Termination and Productivity Checking with Continuous Types

Andreas Abel *

Department of Computer Science, University of Munich
Oettingenstr. 67, 80538 München, Germany
abel@informatik.uni-muenchen.de

Abstract. We analyze the interpretation of inductive and coinductive types as sets of strongly normalizing terms and isolate classes of types with certain continuity properties. Our result enables us to relax some side conditions on the shape of recursive definitions which are accepted by the type-based termination calculus of Barthe, Frade, Giménez, Pinto and Uustalu, thus enlarging its expressivity.

1 Introduction and Related Work

Interactive theorem provers like Coq [13], LEGO [20] and Twelf [18] support proofs by induction on finite-depth (inductive) structures (like natural numbers, lists, infinitely branching trees) and infinite-depth (coinductive) structures (like streams, processes, trees with infinite paths) in the form of recursive programs. However, these programs constitute valid proofs only if they denote *total* functions. In the last decade, considerable effort has been put on the development of means to define total functions in the type theories of the abovementioned theorem provers.

The first solution was to restrict programs to specific recursion schemes like iteration or primitive recursion (like `natrec`). Since these schemes come with complicated reduction behavior, they were inconvenient to use and so the search started how to incorporate the simpler *general recursion* (fix resp. letrec) known from functional programming. Programs using fix are in general not *terminating* (resp. *productive*, for the case of infinite structures), hence could denote partial or undefined functions. Thus, the use of fix has to be restricted in some manner.

Static analysis of the program. The first approach is to check the *code* of a recursive program to make sure some argument is decreasing in each recursive call (cf. Abel and Altenkirch [1, 4], Pientka [19], Lee, Jones and Ben-Amram [14]). For programs defining infinite objects like streams, the recursive calls need to be *guarded*. Guardedness checks have been devised e.g. by Coquand [8], Telford and Turner [22] and Giménez [9].

* Research supported by the *Graduiertenkolleg Logik in der Informatik* (PhD Program Logic in Computer Science) of the Deutsche Forschungsgemeinschaft (DFG). The author thanks Martin Hofmann and Ralph Matthes for helpful discussions.

```
pivot : int → ∀Y ≈ int list . Y → Y × Y
pivot a ([])^Y¹          = ([])^Y¹, ([])^Y¹)
pivot a ((x :: xs^Y)^Y¹  = let (l^Y, r^Y) = pivot a xs in
                             if a > x then ((x :: l)^Y¹, r^Y≤Y¹) else (l^Y≤Y¹, (x :: r)^Y¹)

qsapp : ∀Y ≈ int list . Y → int list → int list
qsapp ([])^Y¹ ys         = ys
qsapp ((x :: xs^Y)^Y¹ ys = let (l^Y, r^Y) = pivot x xs in qsapp l^Y (x :: qsapp r^Y ys)

quicksort : int list → int list
quicksort l = qsapp l []
```

Fig. 1. Example: Quick-sort.

For programs like quicksort (cf. Fig. 1, ignore type annotations for now), input arguments are transformed *via another function* (here: pivot), before fed into the recursive call. Hence, analyses need to infer the size relation between input and output of a function, or at least detect non-size increasing functions like pivot. Well-designed static analyses [19, 14, 15, 22] support some size checking for first-order programs, but fail for higher-order programs. These analyses usually have some other drawbacks: they are quite sophisticated and thus hard to comprehend and to trust, and they are sensitive to little changes in the program code: a program which passed a termination check might not pass any longer if a redex is introduced (cf. discussion in Barthe *et al.* [7]). Furthermore, they all fail for non-strictly positive datatypes.

Type-based analysis tries to address these problems by assigning a specific *type* to a total program instead of analyzing the code: Types are preserved under reduction and scale to higher-order functions and non-strictly positive data. Type theory has a long tradition in the programming language and theorem proving community and people are trained to understand typing rules. So a type-based approach might be easier to comprehend and easier trusted.

The first type-based approach to termination is attributed to Mendler [16]. He used universally quantified type variables to restrict recursive calls. His ideas have been further developed by many [6, 21]; we will only consider approaches here which also detect non-size increasing functions.

Type-based termination in a nutshell. Inductive structures can be viewed as trees and thus classified by their *height*. First-order data structures like lists and binary trees have a finite height $< \omega$ which can be denoted by a natural number. Proof theory deals also with higher-order data structures like infinitely branching trees whose height can only be denoted by an infinite ordinal $\alpha \geq \omega$ below the least uncountable ordinal Ω. The dual concept to height is *definedness* or *observability* upto depth α and applies to coinductive structures.

In type-based termination, total functions are defined by induction on the height/definedness of structures. Let $\nabla X\sigma$ define a inductive ($\nabla \equiv \mu$) or coinductive ($\nabla \equiv \nu$) datatype. Then $\rho \approx \nabla X\sigma$ denotes a subtype or *approximation* ρ of $\nabla X\sigma$ which contains elements below a certain height/definedness α. Following Barthe *et al.* [7] we will refer to such elements as elements at *stage* α. Next, ρ^n denotes the type of elements at stage $\alpha + n$. Types involving bounded quantification $\forall Y{\approx}\nabla X\sigma.\, \tau(Y)$ can be instantiated at any approximation $\rho \approx \nabla X\sigma$ of a datatype to obtain the type $\tau(\rho)$. Recursive functions of result type τ involving $\nabla X\sigma$ are introduced by the rule

$$\frac{Y{\approx}\nabla X\sigma,\ g{:}\tau(Y) \vdash M : \tau(Y^1)}{\mathsf{fix}^{\nabla} g.M : \forall Y{\approx}\nabla X\sigma.\, \tau(Y)}.$$

The premise incarnates the step case of induction, taking the function from stage α to stage $\alpha+1$. The base case $\alpha = 0$ is handled by side conditions on the type τ which ensures that τ at stage 0 is interpreted as the whole universe of programs. This holds trivially for the class

$$\begin{array}{ll} \tau(Y) \equiv Y \to \tau'(Y) & \text{if } \nabla = \mu, \\ \tau(Y) \equiv \tau'(Y) \to Y & \text{if } \nabla = \nu. \end{array}$$

The fact that τ' may depend on Y enables the type system to tract stage dependencies between input and output of a function like $\mathsf{pivot} : \mathsf{int} \to \forall Y \approx \mathsf{int\ list}.\, Y \to Y \times Y$. The type of pivot guarantees that the length of both output lists is bounded by the length of the input list. This feature is very valuable for defining Euclidian division, merge sort, the stream of Fibonacci numbers (see below) etc.

For higher-order datatypes $\nabla X\sigma$ we have to continue induction transfinitely and also handle the limit case $\alpha = \lambda$. More precisely, we have to show that if the recursive program $\mathsf{fix}^{\nabla} g.M$ inhabits τ at stages $\beta < \lambda$ then it inhabits τ also at the limit stage λ. For types in the special class above this is always the case if $\tau'(Y)$ is monotone in Y. But this is not a necessary condition and there *are* interesting types which do not fall into the above class. E.g., let $\mathsf{Stream} = \nu X.\, \mathsf{Nat}{\times}X$ denote streams of natural numbers, $\mathsf{fold} : \forall Y{\approx}\mathsf{Stream}.\, \mathsf{Nat}{\times}Y \to Y^1$ the injection into Stream and

$$\begin{array}{l} \mathsf{sum} : \forall Y{\approx}\mathsf{Stream}.\, Y \to Y \to Y \\ \mathsf{sum}\ (x :: xs)\ (y :: ys) = x + y :: \mathsf{sum}\, xs\, ys \\ \mathsf{fib}' \ : \forall Y{\approx}\mathsf{Stream}.\, \mathsf{Nat} \times Y \\ \mathsf{fib}' \ = (0,\ 1 :: \mathsf{sum}\ (\mathsf{fold}\, \mathsf{fib}')\ (\mathsf{snd}\, \mathsf{fib}')) \end{array}$$

from which we can obtain the stream of Fibonacci numbers $\mathsf{fib} = \mathsf{fold}\, \mathsf{fib}'$. In the type system of Barthe *el al.* [7], the type of fib' is not accepted and it is not clear how to define the Fibonacci stream in a similarly elegant way. As we will see in Sect. 4, both definitions are perfectly legal in our system.

Approach	Flavor	Polym.	Norm.	Datatypes	Stages	Arithmetic	Continuity
Xi [23]	Church	yes	closed	FO-μ	$< \omega$	Presb., $*/$	—
Hughes/Pareto+ [12, 11, 17]	Curry	yes	closed	FO-$\mu\nu$	$\leq \omega$	Presb.	yes
Barthe/Giménez+ [7]	Curry	no	SN	HO-$\mu\nu$	$< \Omega$	$\_+1$ only	no
this	Curry	no	SN	HO-$\mu\nu$	$< \Omega$	$\_+1$ only	yes

Table 1. Recent work on type-based termination.

Contribution. This work addresses the question: Which types *are* legal as the result type τ of a recursive definition? We cannot give a final answer yet, but we have identified a class of types exceeding Barthe *el al.* [7] which follow a regular pattern. We analyzed the interpretation of types by sets of strongly normalizing terms and found that τ needs to be *continuous* for corecursive and something which we called *paracontinuous* for recursive definitions.

Hughes' *et al.* [12] system accepts types τ which are ω-*undershooting* and pass a *bottom-check*, but he worked in a domain theoretic denotational semantics with only first-order datatypes and it is not clear yet whether all of his results carry over when considering *strong* normalization and higher-order datatypes.

Table 1 compares some recent approaches to type-based termination. Xi [23] aims at functional programming with a cbv-reduction semantics for closed terms and first-order inductive datatypes, hence its sufficient for him to consider only finite stages and he does not need to address the problem of limit stages and continuity. His stage expressions are most expressive and support full arithmetic, but this makes type-checking semi-decidable if multiplication of stages is involved. Hughes and Pareto aim at embedded functional programming with streams and processes, they have treated the problem of continuity for ω-instantiation. Barthe and Giménez aim at theorem proving and support higher-order datatypes needed for proof theory. They show strong normalization, but do not investigate continuity, and here this works steps in.

Our calculus $\lambda^{\mathsf{fix}\mu\nu}$ might serve as the core of a total functional programming language or an interactive proof assistant. Since it also handles non-strictly positive types, it could help to investigate programs with continuation types or algorithms extracted from classical proofs via double-negation translation.

This article is organized as follows. In Sect. 2 we will present the untyped term language with reduction semantics, and in Sect. 3 we will define types, subtyping and type interpretation. Sect. 4 is devoted to continuous types and Sect. 5 to co- and paracontinuous types. In Sect. 6 we will introduce the typing rules and prove soundness.

2 Untyped Calculus

Figure 2 presents terms, evaluation contexts and reduction relations of $\lambda^{\mathsf{fix}\mu\nu}$, a core functional programming language with iso-recursive categorical datatypes, (fold, unfold), recursion (fix^{μ}) and corecursion (fix^{ν}). "fold" is the general constructor, sometimes called in; "unfold" the general destructor, also called out.

Terms $M, N \in \mathsf{TM}$.

$$M, N ::= x \mid \lambda x.M \mid M\ N \mid \mathsf{inl}\ M \mid \mathsf{inr}\ M \mid (\mathsf{case}\ M\ \mathsf{of}\ \mathsf{inl}\ x_1 \Rightarrow M_1 \mid \mathsf{inr}\ x_2 \Rightarrow M_2) \mid (M_1, M_2) \mid \mathsf{fst}\ M \mid \mathsf{snd}\ M \mid \mathsf{fold}\ M \mid \mathsf{unfold}\ M \mid \mathsf{fix}^\mu g.M \mid \mathsf{fix}^\nu g.M$$

Evaluation contexts.

$$E[\bullet] ::= \bullet \mid E[\bullet]\ M \mid \mathsf{unfold}\ E[\bullet] \mid (\mathsf{case}\ E[\bullet]\ \mathsf{of}\ \mathsf{inl}\ x_1 \Rightarrow M_1 \mid \mathsf{inr}\ x_2 \Rightarrow M_2) \mid \mathsf{fst}\ E[\bullet] \mid \mathsf{snd}\ E[\bullet] \mid (\mathsf{fix}^\mu g.M)\ E[\bullet]$$

β-Reduction axioms: Lambda calculus with sums and products +

$$\begin{array}{rl} \mathsf{unfold}\ (\mathsf{fold}\ M) & \longrightarrow_\beta M \\ (\mathsf{fix}^\mu g.M)\ (\mathsf{fold}\ N) & \longrightarrow_\beta ([\mathsf{fix}^\mu g.M/g]M)(\mathsf{fold}\ N) \\ \mathsf{unfold}\ E[\mathsf{fix}^\nu g.M] & \longrightarrow_\beta \mathsf{unfold}\ E[[\mathsf{fix}^\nu g.M/g]M] \end{array}$$

Reduction relations.

$\longrightarrow_\beta$	β-reduction
$\longrightarrow$	one-step reduction: closure of $\longrightarrow_\beta$ under all term constructors
$\longrightarrow^+$	transitive closure of $\longrightarrow$
$\longrightarrow^*$	reflexive-transitive closure of $\longrightarrow$

Fig. 2. $\lambda^{\mathsf{fix}\mu\nu}$: Terms, Evaluation Contexts and Reduction

Reduction axioms are standard, solely of interest are the rules for the fixed-points: Recursive functions fix^μ are only unfolded when applied to an argument guarded by a fold and infinite objects fix^ν only under a unfold-destructor. These restrictions are essential for establishing strong normalization. The reduction relation $\longrightarrow$, which results from closing $\longrightarrow_\beta$ under all term constructors, is confluent, which can be shown with the parallel-reduction method by Tait and Martin-Löf.

The set of *strongly normalizing terms* SN is defined as the wellfounded part of the set TM of terms wrt. the reduction relation $\longrightarrow$. All variables x and all subterms of strongly normalizing terms are strongly normalizing. Furthermore, the set SN is closed under reduction.

Proofs of strong normalization typically involve some standardization argument. In our case we use *weak head reduction*; it is a reduction of the form $E[M_1] \longrightarrow E[M_2]$ where $M_1 \longrightarrow_\beta M_2$. Terms of the form $E[x]$ are called *neutral*. A set of terms $P \subseteq \mathsf{TM}$ is called *saturated*, written $P \in \mathcal{SAT}$, if it contains only strongly normalizing terms, all strongly normalizing neutral terms, and if it is closed under weak head expansion. The *saturation* of a set P is defined as the closure under the following rules:

$$\frac{M \in P}{M \in P^*} \qquad \frac{E[x] \in \mathsf{SN}}{E[x] \in P^*} \qquad \frac{M \in \mathsf{SN} \qquad M \longrightarrow_\beta M' \qquad E[M'] \in P^*}{E[M] \in P^*}$$

The set SN is saturated, and so is the *function space* $P \to Q := \{M \mid\ M\ N \in Q \text{ for all } N \in P\}$, provided $P, Q \in \mathcal{SAT}$.

Types ($\nabla \in \{\mu, \nu\}$, $n \in \mathbb{N}$).

$$\rho, \sigma, \tau ::= X \mid \sigma \to \tau \mid \sigma + \tau \mid \sigma \times \tau \mid \nabla X\sigma \mid Y^n \mid \forall Y{\approx}\nabla X\sigma.\,\tau$$

Contexts.

$$\Gamma ::= \cdot \mid \Gamma, x{:}\tau \mid \Gamma, X \mid \Gamma, Y{\approx}\nabla X\sigma$$

Judgments.

Γ cxt	Γ is a wellformed context.
$\Gamma \vdash X$ pos σ	Variable X appears positively in σ
$\Gamma \vdash X$ neg σ	Variable X appears negatively in σ
$\Gamma \vdash X$ only pos in σ	All occurrences of X in σ are positive.
$\Gamma \vdash \tau$: type	τ is a wellformed type.
$\Gamma \vdash \rho \approx \nabla X\sigma$	ρ is an approximation of $\nabla X\sigma$.
$\Gamma \vdash \rho \le \sigma$	ρ is a subtype of σ.

Approximations.

$$\frac{(Y{\approx}\nabla X\sigma) \in \Gamma}{\Gamma \vdash Y^n \approx \nabla X\sigma}\approx Y \qquad \frac{}{\Gamma \vdash \nabla X\sigma \approx \nabla X\sigma}\approx \nabla$$

Subtyping.

$$\frac{\rho \in \{\mu X\sigma, \nu X\sigma\}}{\Gamma \vdash \rho \le \rho}\le \textsc{Base} \qquad \frac{(Y{\approx}\mu X\sigma) \in \Gamma}{\Gamma \vdash Y^n \le \mu X\sigma}\le Y_1^\mu \qquad \frac{(Y{\approx}\mu X\sigma) \in \Gamma \qquad n \le m}{\Gamma \vdash Y^n \le Y^m}\le Y_2^\mu$$

$$\frac{(Y{\approx}\nu X\sigma) \in \Gamma}{\Gamma \vdash \nu X\sigma \le Y^n}\le Y_1^\nu \qquad \frac{(Y{\approx}\nu X\sigma) \in \Gamma \qquad n \le m}{\Gamma \vdash Y^m \le Y^n}\le Y_2^\nu$$

$$\frac{\Gamma \vdash \sigma_1 \le \rho_1 \qquad \Gamma \vdash \rho_2 \le \sigma_2}{\Gamma \vdash \rho_1 \to \rho_2 \le \sigma_1 \to \sigma_2}\le \to \qquad \frac{\Gamma \vdash \rho_1 \le \sigma_1 \qquad \Gamma \vdash \rho_2 \le \sigma_2 \qquad \star \in \{+, \times\}}{\Gamma \vdash \rho_1 \star \rho_2 \le \sigma_1 \star \sigma_2}\le \star$$

$$\frac{\Gamma \vdash \rho \approx \nabla X\sigma \qquad \Gamma \vdash [\rho/Y]\tau \le \tau'}{\Gamma \vdash (\forall Y{\approx}\nabla X\sigma.\,\tau) \le \tau'}\le \forall L \qquad \frac{\Gamma, Y{\approx}\nabla X\sigma \vdash \rho \le \tau}{\Gamma \vdash \rho \le \forall Y{\approx}\nabla X\sigma.\,\tau}\le \forall R$$

Fig. 3. $\lambda^{\mathsf{fix}\mu\nu}$: Types and Subtyping.

3 Types

Figure 3 presents types and type related judgments. Most of these are standard, like wellformedness of contexts and positive and negative occurrences of variables. A type τ is wellformed ($\Gamma \vdash \tau$: type) if it follows the grammar with the restriction that if $\tau \equiv \nabla X\sigma$ is a (co)inductive type, X may appear only positively in σ and σ may not contain quantifiers. Approximation types Y^n and bounded quantification $\forall Y \approx \nabla X\sigma.\,\tau$ have been explained in the introduction already. Note that unit and empty type can be defined: $1 = \nu X.X$, $0 = \mu X.X$.

We regard all variables bound in a context as distinct. (Capture-avoiding) substitution for types is defined as usual and written $[\rho/X]\sigma$. Sometimes we exhibit a free variable in a type $\sigma(X)$ and write $\sigma(\rho)$ for the result of substitution. For approximations we define a special substitution given by the axioms $[X^m/Y]Y^n = X^{n+m}$ and $[\nabla X\sigma/Y]Y^n = \nabla X\sigma$ plus congruence rules for all type constructors. The second axiom is justified by the fact that $\nabla X\sigma$ denotes a fixed-point which is unaffected by n more iterations.

Subtyping is induced by approximation types. Since the semantics $[\![Y]\!]$ of some bounded type variable $Y \approx \mu X\sigma$ denotes some subset of $[\![\mu X\sigma]\!]$, we introduce the

subtyping rule $Y \leq \mu X\sigma$. Furthermore, since the approximations of an inductive type form an ascending chain bounded by the fixed-point of the approximation operator, we can safely add subtyping rules $Y^i \leq Y^j$ and $Y^i \leq \mu X\sigma$ for all indices i and j with $i \leq j$. Note that two different type variables $X \neq Y$ are incomparable. Subtyping rules for approximations of coinductive types are obtained by dualization.

To increase expressivity of subtyping, we have incorporated a reflexivity rule for base types, congruence rules for sum, product and function types (contravariant on the left!) and rules for quantification. The subtyping relation $\leq$ is reflexive and transitive. Since most rules are syntax-directed, it is not hard to come up with a decision algorithm for subtyping. The only critical rules are the ones for quantification: rule $\leq\forall R$ always has to be applied before $\leq\forall L$. When the rule $\leq\forall L$ fires, an approximation ρ has to be guessed. This can be implemented using existential variables and unification.

Type interpretation and soundness. In the following, we give an interpretation of types as sets of terms which we will later show to be saturated. Approximations will be interpreted as iterations Φ^α of monotonic operators $\Phi : \mathcal{P}(\mathsf{TM}) \to \mathcal{P}(\mathsf{TM})$. To distinguish between approximation from above and below, we define iterates as $\Phi^{\nabla,\alpha}$ with a tag $\nabla \in \{\mu, \nu\}$.

$$\begin{array}{rlrl} \Phi^{\mu,0} &= \emptyset^* & \Phi^{\nu,0} &= \mathsf{SN} \\ \Phi^{\mu,\alpha+1} &= \Phi(\Phi^{\mu,\alpha}) & \Phi^{\nu,\alpha+1} &= \Phi(\Phi^{\nu,\alpha}) \\ \Phi^{\mu,\lambda} &= \bigcup_{\alpha<\lambda} \Phi^{\mu,\alpha} & \Phi^{\nu,\lambda} &= \bigcap_{\alpha<\lambda} \Phi^{\nu,\alpha} \end{array}$$

When clear from the context, we omit the tags μ, ν. To interpret context as sets of substitutions, let P, Q from here denote sets of terms. Raw substitutions are given by the grammar $\theta ::= \cdot \mid \theta, x \mapsto M \mid \theta, X \mapsto P \mid \theta, Y \mapsto (\Phi, \nabla, \alpha)$. Note that approximation variables Y are mapped to a triple consisting of a monotonic operator Φ, a flag $\nabla \in \{\mu, \nu\}$ denoting whether Y approximates a least fixed-point from below or a greatest fixed-point from above, plus an ordinal α denoting the current iteration. Γ-substitutions $\theta : \Gamma$ are those with $\mathsf{dom}(\theta) = \mathsf{dom}(\Gamma)$.

Let $\Gamma \vdash \tau : \mathsf{type}$ and $\theta : \Gamma$. We define the type interpretation $[\![\tau]\!]\theta \subseteq \mathsf{TM}$ by recursion on τ.

$$\begin{array}{rlrl} [\![\sigma + \tau]\!]\theta &= \{\mathsf{inl}\, M \mid M \in [\![\sigma]\!]\theta\}^* \cup \{\mathsf{inr}\, M \mid M \in [\![\tau]\!]\theta\}^* & & \\ [\![\sigma \times \tau]\!]\theta &= \{M \mid \mathsf{fst}\, M \in [\![\sigma]\!]\theta, \mathsf{snd}\, M \in [\![\tau]\!]\theta\} & & \\ [\![\sigma \to \tau]\!]\theta &= [\![\sigma]\!]\theta \to [\![\tau]\!]\theta & [\![X]\!]\theta &= \theta(X) \\ [\![\mu X\sigma]\!]\theta &= \Phi^{\Omega}_{\mu X\sigma,\theta} & \Phi_{\mu X\sigma,\theta}(Q) &= \{\mathsf{fold}\, M \mid M \in [\![\sigma]\!](\theta, X \mapsto Q)\}^* \\ [\![\nu X\sigma]\!]\theta &= \Phi^{\Omega}_{\nu X\sigma,\theta} & \Phi_{\nu X\sigma,\theta}(Q) &= \{M \mid \mathsf{unfold}\, M \in [\![\sigma]\!](\theta, X \mapsto Q)\} \\ [\![Y^n]\!]\theta &= \Phi^{\alpha+n} & \text{where } \theta(Y) &= (\Phi, \nabla, \alpha) \\ [\![\forall Y \approx \nabla X\sigma.\, \tau]\!]\theta &= \bigcap_\alpha [\![\tau]\!](\theta, Y \mapsto (\Phi_{\nabla X\sigma,\theta}, \nabla, \alpha)) & & \end{array}$$

The "introduction-based" semantics of $+$ and μ has to be saturated explicitely (note the *), which is not necessary for "elimination-based" constructions ($\to$, $\times$, and ν). The semantics of (co)inductive types $\nabla X\sigma$ are defined as iterates

of operators at stage Ω, which is a (greatest) least fixed-point iff the semantics $[\![\tau]\!]$ is monotonic for each substitution θ, i.e., monotonic in every variable that occurs positively and antitonic in every variable that occurs negatively. To make this observation precise, we define *inclusion for substitutions*. Let $\Gamma \vdash \tau : \mathsf{type}$. Then $\Gamma \vdash_\tau \theta_1 \subseteq \theta_2$ is defined to hold iff θ_1 and θ_2 are Γ-substitutions and

- $\theta_1(X) \subseteq \theta_2(X)$ for all X with $\Gamma \vdash X\ \mathsf{pos}\ \tau$,
- $\theta_1(X) \supseteq \theta_2(X)$ for all X with $\Gamma \vdash X\ \mathsf{neg}\ \tau$, and
- $\theta_1(Y) = \theta_2(Y)$ for all $(Y{\approx}\nabla X\sigma) \in \Gamma$.

Lemma 1 (Soundness of Subtyping). *If $\Gamma \vdash \rho \leq \sigma$ and $\Gamma \vdash_\rho \theta_1 \subseteq \theta_2$ then $[\![\rho]\!]\theta_1 \subseteq [\![\sigma]\!]\theta_2$.*

Proof. By induction on $\Gamma \vdash \rho \leq \sigma$.

By reflexivity of subtyping, this lemma entails monotonicity of all types which, again, entails soundness of our fixed-point construction. By simple induction proofs we also establish that substitution for types and approximations is sound. Finally, we show is that types are interpreted as saturated sets. To this end, we define the semantical version $\theta \in [\![\Gamma]\!]$ of $\theta : \Gamma$.

$$\frac{}{\cdot \in [\![\cdot]\!]} \qquad \frac{M \in [\![\tau]\!]\theta \qquad \theta \in [\![\Gamma]\!]}{(\theta, x \mapsto M) \in [\![\Gamma, x:\tau]\!]} \qquad \frac{P \in \mathcal{SAT} \qquad \theta \in [\![\Gamma]\!]}{(\theta, X \mapsto P) \in [\![\Gamma, X]\!]}$$

$$\frac{\alpha \text{ ordinal} \qquad \theta \in [\![\Gamma]\!]}{(\theta, Y \mapsto (\Phi_{\nabla X\sigma,\theta}, \nabla, \alpha)) \in [\![\Gamma, Y{\approx}\nabla X\sigma]\!]}$$

Lemma 2 (Saturatedness). *If Γ cxt then for all $\theta \in [\![\Gamma]\!]$ and all X the set $\theta(X)$ is saturated. If $\Gamma \vdash \tau : \mathsf{type}$ then for all $\theta \in [\![\Gamma]\!]$ it holds that $[\![\tau]\!]\theta \in \mathcal{SAT}$.*

4 Continuous Types

In this section, we will identify a set of legal result types $\tau(Y)$ for corecursion (see Sect. 1). The key requirement on τ is continuity. An operator Φ is *continuous* if for all families P^α ($\alpha \in I$) of term sets it holds that

$$\bigcap_{\alpha\in I} \Phi(P^\alpha) \subseteq \Phi\left(\bigcap_{\alpha\in I} P^\alpha\right).$$

In the following we will motivate why continuity is a necessary condition on τ. Consider the typing rule for corecursion specialized to streams.

$$\frac{Y{\approx}\mathsf{Stream},\ g{:}\tau(Y) \vdash M : \tau(Y^1)}{\mathsf{fix}^\nu g.M : \forall Y{\approx}\mathsf{Stream}.\,\tau(Y)}$$

Let $S(Q) = \{M \in \mathsf{TM} \mid \mathsf{hd}\,M \in [\![\mathsf{Nat}]\!] \text{ and } \mathsf{tl}\,M \in Q\}$ be the semantical operator to construct the set of streams. Then $[\![\mathsf{Stream}]\!] = \bigcap_{\alpha<\omega} S^\alpha$. Hence, the

corecursion rule for streams can be proven sound by transfinite induction upto ω. In the limit case ω, we can use the induction hypothesis for all smaller stages, i.e., we can assume

$$\mathsf{fix}^\nu g.M \in \bigcap_{\alpha<\omega} [\![\tau]\!](S^\alpha) \quad \text{to show} \quad \mathsf{fix}^\nu g.M \in [\![\tau]\!]\left(\bigcap_{\alpha<\omega} S^\alpha\right) = [\![\tau]\!]([\![\mathsf{Stream}]\!]).$$

Obviously, we require $[\![\tau]\!]$ to be continuous.

A grammar for continuous types. We introduce a new judgment $\Gamma \vdash \boldsymbol{X}\ \mathsf{cont}\ \tau$, meaning that τ is continuous in the variables $\boldsymbol{X}$.

$$\frac{}{\Gamma \vdash \boldsymbol{X}\ \mathsf{cont}\ X_i} \qquad \frac{\boldsymbol{X} \notin \mathsf{FV}(\tau)}{\Gamma \vdash \boldsymbol{X}\ \mathsf{cont}\ \tau} \qquad \frac{\Gamma \vdash \boldsymbol{X}\ \mathsf{only\ pos\ in}\ \sigma \qquad \Gamma \vdash \boldsymbol{X}\ \mathsf{cont}\ \tau}{\Gamma \vdash \boldsymbol{X}\ \mathsf{cont}\ \sigma \to \tau}$$

$$\frac{\Gamma \vdash \boldsymbol{X}\ \mathsf{cont}\ \sigma, \tau}{\Gamma \vdash \boldsymbol{X}\ \mathsf{cont}\ \sigma + \tau} \qquad \frac{\Gamma \vdash \boldsymbol{X}\ \mathsf{cont}\ \sigma, \tau}{\Gamma \vdash \boldsymbol{X}\ \mathsf{cont}\ \sigma \times \tau} \qquad \frac{\Gamma, X \vdash \boldsymbol{X}, X\ \mathsf{cont}\ \sigma}{\Gamma \vdash \boldsymbol{X}\ \mathsf{cont}\ \nabla X\sigma}$$

$$\frac{(Y \approx \nabla X\sigma) \in \Gamma \qquad \Gamma \vdash \boldsymbol{X}\ \mathsf{cont}\ \nabla X\sigma}{\Gamma \vdash \boldsymbol{X}\ \mathsf{cont}\ Y^n} \qquad \frac{\Gamma, Y \approx \nabla X\sigma \vdash \boldsymbol{X}\ \mathsf{cont}\ \tau}{\Gamma \vdash \boldsymbol{X}\ \mathsf{cont}\ \forall Y \approx \nabla X\sigma.\, \tau}$$

For function types to be continuous, the domain just needs to be monotonic and the range continuous. Fixed-points are continuous if the respective operator is continuous and (of course) monotonic. Surprising is the fact that *least* fixed-points are continuous, which requires that union preserves continuity. Let Φ be the operator of an inductive type. Then

$$\bigcap_\alpha \bigcup_{\beta<\lambda} \Phi^\beta(P^\alpha) \subseteq \bigcup_{\beta<\lambda} \Phi^\beta\left(\bigcap_\alpha P^\alpha\right)$$

does not hold simply by set-theoretic means, even if Φ is continuous. In our semantics however, an intersection *can* be pulled into a union. Informally this may be explained as follows:

Consider polymorphic lists $\mathsf{List}(X)$ with operator $L(A)(Q) = \{\mathsf{nil}\} \cup \{a :: l \mid a \in A, l \in Q\}$. Note that for some fixed set A, $L(A)^n$ denotes the nth approximant to $[\![\mathsf{List}]\!](A)$. Recall that S was defined as the operator for streams. We say a stream s has goodness n if $s \in S^n$. Let M be some list of streams with the following property: At the same time, let M be a list of length m_0 of streams of goodness 0, a list of length m_1 of streams of goodness 1, ... a list of length m_n of streams of goodness n, ... Formally, $M \in L(S^i)^{m_i}$ for all i. Since a list can only have one length, all m_i must be equal. Hence $m_i = m_0$ for all i, and

$$\begin{aligned} &\textstyle\bigcap_n [\![\mathsf{List}]\!](S^n) = \bigcap_n (\bigcup_m (L(S^n))^m) = \bigcap_n (L(S^n))^{m_0} \\ &\subseteq (L(\textstyle\bigcap_n S^n))^{m_0} = (L([\![\mathsf{Stream}]\!]))^{m_0} \subseteq [\![\mathsf{List}]\!]([\![\mathsf{Stream}]\!]) \end{aligned}$$

This example shows that $\mathsf{List}(A)$ is continuous in A. To generalize this result for all inductive types, we make the following general observation that new inductive data is only generated by successor iterates:

Lemma 3 (Successor Iterates are Sufficient). *If* $\mathsf{fold}\, M \in \Phi^{\alpha}_{\mu X\sigma,\theta}$ *then* $\mathsf{fold}\, M \in \Phi^{\beta+1}_{\mu X\sigma,\theta}$ *for some* $\beta < \alpha$.

Now we give a formulation of the above explained independence result for inductive types $\mu Y\sigma$ with a free variable X.

Lemma 4 (Independence). *Let* $[\![\sigma(X,Y)]\!]$ *be continuous in the variables* X *and* Y *and* $\Phi(P)(Q) = \{\mathsf{fold}\, M \mid M \in [\![\sigma]\!](X \mapsto P, Y \mapsto Q)\}^*$. *Furthermore, let* (β_α) *a sequence of ordinals and* P^α *a sequence of sets of terms. Then*

$$\bigcap_\alpha (\Phi(P^\alpha))^{\beta_\alpha} \subseteq \Phi(\bigcap_\alpha P^\alpha)^{\beta_0}$$

This lemma is proven by induction on β_0 and can be generalized to continuous types $\sigma(\boldsymbol{X}, Y)$ by defining intersection on substitutions θ. It then yields soundness of syntactic continuity.

Theorem 1 (Soundness of cont). *Let* $\Gamma \vdash \tau : \mathsf{type}$. *If* $\Gamma \vdash \boldsymbol{X}\ \mathsf{cont}\ \tau$, *then* $[\![\tau]\!]$ *is continuous in* $\boldsymbol{X}$.

As a side product of our considerations, we notice that all *strictly positive* coinductive types are continuous. Thus, their fixed-point is reached at iteration ω.

Types for infinite objects. Can all continuous types be used for fix^ν-definitions? Given our reduction semantics, the answer is no. Sums, inductive and coinductive types may be continuous, but they do not provide the necessary guardedness. For example, let

$$\mathsf{map}_0 \;=\; \mathsf{map}\,(\lambda x.0 :: x) \;:\; \forall Y{\approx}\mathsf{Stream.List}(Y) \to \mathsf{List}(Y^1)$$

Then $M = \mathsf{fix}^\nu g.\, \mathsf{map}_0\, g$ could be ascribed type $\forall Y{\approx}\mathsf{Stream.\,List}(Y)$. But unfolding of M diverges: $\mathsf{unfold}\, M \longrightarrow_\beta \mathsf{unfold}(\mathsf{map}_0\, M) \longrightarrow_\beta \mathsf{unfold}(\mathsf{map}_0\,(\mathsf{map}_0\, M)) \longrightarrow_\beta \ldots$ Similar examples can be found for types with $+$ and ν. What remains as legal types for fix^ν, are function space and product. By a judgment $Y\ \mathsf{legal}^\nu\ \tau$, we give a grammar for these types.

$$\frac{}{Y\ \mathsf{legal}^\nu\ Y} \qquad \frac{Y\ \mathsf{only\ pos\ in}\ \sigma \qquad Y\ \mathsf{legal}^\nu\ \tau}{Y\ \mathsf{legal}^\nu\ \sigma \to \tau} \qquad \frac{Y\ \mathsf{legal}^\nu\ \sigma \qquad Y\ \mathsf{legal}^\nu\ \tau}{Y\ \mathsf{legal}^\nu\ \sigma \times \tau}$$

Examples for legal types τ according to this grammar which are *not* of the form $\tau(Y) \to Y$ (τ monotone), like $Y \to Y \to Y$ and $\mathsf{Nat} \times Y$, have been given in the introduction (Fibonacci streams).

5 Co- & Paracontinuous Types

As we have answered for fix^ν, we can ask for the recursive function constructor fix^μ: Which types $\tau(Y)$ would be legal in the rule

$$\frac{Y{\approx}\mu X\sigma,\ g{:}\tau(Y) \vdash M : \tau(Y)}{\mathsf{fix}^\mu g.M : \forall Y{\approx}\mu X\sigma.\, \tau(Y)}$$

This rule would be proven sound by transfinite induction on the approximations Φ^α of the inductive type $\mu X\sigma$. For limit steps λ, we may assume

$$\mathsf{fix}^\mu g.M \in \bigcap_{\alpha<\lambda} [\![\tau]\!](\Phi^\alpha) \qquad \text{to show} \qquad \mathsf{fix}^\mu g.M \in [\![\tau]\!](\bigcup_{\alpha<\lambda} \Phi^\alpha).$$

We call such types $\tau(Y)$, which permit this inference, *paracontinuous*. Obviously paracontinuous are types of the form $Y \to \tau(Y)$ for Y $\mathsf{only\ pos\ in}$ τ. However, this does not include the most precise type for, e.g., the maximum function $\mathsf{max} : \forall Y{\approx}\mathsf{Nat}.\, Y \to (Y \to Y)$. Here Y occurs negatively in $Y \to Y$, but the type is still paracontinuous. On our journey to identify paracontinuous types syntactically, we will first consider cocontinuity, the concept dual to continuity.

Cocontinuity. An operator $\Phi : \mathcal{P}(\mathsf{TM}) \to \mathcal{P}(\mathsf{TM})$ is called *cocontinuous*, if for all chains P (with $P^\alpha \subseteq P^\beta$ for $\alpha < \beta$) it holds that

$$\Phi(\bigcup_\alpha P^\alpha) \subseteq \bigcup_\alpha \Phi(P^\alpha)$$

In the following we will identify the types τ for which the semantics $[\![\tau]\!]$ is cocontinuous. Let us first consider products.

Lemma 5 (Products preserve Cocontinuity). *Let $\Gamma \vdash \sigma(X), \tau(X) : \mathsf{type}$. If $[\![\sigma]\!]$ and $[\![\tau]\!]$ are cocontinuous and monotonic in X, so is $[\![\sigma \times \tau]\!]$.*

Proof. Assume $M \in [\![\sigma \times \tau]\!](\bigcup_\alpha P^\alpha)$. Then by definition, $\mathsf{fst}\, M \in [\![\sigma]\!](\bigcup_\alpha P^\alpha)$ and $\mathsf{snd}\, M \in [\![\tau]\!](\bigcup_\alpha P^\alpha)$. Since σ and τ are cocontinuous by assumption, there are ordinals α, β such that $\mathsf{fst}\, M \in [\![\sigma]\!](P^\alpha)$ and $\mathsf{snd}\, M \in [\![\tau]\!](P^\beta)$. Now let $\gamma = \max(\alpha, \beta)$. Since P is a chain and $[\![\sigma]\!]$ and $[\![\tau]\!]$ are monotonic, $\mathsf{fst}\, M \in [\![\sigma]\!](P^\gamma)$ and $\mathsf{snd}\, M \in [\![\tau]\!](P^\gamma)$. We conclude $M \in [\![\sigma \times \tau]\!](P^\gamma)$.

Thus, products are cocontinuous because the binary maximum always exists. Since function types and coinductive types can be viewed as infinite products, they are not cocontinuous in general. For example, $\lambda x.x \in \mathsf{List} \to \bigcup_{\alpha<\omega} L^\alpha$, but it does not inhabit $\mathsf{List} \to L^\alpha$ for any $\alpha < \omega$. We expect that the same holds for quantification. Sums and inductive types are trivially cocontinuous, provided they are monotonic. Hence, we can give a grammar for cocontinuous types via a judgment $\Gamma \vdash \boldsymbol{X}\ \mathsf{cocont}\ \tau$.

$$\frac{}{\Gamma \vdash \boldsymbol{X}\ \mathsf{cocont}\ X_i} \qquad \frac{X \notin \mathsf{FV}(\tau)}{\Gamma \vdash X\ \mathsf{cocont}\ \tau} \qquad \frac{\Gamma \vdash \boldsymbol{X}\ \mathsf{cocont}\ \sigma, \tau}{\Gamma \vdash \boldsymbol{X}\ \mathsf{cocont}\ \sigma + \tau, \sigma \times \tau}$$

$$\frac{\Gamma, X \vdash \boldsymbol{X}, X\ \mathsf{cocont}\ \sigma}{\Gamma \vdash \boldsymbol{X}\ \mathsf{cocont}\ \mu X\sigma} \qquad \frac{(Y{\approx}\mu X\sigma) \in \Gamma \qquad \Gamma \vdash \boldsymbol{X}\ \mathsf{cocont}\ \mu X\sigma}{\Gamma \vdash \boldsymbol{X}\ \mathsf{cocont}\ Y^n}$$

Note that $\Gamma \vdash \boldsymbol{X}\ \mathsf{cocont}\ \tau$ implies $\Gamma \vdash \boldsymbol{X}\ \mathsf{only\ pos\ in}\ \tau$. The grammar describes polynomial interleaved (nested) inductive types, which includes, e.g., natural numbers, lists and finitely branching trees, i.e., almost all inductive types used in practical programming. Furthermore all of these types are continuous and close at ω.

Lemma 6 (Soundness). *If $\Gamma \vdash \boldsymbol{X}$ cocont τ, then $[\![\tau]\!]$ is cocontinuous and monotonic in $\boldsymbol{X}$.*

Proof. By induction on $\Gamma \vdash \boldsymbol{X}$ cocont τ, using Lemma 5.

Paracontinuity. The considerations at the start of Sect. 5 suggested a definition of paracontinuity immediately. However, to make it work, we have to strengthen it a little.

An operator Φ is *paracontinuous* if for all chains P it holds that

$$\text{for all } \alpha_0 : \bigcap_{\alpha_0 \leq \alpha} \Phi(P^\alpha) \subseteq \Phi(\bigcup_\alpha P^\alpha).$$

Obviously all monotonic operators are paracontinuous. For function types, the domain has to be *cocontinuous*.

Lemma 7 (Paracontinuous Function Types). *If $[\![\tau]\!]$ is paracontinuous in X and $[\![\sigma]\!]$ is cocontinuous and monotonic in X, then $[\![\sigma \to \tau]\!]$ is paracontinuous in X.*

Proof. Fix some α_0 and assume $M \in \bigcap_{\alpha_0 \leq \alpha} [\![\sigma \to \tau]\!](P^\alpha)$ and $N \in [\![\sigma]\!](\bigcup_\alpha P^\alpha)$. Since σ is cocontinuous, $N \in [\![\sigma]\!](P^\beta)$ for some β. Let $\gamma = \max(\alpha_0, \beta)$. Now, since P^α is a ascending chain, we can exploit the monotonicity of $[\![\sigma]\!]$ to infer $N \in [\![\sigma]\!](P^\alpha)$ for all $\alpha \geq \gamma$. By assumption, $M\,N \in [\![\tau]\!](P^\alpha)$ for all $\alpha \geq \gamma$. Finally, since τ is paracontinuous, $M\,N \in [\![\tau]\!](\bigcup_\alpha P^\alpha)$.

For types $\sigma(X, Y)$ which are *continuous* in Y, fixed-point formation $\nabla Y \sigma$ preserves paracontinuity in X. The case of inductive types is especially interesting, since it makes again use of the Independence Lemma.

Lemma 8 (Paracontinuous Inductive Types). *If $[\![\sigma(X, Y)]\!]$ is paracontinuous in X and continuous and monotonic in Y, then $[\![\mu Y \sigma]\!]$ is paracontinuous in X.*

Proof. By transfinite induction on the stages of $[\![\mu Y \sigma]\!]$, using Lemma 4.

For coinductive types $\nu Y \sigma$ the proof is quite similar, but can be done without Lemma 4. The proven lemmata can be generalized to paracontinuity in several variables $\boldsymbol{X}$.

We give a grammar for paracontinuous types by the judgment $\Gamma \vdash \boldsymbol{X}$ para τ.

$$\frac{}{\Gamma \vdash \boldsymbol{X} \text{ para } X_i} \qquad \frac{\boldsymbol{X} \notin \mathsf{FV}(\tau)}{\Gamma \vdash \boldsymbol{X} \text{ para } \tau} \qquad \frac{\Gamma \vdash \boldsymbol{X} \text{ cocont } \sigma \qquad \Gamma \vdash \boldsymbol{X} \text{ para } \tau}{\Gamma \vdash \boldsymbol{X} \text{ para } \sigma \to \tau}$$

$$\frac{\Gamma \vdash \boldsymbol{X} \text{ para } \sigma, \tau}{\Gamma \vdash \boldsymbol{X} \text{ para } \sigma + \tau, \sigma \times \tau} \qquad \frac{\Gamma, X \vdash X \text{ cont } \sigma \qquad \Gamma, X \vdash \boldsymbol{X}, X \text{ para } \sigma}{\Gamma \vdash \boldsymbol{X} \text{ para } \nabla X \sigma}$$

$$\frac{(Y \approx \nabla X \sigma) \in \Gamma \qquad \Gamma \vdash \boldsymbol{X} \text{ para } \nabla X \sigma}{\Gamma \vdash \boldsymbol{X} \text{ para } Y^n} \qquad \frac{\Gamma, Y \approx \nabla X \sigma \vdash \boldsymbol{X} \text{ para } \tau}{\Gamma \vdash \boldsymbol{X} \text{ para } \forall Y \approx \nabla X \sigma . \tau}$$

Theorem 2 (Soundness). *Let* $\Gamma \vdash \tau : \mathsf{type}$. *If* $\Gamma \vdash \boldsymbol{X}\ \mathsf{para}\ \tau$ *then* $[\![\tau]\!]$ *is paracontinuous in* $\boldsymbol{X}$.

Proof. By induction on $\Gamma \vdash \boldsymbol{X}\ \mathsf{para}\ \tau$, using Lemmata 7 and 8.

Note that $\Gamma \vdash \boldsymbol{X}\ \mathsf{cont}\ \tau$ implies $\Gamma \vdash \boldsymbol{X}\ \mathsf{para}\ \tau$. A *non*-paracontinuous type is $\tau(Y) = \mathsf{Stream}(Y) \to \mathsf{Bool}$ (acknowledgment: John Hughes). A function "double" which checks whether a stream has duplicate elements would be total for streams of numbers bounded by some $\alpha < \omega$ (returning always true) but not for streams of arbitrary natural numbers $\alpha = \omega$.

As for continuous types we raise the question: Are all paracontinuous types legal result types for the recursion constructor fix^μ? Again, the answer is no. Our reduction semantics only permits types $Y \to \tau(Y)$ for $Y\ \mathsf{para}\ \tau$, which is still a great improvement over $Y\ \mathsf{only\ pos\ in}\ \tau$.

Typing (τ and all types in Γ closed):

$$\Gamma \vdash M : \tau$$

(Lambda-calculus with products and sums +) Folding.

$$\frac{\Gamma \vdash M : [\nabla X\sigma/X]\sigma}{\Gamma \vdash \mathsf{fold}\, M : \nabla X\sigma} \qquad \frac{(Y \approx \nabla X\sigma) \in \Gamma \qquad \Gamma \vdash M : [Y^n/X]\sigma}{\Gamma \vdash \mathsf{fold}\, M : Y^{n+1}}$$

Unfolding.

$$\frac{\Gamma \vdash M : \nabla X\sigma}{\Gamma \vdash \mathsf{unfold}\, M : [\nabla X\sigma/X]\sigma} \qquad \frac{(Y \approx \nabla X\sigma) \in \Gamma \qquad \Gamma \vdash M : Y^{n+1}}{\Gamma \vdash \mathsf{unfold}\, M : [Y^n/X]\sigma}$$

Fixed-Points.

$$\frac{\Gamma,\ Y \approx \mu X\sigma,\ g : Y \to \tau(Y) \vdash M : Y^1 \to \tau(Y^1) \qquad \Gamma \vdash Y\ \mathsf{para}\ \tau(Y)}{\Gamma \vdash \mathsf{fix}^\mu g.M : \forall Y \approx \mu X\sigma.\, Y \to \tau(Y)}$$

$$\frac{\Gamma,\ Y \approx \nu X\sigma,\ g : \tau(Y) \vdash M : \tau(Y^1) \qquad \Gamma \vdash Y\ \mathsf{legal}^\nu\ \tau(Y)}{\Gamma \vdash \mathsf{fix}^\nu g.M : \forall Y \approx \nu X\sigma.\, \tau(Y)}$$

Subsumption.

$$\frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash \sigma \leq \tau}{\Gamma \vdash M : \tau} \qquad \frac{\Gamma, Y \approx \nabla X\sigma \vdash M : \tau}{\Gamma \vdash M : \forall Y \approx \nabla X\sigma.\, \tau}$$

Fig. 4. $\lambda^{\mathsf{fix}\mu\nu}$: Typing rules.

6 Typing and Soundness

Figure 4 displays the typing rules of $\lambda^{\mathsf{fix}\mu\nu}$. We excluded polymorphism, hence terms are only assigned *closed* types. By this we mean that τ contains no free type variables X, whereas approximations Y^n of fixed-point types are permitted.

The rules for lambda-calculus, sums and products are omitted since they are standard. For wrapping (fold) and unwrapping (unfold) (co)inductive types there

are two rules each, one rule for the fixed-point and one dealing with approximations Y^{n+1}, keeping track of the stage. The rules for (co)recursion carry the side conditions developed in the previous sections. Furthermore, we have rules for subsumption (which includes $\forall$-instantiation) and $\forall$-introduction independent of a fix^{∇}.

Reduction is type preserving, I have shown this for a slightly simpler system [2]. The proof of soundness of typing (implying strong normalization) is similar to Barthe/Giménez [7], although I developed it independently following Abel/Altenkirch [3]. Due to lack of space I can only give a sketch.

Let the application $M\theta$ of a substitution θ to a term M be defined as expected.

Lemma 9 (Soundness for fix^{μ}). *Let $\Gamma \vdash Y\ \mathsf{para}\ \tau$, $\theta \in [\![\Gamma]\!]$ a substitution and $\Phi = \Phi_{\mu X\sigma,\theta}$. Assume $M\theta' \in [\![Y^1 \to \tau(Y^1)]\!]\theta'$ for all $\theta' \in [\![\Gamma, Y{\approx}\mu X\sigma, g : Y \to \tau(Y)]\!]$. Then*

$$\mathsf{fix}^{\mu} g.M\theta \in \bigcap_{\alpha<\Omega} \Phi^{\alpha} \to [\![\tau]\!](\theta, Y \mapsto (\Phi, \mu, \alpha)).$$

Proof. By transfinite induction on α. The base case holds since Φ^0 contains no canonical terms. The step case makes use of saturatedness, the ind. hyp. and the assumption on M. The limit case follows by paracontinuity.

The soundness of fix^{ν} requires a little more work. First we give a judgment $E[\bullet] \in [\![\tau \multimap \rho]\!]\theta$ that types some evaluation contexts.

$$\frac{}{\bullet \in [\![\rho \multimap \rho]\!]\theta} \qquad \frac{E[\bullet] \in [\![\tau \multimap \rho]\!]\theta \qquad N \in [\![\sigma]\!]\theta}{E[\bullet\, N] \in [\![(\sigma \to \tau) \multimap \rho]\!]}$$

$$\frac{E[\bullet] \in [\![\tau_1 \multimap \rho]\!]}{E[\mathsf{fst}\,\bullet] \in [\![\tau_1 \times \tau_2 \multimap \rho]\!]} \qquad \frac{E[\bullet] \in [\![\tau_2 \multimap \rho]\!]}{E[\mathsf{snd}\,\bullet] \in [\![\tau_1 \times \tau_2 \multimap \rho]\!]}$$

The linear arrow $\multimap$ is not a first-class type constructor, rather some notation. It is motivated by the fact that evaluation contexts E have exactly one hole $\bullet$.

Lemma 10. *Let $\Gamma \vdash Y\ \mathsf{legal}^{\nu}\ \tau$ and some $\theta \in [\![\Gamma]\!]$. Assume that for all $E[\bullet] \in [\![\tau \multimap Y]\!]\theta$, $E[M] \in [\![Y]\!]\theta$. Then $M \in [\![\tau]\!]\theta$.*

Proof. By induction on $\Gamma \vdash Y\ \mathsf{legal}^{\nu}\ \tau$.

Lemma 11 (Soundness of fix^{ν}). *Let $\Gamma \vdash Y\ \mathsf{legal}^{\nu}\ \tau$, substitution $\theta \in [\![\Gamma]\!]$ and $\Phi = \Phi_{\nu X\sigma,\theta}$. Assume $M\theta' \in [\![\tau(Y^1)]\!]\theta'$ for all $\theta' \in [\![\Gamma, Y{\approx}\nu X\sigma, g{:}\tau(Y)]\!]$. Then*

$$\mathsf{fix}^{\nu} g.M\theta \in \bigcap_{\alpha<\Omega} [\![\tau]\!](\theta, Y \mapsto (\Phi, \nu, \alpha))$$

Proof. By transfinite induction on α. Note that the assumption implies $G \equiv \mathsf{fix}^{\nu} g.M\theta \in \mathsf{SN}$. The base case follows by Lemma 10 since $\Phi^0 = \mathsf{SN}$, and the limit case is a consequence of continuity of $[\![\tau(Y)]\!]$. The step case makes crucial use of Lemma 10, since we consider $\mathsf{unfold}\, E[G] \longrightarrow_{\beta} \mathsf{unfold}\, E[M\theta']$ for an appropriate extension θ' of θ and all $E[\bullet] \in [\![\tau \multimap Y]\!](\theta, Y \mapsto \Phi^{\alpha+1})$. The r.h.s. can be shown to be in the semantics by assumption, so the l.h.s. as well by saturation, which entails the goal. $\square$

Theorem 3 (Soundness of Typing). $\Gamma \vdash M : \tau$, $\theta \in [\![\Gamma]\!]$ *imply* $M\theta \in [\![\tau]\!]$.

Soundness, proved by induction on $\Gamma \vdash M : \tau$, now directly entails strong normalization (choose some θ which keeps all term variables fixed).

References

1. Andreas Abel. Specification and verification of a formal system for structurally recursive functions. In *TYPES '99*, vol. 1956 of *LNCS*, pages 1–20. Springer, 2000.
2. Andreas Abel. Termination checking with types. Technical Report 0201, Institut für Informatik, Ludwigs-Maximilians-Universität München, 2002.
3. A. Abel and T. Altenkirch. A predicative strong norm. proof for a λ-calculus with interleaving inductive types. In *TYPES '99*, vol. 1956 of *LNCS*. Springer, 2000.
4. Andreas Abel and Thorsten Altenkirch. A predicative analysis of structural recursion. *Journal of Functional Programming*, 12(1):1–41, January 2002.
5. Thorsten Altenkirch. *Constructions, Inductive Types and Strong Normalization*. PhD thesis, University of Edinburgh, November 1993.
6. Roberto M. Amadio and Solange Coupet-Grimal. Analysis of a guard condition in type theory. In *FoSSaCS '98*, volume 1378 of *LNCS*. Springer, 1998.
7. G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. *Math. Struct. in Comp. Sci.*, 2002. To appear.
8. Thierry Coquand. Infinite objects in type theory. In *TYPES '93*, volume 806 of *LNCS*, pages 62–78. Springer, 1993.
9. Eduardo Giménez. Codifying guarded definitions with recursive schemes. In *TYPES´94*, volume 996 of *LNCS*, pages 39–59. Springer, 1995.
10. Martin Hofmann. Non strictly positive datatypes for breadth first search. TYPES mailing list, 1993.
11. John Hughes and Lars Pareto. Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. In *ICFP'99*, pages 70–81, 1999.
12. John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *POPL'96*, pages 410–423. ACM Press, 1996.
13. INRIA. *The Coq Proof Assistant Reference Manual*, version 7.0 edition, April 2001.
14. Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *POPL'01*. ACM Press, 2001.
15. David McAllester and Kostas Arkoudas. Walther Recursion. In *CADE-13*, volume 1104 of *LNCS*. Springer, 1996.
16. Nax Paul Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51(1–2):159–172, 1991.
17. Lars Pareto. *Types for Crash Prevention*. PhD thesis, Chalmers University of Technology, 2000.
18. Frank Pfenning and Carsten Schürmann. Twelf - a meta-logical framework for deductive systems. In *CADE-16*, volume 1632 of *LNAI*. Springer, 1999.
19. Brigitte Pientka. Termination and reduction checking for higher-order logic programs. In *IJCAR 2001*, volume 2083 of *LNAI*, pages 401–415. Springer, 2001.
20. Randy Pollack. *The Theory of LEGO*. PhD thesis, University of Edinburgh, 1994.
21. Christophe Raffalli. Data types, infinity and equality in System AF2. In *CSL '93*, volume 832 of *LNCS*, pages 280–294. Springer, 1994.
22. Alastair J. Telford and David A. Turner. Ensuring streams flow. In *AMAST '97*, volume 1349 of *LNCS*, pages 509–523. Springer, 1997.
23. Hongwei Xi. Dependent types for program termination verification. *Journal of Higher-Order and Symbolic Computation*, 15:91–131, 2002.