interpreted as operators on semantic types. For ρ a *type environment* mapping type variables to semantic types or type constructors, *type interpretation* $[\![F]\!]_\rho$ maps the syntactic type constructors to the corresponding semantic ones.

Kind and type interpretation model the kind- and type-level judgements in the usual way. For lack of space, we cannot provide more detail here, see the extended version of this paper instead.

4.5 Patterns, copatterns, λ-abstractions

In this section, we explain patterns and copatterns by developing semantic notions of pattern and pattern spine typing. These provide us with semantic conditions when a definition $\lambda\vec{D}$ inhabits a semantic type $\mathcal{A}$. As a consequence, we can prove soundness of syntactic pattern, pattern spine, and expression typing.

Semantic typing contexts and semantic pattern typing. A semantic typing context $\mathcal{E} \in \mathsf{CXT}(\cdot)$ ($\mathcal{E}$ for typing *environment*) is a finite map from term variables to semantic types, so $\mathcal{E} \in \mathsf{Var} \rightharpoonup \mathsf{CR}$. We write $\cdot$ for the empty semantic typing context, $x{:}\mathcal{A}$ for the singleton and $\mathcal{E}, \mathcal{E}'$ for the disjoint union. *Semantic substitution typing* $\sigma \in \mathcal{E}$ is defined as $\sigma(x) \in \mathcal{E}(x)$ for all $x \in \mathsf{dom}(\mathcal{E})$.

A parameterized semantic typing context $\mathcal{E} \in \mathsf{CXT}(\mathcal{D})$ is a family $\mathcal{E}(\rho)$ of semantic typing contexts indexed by semantic type substitutions ρ that belong to a semantic kinding context $\mathcal{D}$. Each instance $\mathcal{E}(\rho)$ is a partial function from variables to semantic types. We overload the notation for non-parameterized semantic typing contexts by setting $\cdot(\rho) = \cdot$ and $(x{:}\mathcal{A})(\rho) = x{:}\mathcal{A}(\rho)$ and $(\mathcal{E}, \mathcal{E}')(\rho) = \mathcal{E}(\rho), \mathcal{E}'(\rho)$ with $\mathsf{dom}(\mathcal{E}(\rho)) \cap \mathsf{dom}(\mathcal{E}'(\rho)) = \emptyset$.

For two differently parameterized semantic typing contexts $\mathcal{E}_1 \in \mathsf{CXT}(\mathcal{D}_1)$ and $\mathcal{E}_2 \in \mathsf{CXT}(\mathcal{D}_2)$ we let their disjoint union $\mathcal{E}_1 * \mathcal{E}_2 \in \mathsf{CXT}(\mathcal{D}_1, \mathcal{D}_2)$ be defined by $(\mathcal{E}_1 * \mathcal{E}_2)(\rho_1 \in \mathcal{D}_1, \rho_2 \in \mathcal{D}_2) = (\mathcal{E}_1(\rho_1), \mathcal{E}_2(\rho_2))$. Further, if $\mathcal{E} \in \mathsf{CXT}(\Sigma_{\mathcal{D}}\mathcal{D}')$ and $\rho \in \mathcal{D}$ we let the partial application $\mathcal{E}(\rho, \_) \in \mathsf{CXT}(\mathcal{D}'(\rho))$ be defined by $\mathcal{E}(\rho, \_)(\rho') = \mathcal{E}(\rho, \rho')$.

If $\mathcal{C}(\mathcal{G})(\rho)$ is a type parameterized by another type $\mathcal{G}$ and a type substitution ρ, we let $\mathcal{C}X$ be defined by $(\mathcal{C}X)(\rho) = \mathcal{C}(\rho(X))(\rho \setminus X)$. In particular, $(\mathcal{C}X)(\mathcal{G}/X, \rho) = \mathcal{C}(\mathcal{G})(\rho)$. The notations $\mathcal{D}X$ and $\mathcal{E}X$ are defined analogously.

A pattern p is semantically of type $\mathcal{A}$ in context $\mathcal{E}$ if it acts as a bidirectional (invertible) map from $\mathcal{E}$ to $\mathcal{A}$, i. e., $p\sigma \in \mathcal{A}$ for all $\sigma \in \mathcal{E}$, and, for any substitution σ with $p\sigma \in \mathcal{A}$ we have $\sigma \in \mathcal{E}$. Extending this to type substitutions we define *semantic pattern typing* by

$$\boxed{\mathcal{A} \,/\, p \searrow \mathcal{D}; \mathcal{E}} \;:\Longleftrightarrow\; \forall \tau, \sigma.\ (\exists \rho \in \mathcal{D}.\ \sigma \in \mathcal{E}(\rho)) \iff p\tau\sigma \in \mathcal{A}.$$

Here, and in the following, τ denotes a syntactic type substitution. Note that it is unconstrained, it needs not bear a relationship with the semantic type substitution ρ.

One could have expected that semantic pattern typing implies that p matches any introduction term $v \in \mathcal{A}$. But since we are not interested in pattern coverage, but merely strong normalization, we do not require this strong guarantee.[6]

Lemma 6 (Semantic pattern typing)**.** *The following implications, written as rules, hold.*

$$\frac{}{\mathcal{A} \,/\, x \searrow \cdot; (x{:}\mathcal{A})} \qquad \frac{}{\mathbf{1} \,/\, () \searrow \cdot; \cdot}$$

$$\frac{\mathcal{A}_1 \,/\, p_1 \searrow \mathcal{D}_1; \mathcal{E}_1 \qquad \mathcal{A}_2 \,/\, p_2 \searrow \mathcal{D}_2; \mathcal{E}_2}{\mathcal{A}_1 \times \mathcal{A}_2 \,/\, (p_1, p_2) \searrow \mathcal{D}_1, \mathcal{D}_2; \mathcal{E}_1 * \mathcal{E}_2}$$

$$\frac{\exists_{\beta < \alpha^\uparrow} \mathcal{S}_c(\boldsymbol{\mu}^\beta \mathcal{S}) \,/\, p \searrow \mathcal{D}; \mathcal{E}}{\boldsymbol{\mu}^\alpha \mathcal{S} \,/\, c\, p \searrow \mathcal{D}; \mathcal{E}}$$

$$\frac{\mathcal{F}(\mathcal{G}) \,/\, p \searrow \mathcal{D}(\mathcal{G}); \mathcal{E}(\mathcal{G}) \text{ for all } \mathcal{G} \in \mathcal{K}}{\exists_{\mathcal{K}} \mathcal{F} \,/\, {}^{X}p \searrow \Sigma_{X:\mathcal{K}}\mathcal{D}; \mathcal{E}X}$$

Theorem 7 (Soundness of pattern typing)**.** *Let* $\vdash \Delta_0, \Delta$ *and* $\Delta_0, \Delta \vdash \Gamma$. *If* $\Delta; \Gamma \vdash_{\Delta_0} p \Leftarrow A$ *and* $\rho_0 \in [\![\Delta_0]\!]$ *then* $[\![A]\!]_{\rho_0} \,/\, p \searrow [\![\Delta]\!]_{\rho_0}; [\![\Gamma]\!]_{(\rho_0, \_)}$.

Proof. By induction on $\Delta; \Gamma \vdash_{\Delta_0} p \Leftarrow A$ using the inferences of Lemma 6. □

Semantic typing in context. Given a parameterized semantic type $\mathcal{C} \in \mathcal{D}' \to \mathsf{CR}$ we define weakening $\mathsf{W}_{\mathcal{D}}\mathcal{C} \in (\mathcal{D}, \mathcal{D}') \to \mathsf{CR}$ of $\mathcal{C}$ by semantic kinding context $\mathcal{D}$ as $(\mathsf{W}_{\mathcal{D}}\mathcal{C})(\rho \in \mathcal{D}, \rho') = C(\rho')$. Given a semantic type family $\mathcal{C} \in (\mathcal{D}, \mathcal{D}') \to \mathsf{CR}$ and a semantic type substitution $\rho \in \mathcal{D}$, we let the partial application $\mathcal{C}(\rho, \_) \in \mathcal{D}' \to \mathsf{CR}$ be defined by $\mathcal{C}(\rho, \_)(\rho') = \mathcal{C}(\rho, \rho')$. Semantic typing under a context is defined by

$$\boxed{\mathcal{D}; \mathcal{E} \vdash t \in \mathcal{C}} \;:\Longleftrightarrow\; \forall \rho \in \mathcal{D}, \sigma \in \mathcal{E}(\rho), \tau.\ \ t\tau\sigma \in \mathcal{C}(\rho)$$

Let P be a proposition depending on the pattern variables and pattern type variables of a copattern spine $\vec{q}$. We define the following shorthand for the replacement of the pattern variables by expressions obtained from matching $\vec{q}$ against an elimination list $\vec{e}$:

$$\boxed{P[\vec{e}/\vec{q}]} \;:\Longleftrightarrow\; \exists \tau, \sigma.\ \ \vec{e} \,/\, \vec{q} \searrow \tau; \sigma \ \wedge\ P\tau\sigma$$

Semantic pattern spines. A pattern spine $\vec{q}$ has to be understood by its purpose, to serve as the lhs of a definition. Semantically, q eliminates type $\mathcal{A}$ into $\mathcal{C}$ at contexts $\mathcal{D}; \mathcal{E}$ if any definition $\lambda\{\vec{q} \to t\}$ that can be formed with $\vec{q}$ is in $\mathcal{A}$ as long as the rhs t is in $\mathcal{C}$ under contexts $\mathcal{D}; \mathcal{E}$. We further generalize this to partially applied definitions $\lambda\{\vec{q}'\, \vec{q} \to t\}\vec{e}$ where $\vec{e}$ matches $\vec{q}'$. We let

$$\boxed{\mathcal{A} \mid \vec{q} \searrow \mathcal{D}; \mathcal{E}; \mathcal{C}} \;:\Longleftrightarrow\; \forall t, \vec{e} \in \mathsf{SN}. \forall \vec{q}'.\ \ \mathcal{D}; \mathcal{E} \vdash t[\vec{e}/\vec{q}'] \in \mathcal{C} \implies \lambda\{\vec{q}'\vec{q} \to t\}\vec{e} \in \mathcal{A}.$$

Lemma 8 (Semantic clause typing)**.** *The following implication holds:*

$$\frac{\mathcal{A} \mid \vec{q} \searrow \mathcal{D}; \mathcal{E}; \mathcal{C} \qquad \mathcal{D}; \mathcal{E} \vdash t \in \mathcal{C} \qquad \rho \in \mathcal{D}}{\lambda\{\vec{q} \to t\} \in \mathcal{A}}$$

Proof. With $\sigma_{\mathsf{id}} \in \mathcal{E}(\rho)$ we have $t = t\sigma_{\mathsf{id}} \in \mathcal{C}(\rho) \subseteq \mathsf{SN}$. The rest follows by definition of semantic pattern spine typing with empty $\vec{e}$ and empty $\vec{q}'$. Note that we cannot proceed if $\mathcal{D}$ is inconsistent. □

Lemma 9 (Semantic pattern spine typing)**.** *The following implications hold.*

$$\frac{}{\mathcal{A} \mid \cdot \searrow \cdot; \cdot; \mathcal{A}} \qquad \frac{\mathcal{A}_1 \,/\, p \searrow \mathcal{D}_1; \mathcal{E}_1 \qquad \mathcal{A}_2 \mid \vec{q} \searrow \mathcal{D}_2; \mathcal{E}_2; \mathcal{C}}{\mathcal{A}_1 \to \mathcal{A}_2 \mid p\, \vec{q} \searrow \mathcal{D}_1, \mathcal{D}_2; \mathcal{E}_1 * \mathcal{E}_2; \mathsf{W}_{\mathcal{D}_1}\mathcal{C}}$$

[6] On the contrary, we can live with junk introductions in our semantic types. For instance, it would not endanger normalization to throw the empty tuple into each semantic type.

$$\frac{\boldsymbol{\forall}_{\beta < \alpha^\uparrow} \mathcal{R}_d\,(\boldsymbol{\nu}^\beta \mathcal{R}) \mid \vec{q} \searrow \mathcal{D}; \mathcal{E}; \mathcal{C}}{\boldsymbol{\nu}^\alpha \mathcal{R} \mid .d\,\vec{q} \searrow \mathcal{D}; \mathcal{E}; \mathcal{C}}$$

$$\frac{\forall \mathcal{G} \in \mathcal{K}.\ \ \mathcal{F}(\mathcal{G}) \mid \vec{q} \searrow \mathcal{D}(\mathcal{G}); \mathcal{E}(\mathcal{G}); \mathcal{C}(\mathcal{G})}{\boldsymbol{\forall}_{\mathcal{K}} \mathcal{F} \mid X\,\vec{q} \searrow \Sigma_{X:\mathcal{K}} \mathcal{D}; \mathcal{E}X; \mathcal{C}X}$$

Theorem 10 (Soundness of pattern spine typing)**.** *Let* $\vdash \Delta_0, \Delta$ *and* $\Delta_0, \Delta \vdash \Gamma$. *If* $\Delta; \Gamma \mid A \vdash_{\Delta_0} \vec{q} \rightrightarrows C$ *and* $\rho_0 \in [\![\Delta_0]\!]$ *then* $[\![A]\!]_{\rho_0} \mid \vec{q} \searrow [\![\Delta]\!]_{\rho_0}; [\![\Gamma]\!]_{(\rho_0,\text{-})}; [\![C]\!]_{(\rho_0,\text{-})}$.

Proof. By induction on $\Delta; \Gamma \mid A \vdash_{\Delta_0} \vec{q} \rightrightarrows C$ using Lem. 9. ☐

Semantic declaration and signature well-formedness. Having understood definitons by clauses $\lambda\vec{D}$ we can now show that any well-typed term inhabits its corresponding semantic type. For function symbols f, we simply assume it, by postulating a sematically well-formed signature Σ. We define $\boxed{\models \delta}$ and $\boxed{\models \Sigma}$ by

$$\begin{array}{lll} \models (f : A = \vec{D}) & :\Longleftrightarrow & f \in [\![A]\!] \\ \models \Sigma & :\Longleftrightarrow & \forall \delta \in \Sigma.\ \models \delta. \end{array}$$

Theorem 11 (Soundness of expression typing)**.** *Assume* $\models \Sigma$. *Let* $\vdash \Delta$ *and* $\Delta \vdash \Gamma$ *and* $\Delta \vdash C$ *and* $\mathcal{D} = [\![\Delta]\!]$ *and* $\mathcal{E}(\rho) = [\![\Gamma]\!]_\rho$ *and* $\mathcal{C}(\rho) = [\![C]\!]_\rho$.

1. *If* $\Delta; \Gamma \vdash r \rightrightarrows C$ *in* Σ *then* $\mathcal{D}; \mathcal{E} \vdash r \in \mathcal{C}$.
2. *If* $\Delta; \Gamma \vdash t \leftleftarrows C$ *in* Σ *then* $\mathcal{D}; \mathcal{E} \vdash t \in \mathcal{C}$.
3. *If* $\Delta; \Gamma \vdash \vec{D} \leftleftarrows C$ *in* Σ *then* $\mathcal{D}; \mathcal{E} \vdash \lambda\vec{D} \in \mathcal{C}$.

What remains to be proven is that well-typed programs yield, after measure erasure, semantically well-formed signatures. This is shown mutual block by mutual block using a lexicographic induction on ordinals as given by the termination measure assigned to each block. A formal description of program typing and its soundness proof has to be delegated to the long version of this paper due to lack of space.

5. Conclusion

Our work provides a uniform type-based approach to proving termination of (co)inductive definitions. It is centered around patterns and copatterns which allow us to reason about both finite and infinite data by well-founded induction. Proving strong normalization for this language is a significant step towards understanding well-founded corecursion in terms of the depth of observation we can safely make.

As a next step, we plan to extend our work to full dependently typed systems to allow coinductive definitions to be defined and reasoned with by observations. This will put coinduction in these systems on a robust foundation. We have already implemented size-based type checking for patterns and copatterns in MiniAgda (Abel 2012) which gives us confidence in the approach.

References

A. Abel. Semi-continuous sized types and termination. *Logical Meth. in Comput. Sci.*, 4(2:3):1–33, 2008a. CSL'06 special issue.

A. Abel. Polarized subtyping for sized types. *Math. Struct. in Comput. Sci.*, 18:797–822, 2008b. Special issue on subtyping, edited by Healfdene Goguen and Adriana Compagnoni.

A. Abel. Type-based termination, inflationary fixed-points, and mixed inductive-coinductive types. *Electr. Proc. in Theor. Comp. Sci.*, 77:1–11, 2012. Proceedings of FICS 2012.

A. Abel and B. Pientka. Wellfounded recursion with copatterns: A unified approach to termination and productivity. Extended version, 2013. URL http://www.tcs.ifi.lmu.de/~abel/icfp13-long.pdf.

A. Abel, B. Pientka, D. Thibodeau, and A. Setzer. Copatterns: Programming infinite structures by observations. In *Proc. of the 40th ACM Symp. on Principles of Programming Languages, POPL 2013*, pages 27–38. ACM Press, 2013.

T. Altenkirch and N. A. Danielsson. Termination checking in the presence of nested inductive and coinductive types. Short note supporting a talk given at PAR 2010, Workshop on Partiality and Recursion in Interactive Theorem Provers, FLoC 2010, 2010. URL http://www.cse.chalmers.se/~nad/publications/altenkirch-danielsson-par2010.pdf.

R. M. Amadio and S. Coupet-Grimal. Analysis of a guard condition in type theory (extended abstract). In *Proc. of the 1st Int. Conf. on Foundations of Software Science and Computation Structure, FoSSaCS'98*, volume 1378 of *Lect. Notes in Comput. Sci.*, pages 48–62. Springer, 1998.

G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. *Math. Struct. in Comput. Sci.*, 14 (1):97–141, 2004.

G. Barthe, B. Grégoire, and C. Riba. Type-based termination with sized products. In *Computer Science Logic, 22nd Int. Wksh., CSL 2008, 17th Annual Conf. of the EACSL*, volume 5213 of *Lect. Notes in Comput. Sci.*, pages 493–507. Springer, 2008.

F. Blanqui. A type-based termination criterion for dependently-typed higher-order rewrite systems. In *Rewriting Techniques and Applications (RTA 2004), Aachen, Germany*, volume 3091 of *Lect. Notes in Comput. Sci.*, pages 24–39. Springer, 2004.

F. Blanqui and C. Riba. Combining typing and size constraints for checking the termination of higher-order conditional rewrite systems. In *Proc. of the 13th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR 2006*, volume 4246 of *Lect. Notes in Comput. Sci.*, pages 105–119. Springer, 2006.

N. Ghani, P. Hancock, and D. Pattinson. Representations of stream processors using nested fixed points. *Logical Meth. in Comput. Sci.*, 5(3), 2009.

E. Giménez. *Un Calcul de Constructions Infinies et son application a la vérification de systèmes communicants*. PhD thesis, Ecole Normale Supérieure de Lyon, 1996. Thèse d'université.

J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoret. Comput. Sci.* Cambridge University Press, 1989.

J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *Proc. of the 23rd ACM Symp. on Principles of Programming Languages, POPL'96*, pages 410–423, 1996.

INRIA. *The Coq Proof Assistant Reference Manual*. INRIA, version 8.4 edition, 2012. URL http://coq.inria.fr/.

U. Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Dept of Comput. Sci. and Engrg., Chalmers, Göteborg, Sweden, 2007.

J. L. Sacchini. Type-based productivity of stream definitions in the calculus of constructions. In *Logics in Computer Science (LICS 2013), June 25-28, 2013, New Orleans*, 2013.

B. A. Sijtsma. On the productivity of recursive list definitions. *ACM Trans. Prog. Lang. Syst.*, 11(4):633–649, 1989.

C. Sprenger and M. Dam. On the structure of inductive reasoning: Circular and tree-shaped proofs in the μ-calculus. In *Proc. of the 6th Int. Conf. on Foundations of Software Science and Computational Structures, FoSSaCS 2003*, volume 2620 of *Lect. Notes in Comput. Sci.*, pages 425–440. Springer, 2003.

M. Steffen. *Polarized Higher-Order Subtyping*. PhD thesis, Technische Fakultät, Universität Erlangen, 1998.

P. Taylor. Intuitionistic sets and ordinals. *J. Symb. Logic*, 61(3):705–744, 1996.

J. Vouillon and P.-A. Melliès. Semantic types: A fresh look at the ideal model for types. In *Proc. of the 31st ACM Symp. on Principles of Programming Languages, POPL 2004*, pages 52–63. ACM Press, 2004.

H. Xi. Dependent types for program termination verification. *J. Higher-Order and Symb. Comput.*, 15(1):91–131, 2002.