GENERAL RECURSION VIA COINDUCTIVE TYPES

VENANZIO CAPRETTA

Department of Mathematics and Statistics, University of Ottawa, 585 King Edward Avenue, Room 204, Ottawa, ON, Canada, K1N 4N5
e-mail address: venanzio.capretta@mathstat.uottawa.ca

ABSTRACT. A fertile field of research in theoretical computer science investigates the representation of general recursive functions in intensional type theories. Among the most successful approaches are: the use of wellfounded relations, implementation of operational semantics, formalization of domain theory, and inductive definition of domain predicates. Here, a different solution is proposed: exploiting coinductive types to model infinite computations. To every type A we associate a type of partial elements A^ν, coinductively generated by two constructors: the first, $\ulcorner a \urcorner$ just returns an element $a \colon A$; the second, $\triangleright x$, adds a computation step to a recursive element $x \colon A^\nu$. We show how this simple device is sufficient to formalize all recursive functions between two given types. It allows the definition of fixed points of finitary, that is, continuous, operators. We will compare this approach to different ones from the literature. Finally, we mention that the formalization, with appropriate structural maps, defines a strong monad.

1. INTRODUCTION

Type theory is—we often claim—a rich functional programming language with dependent types and, at the same time, a constructive logical system. This view is consistently adopted in Martin-Löf's type theory [39, 48]. However, a serious limitation of type theory with respect to standard functional programming languages is that all computations must terminate. This restriction has two reasons. First, to decide type-checking of dependent types, we need to reduce type expressions to normal form (see the work by Barendregt and Geuvers [5] for a good exposition of technical issues of type-theoretic proof assistants). Second, since propositions are represented by types and proofs by programs, according to the *Curry-Howard isomorphism* [37, 28, 4, 55], we cannot allow non-terminating proofs, because they would lead to inconsistency.

There have been attempts to overcome both problems in the literature, starting from the work of Paulson [50] and Nordström [47].

There are implementation of *extensional* type theory for which the type-checking problem is undecidable, notably Nuprl [20]. In such a system the correctness of a judgment cannot be determined by just analyzing the terms, but a supplementary proof must be

given. In principle, a complete type-checkable term can be generated from such proof. Types of partial functions have been devised by Constable and added to Nuprl [21, 22].

Geuvers, Poll, and Zwanenburg [29] added a fixed point operator Y to type theory and proved the conservativity of the extension. This means that a judgment derived in the extended system and not containing Y can be derived in the non-extended system. This allows the use of Y for proof search: We construct a proof containing Y, then we reduce it; if we obtain a term not containing Y, we have a valid proof. However, statements about diverging elements cannot be proved, because valid proofs cannot contain Y.

Recent research has tried to find good representations of general recursive functions in type theory following various avenues: Balaa and Bertot used well-founded recursion [3]; Dubois and Donzeau-Gouge and, independently, Bove and Capretta used inductive characterizations of domain predicates [26, 14, 15, 16, 17]; Bertot, Capretta, and Das Barman combined the two methods to give a semantics of imperative programming [13] and Bertot extended the work to coinductive types [12]; Barthe and others used type labeling to strengthen termination conditions [7, 9]; McBride and McKinna used *views*, that is, different potential inductive characterizations of data types [41].

The first problem is how to represent partial functions. The most popular way consists in seeing a partial function $f \colon A \rightharpoonup B$ as a total function defined on a subset of the type A. In intensional type theory, a subset of a type is represented by a predicate $D \colon A \to \mathsf{Prop}$. A partial function is represented by a term $f \colon \Pi x \colon A.(D\ x) \to B$, provided that the result is not dependent on the proof of the predicate, that is, for $x \colon A$, if p_1 and p_2 are two proofs of $(D\ x)$, then $(f\ x\ p_1) = (f\ x\ p_2)$.

This formalization of partial functions has two defects. First, if we want to define the type of partial functions from A to B, we need second order sum types:

$$A \rightharpoonup B \equiv \ \Sigma D \colon A \to \mathsf{Prop}.$$
$$\Sigma f \colon \Pi x \colon A.(D\ x) \to B.$$
$$\forall x \colon A.\forall p_1, p_2 \colon (D\ x).(f\ x\ p_1) = (f\ x\ p_2).$$

This requires either an impredicative type theory [31, 32, 24] or a predicative theory with type universes [40]. In the second case, there are some difficulties when we try to define the predicate inductively from the recursive equations of the function: The counterexample ltz in [17] cannot be formalized without impredicativity. Second, diverging elements are not represented. We can apply a partial function only to the elements in its domain, but we cannot translate an expression $(f\ a)$ if a does not satisfy D.

I propose a different approach. The type of partial functions from A to B is defined as $A \to B^\nu$, where B^ν is a type in which B can be embedded and that contains terms representing partial elements. The standard choice in the semantics of programming languages is to put $B^\nu \equiv B + \{\bot\}$, where $\{\bot\}$ is a one-element set [54, 60]. This solution does not obtain in type theory. Sum types are decidable, that is, it is decidable whether an element of $B + \{\bot\}$ is in B or is $\bot$. This would imply that we can decide when a function diverges. It is then clear that not all recursive functions are representable. However, adopting ideas of Scott [54] we can constructivize this notion by formalizing finite elements and then take the filter completion. This approach was adopted by Michael Hedberg [35], who formalized constructive domain theory as a step toward a model of partial type theory inside total type theory.

Instead, here we choose as B^ν a certain coinductive type. Coinductive types were first introduced in type theory by Hagino [34], their second order implementation was studied

by Wraith [61] and Geuvers [27]. The version we use here was developed by Coquand [23] and implemented in Coq by Gimenez [30].

Before defining B^ν formally, let us see the model of computation on which it is based.

We give a description of computable functions that is Turing complete and is as simple as standard formalizations like Kleene's recursive functions, the pure λ-calculus, and Turing machines. It has the advantage that it translates easily in type theory.

We start with a system $\lambda_{\mathbf{Alg}}$ containing algebraic data types with definition by cases and function types with λ-abstraction and application. Then we introduce a new rule **Iter** to define functions. To define a function f from a type A to a type B we assign to every element x of A either directly a value b in B, and we indicate it by $\ulcorner b \urcorner$, or another element a' in A, and we indicate it by $\triangleright a'$. In other words, we give a function $g\colon A \to A + B$ and we indicate by $\triangleright \cdot$ and $\ulcorner \cdot \urcorner$ the left and right injection, respectively. The rule **Iter** states that we obtain a function $f = (\mathsf{iter}\ g)\colon A \to B$:

$$
\begin{array}{lll}
\begin{array}{l} f\colon A \to B \\ a_1 \mapsto \ulcorner b \urcorner \\ a_2 \mapsto \triangleright a' \end{array}
& \text{means} &
\begin{array}{l} g\colon A \to A + B \\ a_1 = \mathsf{inr}\ b \\ a_2 = \mathsf{inl}\ a' \end{array}
\qquad
\dfrac{g\colon A \to A + B}{f = (\mathsf{iter}\ g)\colon A \to B}.
\end{array}
$$

The function f has the following computation behavior. The computation of f on input a proceed recursively: If $(g\ a) = \mathsf{inr}\ b$, then the computation of $(f\ a)$ terminates with result b; if $(g\ a) = \mathsf{inl}\ a'$, we proceed to the computation of $(f\ a')$ and let $(f\ a) \rightsquigarrow (f\ a')$: if, continuing in the process, we eventually obtain a result $(f\ a') = b$, then we put $(f\ a) = b$; on the other hand, if the evaluation always gives a new element of A on which to compute f, then the computation will continue indefinitely and the value of $(f\ a)$ will be undetermined:

$$
\begin{array}{l}
f\ a_1 \rightsquigarrow b \\
f\ a_2 \rightsquigarrow f\ a'.
\end{array}
$$

Theorem 1.1. *The system* $\lambda_{\mathbf{Alg}} + \mathbf{Iter}$ *is Turing-complete.*

Proof. We give just an outline of the proof. A complete version of it, formalized in type theory, is given later (Definition 4.1 and Theorem 4.2). We show that all recursive functions can be programmed. The zero constant, successor, and projections can be defined directly by constructors and cases.

Let f be defined by primitive recursion, that is, given $g\colon \mathbb{N}^n \to \mathbb{N}$ and $h\colon \mathbb{N}^{n+2} \to \mathbb{N}$, we have

$$
\begin{array}{l}
f\colon \mathbb{N}^{n+1} \to \mathbb{N} \\
(f\ \overline{x}\ 0) = (g\ \overline{x}) \\
(f\ \overline{x}\ (y+1)) = (h\ \overline{x}\ y\ (f\ \overline{x}\ y)).
\end{array}
$$

We use a technique called *continuation-passing style programming*, that consists in seeing programs as maps on functions [53]. We add a functional parameter and define a function on $\mathbb{N}^n \times (\mathbb{N} \to \mathbb{N}) \times \mathbb{N}$ using definition by cases and the rule **Iter**:

$$
\begin{array}{l}
\langle \overline{x}, u, 0 \rangle \mapsto \ulcorner (u\ (g\ \overline{x})) \urcorner \\
\langle \overline{x}, u, y+1 \rangle \mapsto \triangleright \langle \overline{x}, \lambda z.(u\ (h\ \overline{x}\ y\ z)), y \rangle.
\end{array}
$$

Call f^+ the function so defined, $f^+\colon \mathbb{N}^n \times (\mathbb{N} \to \mathbb{N}) \times \mathbb{N} \to \mathbb{N}$. Then we put

$$
\begin{array}{l}
f\colon \mathbb{N}^{n+1} \to \mathbb{N} \\
f\ \overline{x}\ y = f^+\ \langle \overline{x}, \mathsf{id}, y \rangle.
\end{array}
$$

Let f be defined by minimization, that is, given $g\colon \mathbb{N}^{n+1} \to \mathbb{N}$, we have

$$f\colon \mathbb{N}^n \to \mathbb{N}$$
$$f\ \overline{x} = \mathsf{min}\ y.(g\ \overline{x}\ y)$$

where $\mathsf{min}\ y.(g\ \overline{x}\ y)$ denotes the least natural number y for which $(g\ \overline{x}\ y) = 0$ if this value exists, it is undefined otherwise. Also in this case we use the technique of *accumulating results* by adding a new natural number parameter containing the partial result. We use **Iter** to define a function f^+ on $\mathbb{N}^{n+1}$.

$$\langle \overline{x}, y \rangle \mapsto \begin{cases} \ulcorner y \urcorner & \text{if } (g\ \overline{x}\ y) = 0 \\ \triangleright \langle \overline{x}, y+1 \rangle & \text{otherwise.} \end{cases}$$

This defines a function $f^+\colon \mathbb{N}^{n+1} \to \mathbb{N}$. Now we just put

$$f\colon \mathbb{N}^n \to \mathbb{N}$$
$$f\ \overline{x} = (f^+\ \overline{x}\ 0).$$

$\square$

The definition of a recursive function $f\colon A \to B$ is then given by a map from A to $A+B$, if we write $\triangleright a$ and $\ulcorner b \urcorner$ for the left and right injections, respectively. Such a function is a coalgebra for the functor $F_B X := X + B$. (See [38] and [2] for elementary introductions to coalgebras.) We recall the categorical notion of coalgebra: A coalgebra for an endofunctor F is an object A together with a morphism $u\colon A \to FA$. If the functor F has a final coalgebra νF, $\gamma\colon \nu F \to F\nu F$, then all coalgebras can be embedded in it. This means that if A and u constitute an F-coalgebra, then there is a unique morphism $\hat{u}\colon A \to \nu F$ such that $\gamma \circ \hat{u} = F\hat{u} \circ u$. In type theory, final coalgebras are modeled by coinductive types. Let then B^ν be the coinductive type associated to F_B. Then, as we have said, every coalgebra $u\colon A \to A + B$ defines a function $\hat{u}\colon A \to B^\nu$. In conclusion, we can model partial recursive functions from a type A to a type B in type theory by elements of $A \to B^\nu$.

The rest of the paper formalizes this idea and compares it with other formalizations of partial recursive functions.

Section 2 gives a succinct introduction to type theory, with special emphasis on recursive (inductive and coinductive) types. Section 3 presents the type constructor for partial elements, the formal definition of convergence and divergence, and the formal proofs of basic properties. Section 4 contains the representation of general recursive functions in type theory using the type for partial elements. Section 5 tackles the insidious problem of the representation of nested recursive functions. Section 6 shows how to construct least fixed points of function operators. Section 7 discusses a variant of the construct that supports a lazy evaluation order. Finally, Section 8 proves that the constructor for partial elements is a strong monad.

The whole development has been completely formalized using the proof assistant Coq [57], version 8. Specifically, every numbered definition has been formalized and all numbered lemmas and theorems have been proved formally; with the exception of the content of Section 4, that we can see as a consequence of the the more abstract results of Section 6. The main results, both in this article and in the Coq development, are Theorems 6.19 and 6.20. The examples of nested recursive functions of Sections 5 have also been programmed in Coq. The file of the formalization is available on the author's web site:

http://www.science.uottawa.ca/~vcapr396/

2. Type Theory and Recursive Types

We work in a dependent type theory with inductive and coinductive types. If you are familiar with this kind of system, you can skip this section. See [4], [5], or [6] for a good introduction to dependent type theory. We work in a system $\lambda_{\mathbf{P}}^{\mu\nu}$ consisting of the the pure type system $\lambda\mathbf{P}$ with the addition of sum types and (co)inductive types. We use a formulation of (co)inductive types as recursive sum types, as in [30].

We use some simplifying notation to make the treatment more intuitive and concise. We write both Type and Prop for the sort of small types, usually denoted by $*$ in $\lambda\mathbf{P}$, to distinguish computational from logical types and to facilitate applications to other type systems in which these sorts are distinguished. If Γ is a valid context and Δ is a sequence of assignments of types to variables, we write $\Gamma \vdash \Delta$ *valid* to express that Γ, Δ is also a valid context. We denote by $\overline{x_\Delta}$ the sequence of variables assumed in Δ.

If $\Delta \equiv x_1 \colon A_1, \ldots, x_m \colon A_m$ and $\overline{d} \equiv d_1, \ldots, d_m$ is a sequence of pseudo-terms, then the judgment $\Gamma \vdash \overline{d} \colon\colon \Delta$ expresses the conjunction of the judgments

$$\Gamma \vdash d_1 \colon A_1,$$
$$\Gamma \vdash d_2 \colon A_2[d_1/x_1],$$
$$\vdots$$
$$\Gamma \vdash d_m \colon A_m[d_1/x_1, \ldots, d_{m-1}/x_{m-1}].$$

If $\Gamma, \Delta \vdash t \colon T$ and $\Gamma \vdash \overline{d} \colon\colon \Delta$, we write $t[\overline{d}]$ for $t[d_1/x_1, \ldots, d_m/x_m]$, leaving it implicit that we are substituting the last n variables in the context of t. To avoid confusion, we may sometimes introduce the term t as $t[x_1, \ldots, x_n]$..

If $\Gamma, \Delta \vdash B \colon$ Type, we write $\Pi\Delta.B$ for the successive product of B over each of the variables in Δ: $\Pi\Delta.B \equiv \Pi x_1 \colon A_1.\Pi x_2 \colon A_2 \ldots \Pi x_n \colon A_n.B$. We also use the notation $(\Delta)B$ for $\Pi\Delta.B$.

We add constant definition to the system, that is, if $\Gamma \vdash e \colon T$, then we can add a declaration of the form $\mathsf{t} := e$ to the context, with the typing rule $\Gamma, \mathsf{t} := e \vdash \mathsf{t} \colon T$ and the reduction rule $\mathsf{t} \leadsto_\delta e$. A constant name can be declared at most once. The classes of names for variables and constants are disjoint. However, sometimes we use the same name in different fonts to denote a variable and a constant. It will then be understood that the variable is automatically substituted with the constant whenever the latter is defined in the context. For example, if we have $\Gamma, t \colon T \vdash u \colon U$ and $\Gamma \vdash e \colon T$, we write $\Gamma, \mathsf{t} := e \vdash u \colon U$ for $\Gamma, \mathsf{t} := e \vdash u[\mathsf{t}/t] \colon U$.

Sum types are specified by a sequence of constructors with arguments of previously defined types.

Definition 2.1. Let Γ be a valid context and assume that the following judgments are derivable:

$$\Gamma \vdash \Delta \ valid,$$
$$\Gamma \vdash \Theta_i \ valid,$$
$$\Gamma, \Theta_i \vdash \overline{p_i} \colon\colon \Delta$$

for $1 \le i \le n$. Let T and $\mathsf{c}_1, \ldots, \mathsf{c}_n$ be new constant names. Then we can add the definition of the sum type

$$\mathsf{Sum}\ \mathsf{T}\ [\Delta] \colon \mathsf{Type} \ := \begin{cases} \mathsf{c}_1 \colon (\Theta_1)(\mathsf{T}\ \overline{p_1}) \\ \vdots \\ \mathsf{c}_n \colon (\Theta_n)(\mathsf{T}\ \overline{p_n}) \end{cases}$$

in the context Γ. Let Γ' be the extension of Γ with this sum type definition. We have the following rules for T:

Formation:

$$\frac{\Gamma' \vdash \overline{d} :: \Delta}{\Gamma' \vdash (\mathsf{T}\ \overline{d}) \colon \mathsf{Type}};$$

Introduction:

$$\frac{\Gamma' \vdash \overline{a} :: \Theta_i}{\Gamma' \vdash (\mathsf{c}_i\ \overline{a}) \colon (\mathsf{T}\ \overline{p_i[\overline{a}]})} \quad \text{for } 1 \leq i \leq n;$$

Elimination (definition by cases):

$$\frac{\begin{array}{l} \Gamma' \vdash P \colon (\Delta)(\mathsf{T}\ \overline{x_\Delta}) \to \mathsf{Type} \\ \Gamma', \Theta_i \vdash e_i \colon (P\ \overline{p_i}\ (\mathsf{c}_i\ \overline{x_{\Theta_i}})) \quad 1 \leq i \leq n \\ \Gamma' \vdash \overline{d} :: \Delta \\ \Gamma' \vdash t \colon (\mathsf{T}\ \overline{d}) \end{array}}{\Gamma' \vdash \left(\mathsf{Cases}\ t\ \mathsf{of} \left\{ \begin{array}{l} (\mathsf{c}_1\ \overline{x_{\Theta_1}}) \mapsto e_1 \\ \vdots \\ (\mathsf{c}_n\ \overline{x_{\Theta_n}}) \mapsto e_n \end{array} \right. \right) \colon (P\ \overline{d}\ t)};$$

Reduction:

$$\left(\mathsf{Cases}\ (\mathsf{c}_i\ \overline{a})\ \mathsf{of} \left\{ \begin{array}{l} (\mathsf{c}_1\ \overline{x_{\Theta_1}}) \mapsto e_1 \\ \vdots \\ (\mathsf{c}_n\ \overline{x_{\Theta_n}}) \mapsto e_n \end{array} \right. \right) \rightsquigarrow e_i[\overline{a}/\overline{x_{\Theta_i}}] \quad \text{for } 1 \leq i \leq n.$$

When a function is defined by case analysis, we use a *pattern matching* notation to facilitate reading. A function

$$\mathsf{f} \colon (\Delta)(t \colon (\mathsf{T}\ \overline{x_\Delta}))(P\ \overline{x_\Delta}\ t)$$

$$\mathsf{f} \equiv \lambda \overline{x_\Delta} :: \Delta. \lambda t \colon (\mathsf{T}\ \overline{x_\Delta}). \mathsf{Cases}\ t\ \mathsf{of} \left\{ \begin{array}{l} (\mathsf{c}_1\ \overline{x_{\Theta_1}}) \mapsto e_1 \\ \vdots \\ (\mathsf{c}_n\ \overline{x_{\Theta_n}}) \mapsto e_n \end{array} \right.$$

is written

$$\mathsf{f} \colon (\Delta)(t \colon (\mathsf{T}\ \overline{x_\Delta}))(P\ \overline{x_\Delta}\ t)$$
$$\mathsf{f}\ \overline{p_1}\ (\mathsf{c}_1\ \overline{x_{\Theta_1}}) = e_1$$
$$\vdots$$
$$\mathsf{f}\ \overline{p_n}\ (\mathsf{c}_n\ \overline{x_{\Theta_n}}) = e_n.$$

We formulate both inductive and coinductive types as recursive sum types. By this we mean that the type T may occur in the types of arguments of the constructors Θ_i. The only restriction is that these occurrences must be strictly positive. See [25, 30] for the notion of strictly positive occurrence.

Inductive and coinductive types were introduced in typed λ-calculus by Hagino in [33, 34] with the name of *categorical data types*. In [61] and [27] their expression in polymorphic typed λ-calculus was studied.

In our formulation, inductive and coinductive types have the same rules as sum types plus a fixpoint rule for inductive types and a cofixpoint rule for coinductive types allowing the definition of recursive functions.

Inductive types were introduced in dependent type theory in [25] (see also [51], [49], and [59]). Coinductive types were studied in [42, 43, 23]. The version given here comes from [30] and is the one implemented in the proof tool Coq [56].

Definition 2.2. We make the same assumptions as in the definition of sum types, except that

$$\Gamma, T\colon (\Delta)\mathsf{Type} \vdash \Theta_i \ valid \qquad \Gamma, T\colon (\Delta)\mathsf{Type}, \Theta_i \vdash \overline{p_i}\colon\colon \Delta$$

with T occurring only strictly positively in Θ_i, for $1 \leq i \leq n$ (see [30], Section 2.2, pg.43, for the definition of strict positivity). Let T and c_1, ..., c_n be new constant names. Then we can add the definition of the inductive type

$$\mathsf{Inductive}\ \mathsf{T}\ [\Delta]\colon \mathsf{Type}\ := \left\{ \begin{array}{l} \mathsf{c}_1\colon (\Theta_1)(\mathsf{T}\ \overline{p_1}) \\ \vdots \\ \mathsf{c}_n\colon (\Theta_n)(\mathsf{T}\ \overline{p_n}) \end{array} \right.$$

in the context Γ. Let Γ' be the extension of Γ with this inductive type definition. We have the same rules as for sum types, plus the following:

Fixpoint:

$$\frac{\Gamma' \vdash P\colon (\Delta)(\mathsf{T}\ \overline{x_\Delta}) \to \mathsf{Type} \quad \Gamma', f\colon (\Delta)(t\colon (\mathsf{T}\ \overline{x_\Delta}))(P\ \overline{x_\Delta}\ t), \Delta, t\colon (\mathsf{T}\ \overline{x_\Delta}) \vdash e\colon (P\ \overline{x_\Delta}\ t)}{\Gamma' \vdash (\mathsf{fix}\ [f, \overline{x_\Delta}, t]e)\colon (\Delta)(t\colon (\mathsf{T}\ \overline{x_\Delta}))(P\ \overline{x_\Delta}\ t)} \mathcal{D}\{f, t, e\}$$

where $\mathcal{D}\{f, t, e\}$ is a side condition defined in [30], Section 3.1, pg. 47;

Reduction:

$$(\mathsf{f}\ \overline{d}\ (\mathsf{c}_i\ \overline{a})) \rightsquigarrow e[\mathsf{f}, \overline{d}, (\mathsf{c}_i\ \overline{a})]$$

where $\mathsf{f} \equiv (\mathsf{fix}\ [f, \overline{x_\Delta}, t]e)$.

Definition 2.3. We make the same assumptions as in the definition of inductive types. Let T and c_1, ..., c_n be new constant names. Then we can add the definition of the coinductive type

$$\mathsf{CoInductive}\ \mathsf{T}\ [\Delta]\colon \mathsf{Type}\ := \left\{ \begin{array}{l} \mathsf{c}_1\colon (\Theta_1)(\mathsf{T}\ \overline{p_1}) \\ \vdots \\ \mathsf{c}_n\colon (\Theta_n)(\mathsf{T}\ \overline{p_n}) \end{array} \right.$$

in the context Γ. Let Γ' be the extension of Γ with this coinductive type definition. We have the same rules as for sum types, plus the following, where Ξ and Γ'' are valid context extensions of Γ':

Cofixpoint:

$$\frac{\Gamma', \Xi \vdash \overline{u}\colon\colon \Delta \quad \Gamma', f\colon (\Xi)(\mathsf{T}\ \overline{u}) \vdash e\colon (\Xi)(\mathsf{T}\ \overline{u})}{\Gamma' \vdash (\mathsf{cofix}\ [f]e)\colon (\Xi)(\mathsf{T}\ \overline{u})} \mathcal{C}\{f, e\}$$

where $\mathcal{C}\{f, e\}$ is a side condition defined in [30], Section 4.1, pg. 53;

Reduction: if $\Gamma', \Gamma'' \vdash \overline{v}\colon\colon \Xi$, then

$$\left(\mathsf{Cases}\ (\mathsf{f}\ \overline{v})\ \mathsf{of} \left\{ \begin{array}{l} (\mathsf{c}_1\ \overline{x_{\Theta_1}}) \mapsto e_1 \\ \vdots \\ (\mathsf{c}_n\ \overline{x_{\Theta_n}}) \mapsto e_n \end{array} \right. \right) \rightsquigarrow \left(\mathsf{Cases}\ (e[\mathsf{f}]\ \overline{v})\ \mathsf{of} \left\{ \begin{array}{l} (\mathsf{c}_1\ \overline{x_{\Theta_1}}) \mapsto e_1 \\ \vdots \\ (\mathsf{c}_n\ \overline{x_{\Theta_n}}) \mapsto e_n \end{array} \right. \right)$$

where $\mathsf{f} \equiv (\mathsf{cofix}\ [f]e)$.

We refer to [30] for the definition of the side conditions $\mathcal{D}$ and $\mathcal{C}$ of the fixpoint and cofixpoint rules. Let us give only an intuitive reminder of their meaning. The side condition $\mathcal{D}\{f, t, e\}$ guarantees that whenever the fixpoint function is applied to an argument in the inductive type, it performs recursive calls only on structurally smaller objects. The side condition $\mathcal{C}\{f, e\}$ guarantees that whenever the cofixpoint function is applied to obtain an element of the coinductive type, this element is *productive* [23], that is, it can be reduced to constructor form.

We will often use a simpler notation for fixed and cofixed points. We write $\mathsf{f} = e[\mathsf{f}]$ for $\mathsf{f} \equiv (\mathsf{fix} \ [f, \overline{x_\Delta}, t]e)$ or $\mathsf{f} \equiv (\mathsf{cofix} \ [f]e)$. Which one of the two is meant will be clear from the type. There can be ambiguity when the domain of f is an inductive type and its codomain is a coinductive type. In that case we state explicitly whether a fixed or cofixed point is meant.

When coinductive types are considered categorically as the dual of inductive types, their formulation is different from the one given here. They are not recursive sum types like the inductive ones, but rather recursive record types, since records are the dual of sums. This is the way they are presented in [27]. Our formulation conforms to the intuitive conception of coinductive types as types of possibly infinite elements. Objects of coinductive types are built up by constructors as those of inductive types, but while inductive objects must be wellfounded, coinductive ones may be infinitely deep. This formulation is given in [23]. The equivalence of the two formulations is proved in [30].

3. COINDUCTIVE TYPES OF PARTIAL ELEMENTS

For every type A, we define a coinductive type whose elements can be thought of as possibly undefined elements of A.

Definition 3.1. Let A be a type, that is, $\Gamma \vdash A \colon \mathsf{Type}$. Then we define

$$\mathsf{CoInductive} \ A^\nu \colon \mathsf{Type} \ := \begin{cases} \mathsf{return} \colon A \to A^\nu \\ \mathsf{step} \colon A^\nu \to A^\nu. \end{cases}$$

We use the notation $\ulcorner a \urcorner$ for $(\mathsf{return} \ a)$ and $\rhd x$ for $(\mathsf{step} \ x)$.

Intuitively, an element of A^ν is either an element of A or a computation step followed by an element of A^ν. Since in coinductive types it is possible to define infinite elements, there is an object $\rhd \rhd \rhd \cdots$ denoting an infinite computation. Formally, it is defined as

$$\rhd^\infty = \rhd \rhd^\infty \colon A^\nu.$$

We want to identify all terms of the form $\rhd^k \ulcorner a \urcorner$ as equivalent representations of the element a. We do it by defining an equality relation on A^ν that is a strengthening of bisimulation (see, for example, [44, 1, 10, 36]). First we define, inductively, when an element of A^ν converges to a value in A and, coinductively, when it diverges.

Definition 3.2.

$$\mathsf{Inductive \ Value} \ [x \colon A^\nu, a \colon A] \colon \mathsf{Prop} \ :=$$
$$\begin{cases} \mathsf{value\_return} \colon (a \colon A)(\mathsf{Value} \ \ulcorner a \urcorner \ a) \\ \mathsf{value\_step} \colon (x \colon A^\nu; a \colon A)(\mathsf{Value} \ x \ a) \to (\mathsf{Value} \ \rhd x \ a) \end{cases}$$

$$\mathsf{CoInductive \ Diverge} \ [x \colon A^\nu] \colon \mathsf{Prop} \ :=$$
$$\begin{cases} \mathsf{diverge} \colon (x \colon A^\nu)(\mathsf{Diverge} \ x) \to (\mathsf{Diverge} \ \rhd x) \end{cases}$$

We use the notation $x \downarrow a$ for $(\mathsf{Value} \ x \ a)$ and $x^\uparrow$ for $(\mathsf{Diverge} \ x)$.

Our first example of a proof by coinduction shows that $\triangleright^{\infty}$ diverges. This is trivial, but we give the proof to illustrate the style of derivation that is used to prove coinductive predicates.

Lemma 3.3. $(\triangleright^{\infty})^{\uparrow}$.

Proof. If we think of coinductive definitions as representing infinite objects, coinductive predicates are proved by infinite proofs. The infinite proof showing the truth of the statement is

$$H = (\mathsf{diverge}\ \triangleright^{\infty}\ (\mathsf{diverge}\ \triangleright^{\infty}\ \cdots)) \colon (\triangleright^{\infty})^{\uparrow}.$$

It consists of an infinite sequence of nested applications of the constructor $\mathsf{diverge}$, always with $\triangleright^{\infty}$ as first argument. At first sight this may seem a circular proof that doesn't prove anything. However, if we analyze it, we discover that it actually establishes what it alleges. Let us give names to the subterms of $x_0 = \triangleright^{\infty}$:

$$x_0 = \triangleright x_1 \quad x_1 = \triangleright x_2 \quad x_2 = \triangleright x_3 \quad \cdots.$$

Of course, they all coincide: $x_0 = x_1 = x_2 = x_3 = \cdots = \triangleright^{\infty}$. Let us call H_i the proof of $x_i^{\uparrow}$. Again, it will be clear that we can use the same proof for every H_i. The first step of the proof constructs H_0 from H_1:

$$H_0 = (\mathsf{diverge}\ x_1\ H_1).$$

This just says that if x_1 diverges, then x_0 must also diverge because it is obtained by adding one $\triangleright$ to x_1. In turn the proof H_1 is constructed by

$$H_1 = (\mathsf{diverge}\ x_2\ H_2)$$

and so on, producing the infinite proof, which now doesn't seem purposeless anymore: Every occurrence of $\mathsf{diverge}$ shows that the term contains a separate $\triangleright$ constructor.

In practice, we will define this proof H in terms of itself:

$$H = (\mathsf{diverge}\ \triangleright^{\infty}\ H).$$

Formally, this definition is valid because the recursive occurrence of H is an argument of $\mathsf{diverge}$, that is, in coinductive jargon, H is *guarded* by $\mathsf{diverge}$. This guarantees that H can be expanded to an infinite proof not containing any occurrence of H itself. $\square$

Intuitively, A^{ν} contains copies of the elements of A plus a diverging element $\triangleright^{\infty}$. However, A^{ν} is not isomorphic to $A + 1$, where 1 is the one element type, because divergence is undecidable. In fact, the proposition

$$\forall x \colon A^{\nu}.(\exists a \colon A.x \downarrow a) \vee x^{\uparrow}$$

is not provable, while the corresponding

$$\forall x \colon A + 1.(\exists a \colon A.x = (\mathsf{inl}\ a)) \vee x = (\mathsf{inr}\ 0_1)$$

is trivially true by case analysis. However, a different property, classically equivalent to the above decidability property, is constructively provable.

Theorem 3.4.

$$\forall x \colon A^{\nu}.(\neg \exists a \colon A.x \downarrow a) \to x^{\uparrow}.$$

Proof. This is our first non-trivial proof by coinduction, so we take care of doing it in detail and explain how such proofs work. We know that a coinductive object can be defined in terms of itself, as long as the definition is *productive,* that is, it can generate leading constructors to an arbitrary depth. This is true for coinductive proofs as well, leading to the apparently paradoxical fact that we can assume the statement that we want to prove, provided that we use it in such a way that every step of the (infinite) proof can be generated.

Let us see how this works specifically in our case. We give the name H to the proof that we are constructing:

$$H : \forall x \colon A^\nu . (\neg \exists a \colon A.x \downarrow a) \to x^\uparrow .$$

We now describe how this function is constructed. Let x be an element of A^ν. By case analysis we know that either $x = \ulcorner a \urcorner$ for some $a \colon A$ or $x = \triangleright x'$ for some other $x' \colon A^\nu$. We consider separately the two cases.

If $x = \ulcorner a \urcorner$, then trivially $\exists a \colon A.x \downarrow a$. The premise of the statement is then false and, therefore, the implication is trivially true.

If, on the other hand, $x = \triangleright x'$, let us assume that there is no $a \colon A$ such that $x \downarrow a$. We have to prove that x diverges. The constructor diverge allows us to deduce this, if we can prove that x' diverges. Now we apply the proof H to x': If we can prove $\neg \exists a \colon A.x' \downarrow a$, then we can conclude that x' diverges. But this assertion is true because, if there were an a such that $x' \downarrow a$, then, by value_step, also $x \downarrow a$ against the assumption.

In conclusion, H allows us to deduce that x' diverges and so, by diverge, that x diverges too, as desired. We used the proof H inside the definition of H itself; isn't this circular? No, because we used it to prove an hypothesis $(x'^\uparrow)$ that was generated by diverge. In the coinduction jargon, the occurrence of H is *guarded* by the occurrence of diverge.

If you want to convince yourself of the soundness of this proof style, consider what concrete proofs are generated by H in specific cases. For a converging element $x = \triangleright \triangleright \cdots \triangleright \ulcorner a \urcorner$, the proof is $(H\ x)$. Let us expand it. Since $x = \triangleright x'$ for $x' = \triangleright \cdots \triangleright \ulcorner a \urcorner$, the second case of the proof applies:

$$(H\ x) = (\mathsf{diverge}\ x'\ (H\ x')).$$

In turn, the proof $(H\ x')$ reduces similarly, and so on, until we reach the proof

$$(H\ x) = (\mathsf{diverge}\ x'\ (\mathsf{diverge}\ x''\ \cdots (\mathsf{diverge}\ \ulcorner a \urcorner\ (H\ \ulcorner a \urcorner)) \cdots)).$$

Finally, $(H\ \ulcorner a \urcorner)$ trivially reduces to a proof by *ex falso quodlibet* as described in the first case, in which H does not occur.

On the other hand, if $x = \triangleright^\infty$, the described reduction procedure of $(H\ x)$ continues indefinitely, generating the infinite proof

$$(H\ \triangleright^\infty) = (\mathsf{diverge}\ \triangleright^\infty\ (\mathsf{diverge}\ \triangleright^\infty\ \cdots))$$

of Lemma 3.3. This proof also does not contain any occurrence of H. $\square$

From now on, proofs of coinductive properties are less prolix. We take it for granted that we can assume the validity of the statement that we are proving. In case of doubt, it can always be checked, intuitively, that the proofs are productive or, formally, that they satisfy the condition $\mathcal{C}\{f, e\}$. The correctness of the proofs has been checked formally using the proof assistant Coq.

Now all the elements are in place for the definition of equality of partial elements: It is also a coinductive relation.

Definition 3.5. Let A be a type. We define equality on A^ν by

$$\mathsf{CoInductive\ Eq}_A^\nu\ [x, y\colon A^\nu]\colon \mathsf{Prop}\ :=$$
$$\left\{\begin{array}{l} \mathsf{eq\_value}\colon (x, y\colon A^\nu; a\colon A)x \downarrow a \to y \downarrow a \to (\mathsf{Eq}_A^\nu\ x\ y) \\ \mathsf{eq\_step}\colon (x, y\colon A^\nu)(\mathsf{Eq}_A^\nu\ x\ y) \to (\mathsf{Eq}_A^\nu\ \triangleright x\ \triangleright y) \end{array}\right.$$

We use the notation $x \overset{\nu}{=} y$ for $(\mathsf{Eq}_A^\nu\ x\ y)$.

Note that, since convergence and divergence are in general undecidable, the equality $x \overset{\nu}{=} y$ is not equivalent to the proposition

$$(\exists a\colon A.x \downarrow a \wedge y \downarrow a) \vee (x^\uparrow \wedge y^\uparrow).$$

A series of results can be proved easily, stating that the defined operations, relations, and predicates behave as expected.

Lemma 3.6. *For all $x, y\colon A^\nu$ and $a, a_1, a_2\colon A$, we have*

1. $x \downarrow a \to y \downarrow a \to x \overset{\nu}{=} y$ 2. $x^\uparrow \to y^\uparrow \to x \overset{\nu}{=} y$

3. $x \downarrow a \to x \overset{\nu}{=} y \to y \downarrow a$ 4. $x^\uparrow \to x \overset{\nu}{=} y \to y^\uparrow$

5. $x \downarrow a \to x \overset{\nu}{=} \ulcorner a \urcorner$ 6. $x \overset{\nu}{=} \ulcorner a \urcorner \to x \downarrow a$

7. $\triangleright x \downarrow a \to x \downarrow a$ 8. $\triangleright x^\uparrow \to x^\uparrow$

9. $x \downarrow a \to \neg x^\uparrow$ 10. $x \downarrow a \to y^\uparrow \to \neg x \overset{\nu}{=} y$

11. $x \overset{\nu}{=} y \to x \overset{\nu}{=} \triangleright y$ 12. $\triangleright x \overset{\nu}{=} y \to x \overset{\nu}{=} y$

13. $\ulcorner a_1 \urcorner \downarrow a_2 \to a_1 = a_2$ 14. $x \overset{\nu}{=} x$

15. $x \overset{\nu}{=} y \to y \overset{\nu}{=} x$ 16. $x \overset{\nu}{=} y \to y \overset{\nu}{=} z \to x \overset{\nu}{=} z$

Proof.   1. By $\mathsf{eq\_value}$.

2. Proof by coinduction. Assume H is the proof of the statement:

$$H\colon \forall x, y\colon A^\nu.x^\uparrow \to y^\uparrow \to x \overset{\nu}{=} y.$$

Let x and y be diverging elements. Then they must be in the form $x = \triangleright x'$ and $y = \triangleright y'$, where x' and y' are also diverging. We apply the proof H to x' and y' to obtain $x' \overset{\nu}{=} y'$. By $\mathsf{eq\_step}$ we also have $x \overset{\nu}{=} y$. The recursive application of H is guarded by $\mathsf{eq\_step}$.

5. Assume $x \downarrow a$. By $\mathsf{value\_return}$ we have that $\ulcorner a \urcorner \downarrow a$. Thus, by $\mathsf{eq\_value}$ we can conclude that $x \overset{\nu}{=} \ulcorner a \urcorner$.

13. Assume that $\ulcorner a_1 \urcorner \downarrow a_2$. Any proof H of this statement must be constructed by using $\mathsf{value\_return}$, since the only other constructor $\mathsf{value\_step}$ gives a conclusion of the wrong form, that is, with a $\triangleright$ constructor in the first argument. Therefore, it must be

$$H = (\mathsf{value\_return}\ a')$$

for some a'. But, since $(\mathsf{value\_return}\ a')$ is a proof of $\ulcorner a' \urcorner \downarrow a'$, we may conclude that $a' = a_1 = a_2$. This is an example of *proof by inversion*; it consists in proving a goal by reasoning on the possible form of an assumption.

6. Assume that $x \overset{\nu}{=} \ulcorner a \urcorner$. We proceed again by inversion on this assumption. If H is a proof of this premise, it must have been obtained by an application of $\mathsf{eq\_value}$,

since the other constructor eq_step gives a conclusion with the wrong form, that is, with a $\triangleright$ constructor in the second argument. Therefore, it must be

$$H = (\textsf{eq\_value } x' \ y' \ a' \ h_1 \ h_2)$$

for some $x', y' \colon A^\nu$, $a' \colon A$, h_1 a proof of $x' \downarrow a'$, and h_2 a proof of $y' \downarrow a'$. But this is a proof of $x' \overset{\nu}{=} y'$, so we can conclude that $x' = x$ and $y' = \ulcorner a \urcorner$. Then h_2 is a proof of $\ulcorner a \urcorner \downarrow a'$, from which we deduce, by point 13, that $a = a'$. But then h_1 is a proof of $x \downarrow a$, and the conclusion is established.

14. By coinduction: Assume that H is the proof of : $\forall x \colon A^\nu.x \overset{\nu}{=} x$. Any element x must be of one of the two forms: $x = \ulcorner a' \urcorner$ or $x = \triangleright x'$.

    If $x = \ulcorner a' \urcorner$, we must prove that $\ulcorner a' \urcorner \overset{\nu}{=} \ulcorner a' \urcorner$. By value_return we have that $\ulcorner a' \urcorner \downarrow a'$, so, by eq_value, we have that $\ulcorner a' \urcorner \overset{\nu}{=} \ulcorner a' \urcorner$.

    If $x = \triangleright x'$ we must prove that $\triangleright x' \overset{\nu}{=} \triangleright x'$. By H we have that $x' \overset{\nu}{=} x'$, so, by eq_step, we have the conclusion. The application of H is guarded by eq_step.

15. By coinduction: Assume that H is the proof of $\forall x, y \colon A^\nu.x \overset{\nu}{=} y \to y \overset{\nu}{=} x$. Assume $x \overset{\nu}{=} y$ and let h be a proof of this premise. We proceed by inversion on the proof h. It must have been obtained by applying one of the two constructors eq_value or eq_step.

    If $h = (\textsf{eq\_value } x \ y \ a' \ h_1 \ h_2)$, with h_1 a proof of $x \downarrow a'$ and h_2 a proof of $y \downarrow a'$; then we just have to apply the same constructor with inverted arguments to obtain the desired goal: $(\textsf{eq\_value } y \ x \ a' \ h_2 \ h_1)$ is a proof of $y \overset{\nu}{=} x$.

    If $h = (\textsf{eq\_step } \triangleright x' \ \triangleright y' \ h')$, with $x = \triangleright x'$, $y = \triangleright y'$, and h' a proof of $x' \overset{\nu}{=} y'$; then we apply H to x', y', and h' to obtain $y' \overset{\nu}{=} x'$, and then we have $y' = \triangleright y' \overset{\nu}{=} \triangleright x' = x$ by eq_step. The application of H is guarded by eq_step.

7. By inversion on the proof of the premise $\triangleright x \downarrow a$, which can only have been obtained by using value_step on a proof of $x \downarrow a$.

11. By coinduction: Assume that H is the proof of $\forall x, y \colon A^\nu.x \overset{\nu}{=} y \to x \overset{\nu}{=} \triangleright y$. Assume $x \overset{\nu}{=} y$ and let h be a proof of this premise. We proceed by inversion on h.

    If $h = (\textsf{eq\_value } x \ y \ a' \ h_1 \ h_2)$, with h_1 a proof of $x \downarrow a'$ and h_2 a proof of $y \downarrow a'$; then, by value_step, from h_2 we can deduce $\triangleright y \downarrow a'$. By eq_value, we conclude that $x \overset{\nu}{=} \triangleright y$.

    If $h = (\textsf{eq\_step } \triangleright x' \ \triangleright y' \ h')$, with $x = \triangleright x'$, $y = \triangleright y'$, and h' a proof of $x' \overset{\nu}{=} y'$; then we apply H to x', y', and h' to obtain $x' \overset{\nu}{=} \triangleright y'$. By eq_step we then have that $x = \triangleright x' \overset{\nu}{=} \triangleright \triangleright y' = \triangleright y'$. The application of H is guarded by eq_step.

12. By inversion on the proof of the premise $\triangleright x \overset{\nu}{=} y$ using point 7 in case eq_value was used and point 11 in case eq_step was used.

3. Assume that $x \downarrow a$ and let h be a proof of this premise. We proceed by induction on the structure of h.

    If $h = (\textsf{value\_return } a) \colon \ulcorner a \urcorner \downarrow a$ then we have to prove that if $\ulcorner a \urcorner \overset{\nu}{=} y$ then $y \downarrow a$. By point 15, $y \overset{\nu}{=} \ulcorner a \urcorner$ and, by point 6, $y \downarrow a$.

    If $h = (\textsf{value\_step } x' \ a \ h')$ with $x = \triangleright x'$ and h a proof of $x' \downarrow a$, we know by induction hypothesis that $\forall y \colon A^\nu.x' \overset{\nu}{=} y \to y \downarrow a$. Assume now that $x \overset{\nu}{=} y$, that is, $\triangleright x' \overset{\nu}{=} y$. By point 12 we can deduce that $x' \overset{\nu}{=} y$, and therefore, by induction hypothesis, that $y \downarrow a$.

8. By inversion on the proof of the premise $\triangleright x^\uparrow$.

9. By induction on the proof h of the premise $x \downarrow a$.

If $h = (\mathsf{value\_return}\ a)$, with $x = \ulcorner a \urcorner$, we have to prove that $\neg \ulcorner a \urcorner^{\uparrow}$. This is true because it is impossible to build a proof of $\ulcorner a \urcorner^{\uparrow}$. If such a proof existed, it should be obtained by an application of the only constructor $\mathsf{diverge}$ of the predicate $\mathsf{Diverge}$. But such a proof $(\mathsf{diverge}\ x\ h)$ can only prove statements in the form $\rhd x^{\uparrow}$. Since in our case the argument of $\mathsf{Diverge}$ is $\ulcorner a \urcorner$, we reach a contradiction. This proof is a degenerate case of proof by inversion: By analyzing the possible form of the proof we conclude that no form is possible.

If $h = (\mathsf{value\_step}\ x'\ a\ h')$ with $x = \rhd x'$ and h a proof of $x' \downarrow a$, we know by induction hypothesis that $\neg x'^{\uparrow}$. To prove that $\neg x^{\uparrow}$, assume that $x^{\uparrow}$, that is $\rhd x'^{\uparrow}$. By point 8 we then have that $x'^{\uparrow}$, against the induction hypothesis. Having reached a contradiction, we conclude that $\neg x^{\uparrow}$.

4. By coinduction: Assume that H is the proof of $\forall x, y \colon A^{\nu}. x^{\uparrow} \to x \stackrel{\nu}{=} y \to y^{\uparrow}$. Assume $x \stackrel{\nu}{=} y$ and let h be a proof of this assumption. We must prove that $x^{\uparrow} \to y^{\uparrow}$. We proceed by inversion on h.

Suppose $h = (\mathsf{eq\_value}\ x\ y\ a'\ h_1\ h_2)$ with h_1 a proof of $x \downarrow a'$ and h_2 a proof of $y \downarrow a'$. If we assume that $x^{\uparrow}$ then we obtain a contradiction by point 9. The conclusion then follows by *ex falso quodlibet*.

Suppose $h = (\mathsf{eq\_step}\ \rhd x'\ \rhd y'\ h')$, with $x = \rhd x'$, $y = \rhd y'$, and h' a proof of $x' \stackrel{\nu}{=} y'$. From the assumption $x^{\uparrow}$, that is $\rhd x'^{\uparrow}$, and point 8 we derive $x'^{\uparrow}$. We apply H to this proof and h' to obtain $y'^{\uparrow}$. By $\mathsf{diverge}$ we then obtain $\rhd y'^{\uparrow}$, that is, $y^{\uparrow}$. The application of H is guarded by $\mathsf{diverge}$.

10. By induction on the proof h of the premise $x \downarrow a$.

If $h = (\mathsf{value\_return}\ a)$, with $x = \ulcorner a \urcorner$, we have to prove that $y^{\uparrow} \to \ulcorner a \urcorner \neq y$. Assume that $y^{\uparrow}$ and $\ulcorner a \urcorner \stackrel{\nu}{=} y$. By point 15, $y \stackrel{\nu}{=} \ulcorner a \urcorner$; by point 6, $y \downarrow a$; by point 9, $\neg y^{\uparrow}$ against the hypothesis. Discharging the second hypothesis, we obtain $\neg \ulcorner a \urcorner \stackrel{\nu}{=} y$.

If $h = (\mathsf{value\_step}\ x'\ a\ h')$ with $x = \rhd x'$ and h a proof of $x' \downarrow a$, we know by induction hypothesis that $y^{\uparrow} \to \neg x' \stackrel{\nu}{=} y$. Assume that $y^{\uparrow}$, so $\neg x' \stackrel{\nu}{=} y$ by induction hypothesis. We have to prove $\neg x \stackrel{\nu}{=} y$, that is, $\neg \rhd x' \stackrel{\nu}{=} y$. Assume that $\rhd x' \stackrel{\nu}{=} y$. By point 12 we then have that $x' \stackrel{\nu}{=} y$, contradicting a previous conclusion. Having reached a contradiction, we conclude that $\neg x \stackrel{\nu}{=} y$.

16. By coinduction: Assume that H is the proof of $\forall x, y, z \colon A^{\nu}. x \stackrel{\nu}{=} y \to y \stackrel{\nu}{=} z \to x \stackrel{\nu}{=} z$. Assume h_1 is a proof of $x \stackrel{\nu}{=} y$ and h_2 is a proof of $y \stackrel{\nu}{=} z$. We proceed by inversion on these proofs: We have three cases.

Suppose $h_1 = (\mathsf{eq\_value}\ x\ y\ a'\ k_1\ k_2)$ with k_1 a proof of $x \downarrow a'$ and k_2 a proof of $y \downarrow a'$. By point 3, from k_2 and h_2 we can derive that $z \downarrow a'$. By k_1 and $\mathsf{eq\_value}$, it follows that $x \stackrel{\nu}{=} z$.

Suppose $h_2 = (\mathsf{eq\_value}\ y\ z\ a'\ k_2\ k_3)$. The proof is similar to the previous case.

Suppose $h_1 = (\mathsf{eq\_step}\ x'\ y'\ h_1')$ with $x = \rhd x', y = \rhd y'$, and h_1' a proof of $x' \stackrel{\nu}{=} y'$; and $h_2 = (\mathsf{eq\_step}\ y'\ z'\ h_2')$ with $y = \rhd y', z = \rhd z'$, and h_2' a proof of $y' \stackrel{\nu}{=} z'$. We apply H to x', y', z' and the proofs h_1' and h_2', obtaining that $x' \stackrel{\nu}{=} z'$. By $\mathsf{eq\_step}$ it follows that $\rhd x' \stackrel{\nu}{=} \rhd z'$, that is, $x \stackrel{\nu}{=} z$. The application of H is guarded by $\mathsf{eq\_step}$. $\qquad\square$

Points 14, 15, and 16 of the previous lemma can be summarized in the following.

Theorem 3.7. *For every type A, Eq_A^{ν} is an equivalence relation on A^{ν}.*

It follows that the pair $\langle A^\nu, \mathsf{Eq}^\nu_A \rangle$ is a total setoid [8].

The *finite* elements of A^ν are those that have a value in A. They can be characterized directly by an inductive predicate.

Definition 3.8.

$$\mathsf{Inductive\ Finite}\ [x\colon A^\nu]\colon \mathsf{Prop}\ :=\ \left\{ \begin{array}{l} \mathsf{finite\_return}\colon (a\colon A)(\mathsf{Finite}\ \ulcorner a \urcorner) \\ \mathsf{finite\_step}\colon (x\colon A^\nu)(\mathsf{Finite}\ x) \to (\mathsf{Finite}\ \rhd x). \end{array} \right.$$

Lemma 3.9. $\forall x\colon A^\nu.(\mathsf{Finite}\ x) \leftrightarrow \exists a\colon A.x \downarrow a$.

Proof. From left to right: by induction on the proof of $(\mathsf{Finite}\ x)$. From right to left: by induction on the proof of $x \downarrow a$. □

Hereafter, we use the notation $x_\downarrow$ for $(\mathsf{Finite}\ x)$.

A partial function from a type A to a type B is seen as a function from A^ν to B^ν. To be precise, we must require that a function $f\colon A^\nu \to B^\nu$ preserves equality: $\forall x_1, x_2\colon A^\nu.x_1 \overset{\nu}{=} x_2 \to (f\ x_1) \overset{\nu}{=} (f\ x_2)$. In other words, f must be a setoid function. Using the notation of [18], this is written

$$\langle A^\nu, \mathsf{Eq}^\nu_A \rangle\ [\to]\ \langle B^\nu, \mathsf{Eq}^\nu_B \rangle.$$

($[\to]$ is the *functionoid* constructors: Given two setoids, it constructs the setoid of functions between the two. Its elements are pairs consisting of a function and a proof that it preserves the setoid equality. The equality of the functionoid is the extensional equality on the first component, that is, the underlying function.) Anyway, we will use the simple definition and leave it to the reader to check that all the functions that we define (with one exception) preserve equality.

A function is called *strict* if it always maps diverging elements to diverging elements. Since a strict function is always determined on the diverging elements and it must preserve equality, its type can be strengthened to $A \to B^\nu$.

The first partial functions that we define are just the lifting of the total functions.

Definition 3.10. If $f\colon A \to B$, we define its *lifting* to the partial elements as

$$\begin{array}{l} f_\nu\colon A^\nu \to B^\nu \\ f_\nu\ \ulcorner a \urcorner = \ulcorner (f\ a) \urcorner \\ f_\nu\ \rhd y = \rhd(f_\nu\ y) \end{array}$$

Lemma 3.11. *For every* $a\colon A$, $(f_\nu\ \ulcorner a \urcorner) \overset{\nu}{=} \ulcorner (f\ a) \urcorner$.

A special case arises when we consider tuples and projections. The types $A^\nu \times B^\nu$ and $(A \times B)^\nu$ are not isomorphic. We want to define a strict version of the projection functions. That is, the projection should diverge whenever either of the components of the tuple diverges. To do this, we first map $A^\nu \times B^\nu$ to $(A \times B)^\nu$:

$$\begin{array}{l} \mathsf{mixed\_pair}\colon A \times B^\nu \to (A \times B)^\nu \\ \mathsf{mixed\_pair}\ \langle a, y \rangle = \mathsf{Cases}\ y\ \mathsf{of}\ \left\{ \begin{array}{l} \ulcorner b \urcorner \mapsto \ulcorner \langle a, b \rangle \urcorner \\ \rhd y' \mapsto \rhd(\mathsf{mixed\_pair}\ \langle a, y' \rangle) \end{array} \right. \end{array}$$

$$\begin{array}{l} \mathsf{strict\_pair}\colon A^\nu \times B^\nu \to (A \times B)^\nu \\ \mathsf{strict\_pair}\ \langle x, y \rangle = \mathsf{Cases}\ x\ \mathsf{of}\ \left\{ \begin{array}{l} \ulcorner a \urcorner \mapsto (\mathsf{mixed\_pair}\ \langle a, y \rangle) \\ \rhd x' \mapsto \rhd(\mathsf{strict\_pair}\ \langle x', y \rangle) \end{array} \right. \end{array}$$

The function strict_pair just moves all the steps outside the pairing constructor. Iterating strict_pair n times, we obtain $\text{strict\_tuple}_n \colon (A^\nu)^n \to (A^n)^\nu$. We use the notation $\lhd x_1, \ldots, x_n \rhd$ for $(\text{strict\_tuple}_n \langle x_1, \ldots, x_n \rangle)$. The *strict projections* are then defined by

$$\pi_i^\nu \colon (A^\nu)^n \to A^\nu$$
$$\pi_i^\nu = (\pi_i)_\nu \circ \text{strict\_tuple}_n$$

where $\pi_i \colon A^n \to A$ is the standard projection. Note that $(\pi_i^\nu \lhd x_1, \ldots, x_n \rhd) = x_i$ if and only if all the x_js converge or x_i diverges.

4. Representation of partial recursive functions

We show that every partial recursive function $f \colon \mathbb{N}^n \rightharpoonup \mathbb{N}$ can be implemented in type theory as $f^\nu \colon (\mathbb{N}^\nu)^n \to \mathbb{N}^\nu$. We use the notation $A \rightharpoonup B$ to denote partial recursive functions from A to B, that is, any function obtained from the base functions zero, successor, and projections, by composition, primitive recursion, and minimization. Let us define f^ν for every basic function and function-forming operations (see [52] or any other generic introduction to recursion theory).

Definition 4.1. By recursion on the definition of a partial recursive function $f \colon \mathbb{N}^n \rightharpoonup \mathbb{N}$, we define its type-theoretic version $f^\nu \colon (\mathbb{N}^\nu)^n \to \mathbb{N}^\nu$.

Zero: The zero function is the constant 0:

$$\underline{0} \colon \mathbb{N} \to \mathbb{N}$$
$$\underline{0}\, x = 0.$$

Since $\underline{0}$ is a total function definable in type theory, we just put $\underline{0}^\nu = \underline{0}_\nu$.

Successor: Since the successor function S is definable in type theory, we just put $\mathsf{S}^\nu = \mathsf{S}_\nu$.

Projections: The projections are the functions π_i^ν defined in the previous section.

Composition: Composition is the standard composition in type theory. If $f \colon \mathbb{N}^k \rightharpoonup \mathbb{N}$ and $g_i \colon \mathbb{N}^n \rightharpoonup \mathbb{N}$ for $1 \leq i \leq k$, then we define

$$(f \circ \langle g_1, \ldots, g_k \rangle)^\nu \colon (\mathbb{N}^\nu)^n \to \mathbb{N}^\nu$$
$$(f \circ \langle g_1, \ldots, g_k \rangle)^\nu\, \overline{x} = (f^\nu\, \langle (g_1^\nu\, \overline{x}), \ldots, (g_k^\nu\, \overline{x}) \rangle).$$

Primitive Recursion: Let $f \colon \mathbb{N}^n \rightharpoonup \mathbb{N}$ and $g \colon \mathbb{N}^{n+2} \rightharpoonup \mathbb{N}$, and let h be defined by primitive recursion from f and g:

$$h \colon \mathbb{N}^{n+1} \rightharpoonup \mathbb{N}$$
$$h\, \langle \overline{x}, 0 \rangle = (f\, \overline{x})$$
$$h\, \langle \overline{x}, (\mathsf{S}\, y) \rangle = (g\, \overline{x}\, y\, (h\, \overline{x}\, y)).$$

We first define a version h' of h by recursion (fixpoint) on the natural numbers:

$$h' \colon (\mathbb{N}^\nu)^n \to \mathbb{N} \to \mathbb{N}^\nu$$
$$h'\, \overline{x}\, 0 = (f^\nu\, \overline{x})$$
$$h'\, \overline{x}\, (\mathsf{S}\, m) = (g^\nu\, \overline{x}\, \ulcorner m \urcorner\, (h'\, \overline{x}\, m))$$

and then lift it to the partial elements:

$$h^\nu \colon (\mathbb{N}^\nu)^{n+1} \to \mathbb{N}^\nu$$
$$h^\nu\, \overline{x}\, \ulcorner m \urcorner = (h'\, \overline{x}\, m)$$
$$h^\nu\, \overline{x}\, \rhd y = \rhd (h^\nu\, \overline{x}\, y).$$

Minimization: Let $f\colon \mathbb{N}^{n+1} \rightharpoonup \mathbb{N}$ and let g be defined from f by minimization:

$$g\colon \mathbb{N}^n \rightharpoonup \mathbb{N}$$
$$g\ \overline{x} = \text{least } y \text{ such that } (f\ \overline{x}\ y) = 0.$$

To define the type-theoretic version of g we use an auxiliary function that has an extra accumulation parameter, defined by corecursion (cofixpoint) on the result:

$$g'\colon (\mathbb{N}^\nu)^n \to \mathbb{N} \to \mathbb{N}^\nu \to \mathbb{N}^\nu$$
$$g'\ \overline{x}\ i\ \ulcorner m \urcorner = \mathsf{Cases}\ m\ \mathsf{of}\ \left\{ \begin{array}{ll} 0 & \mapsto \ulcorner i \urcorner \\ (\mathsf{S}\ m') & \mapsto \rhd(g'\ \overline{x}\ (\mathsf{S}\ i)\ (f^\nu\ \langle \overline{x}, \ulcorner(\mathsf{S}\ i)\urcorner\rangle)) \end{array} \right.$$
$$g'\ \overline{x}\ i\ \rhd y = \rhd(g'\ i\ y)$$

and then

$$g^\nu\colon (\mathbb{N}^\nu)^n \to \mathbb{N}^\nu$$
$$g^\nu\ \overline{x} = (g'\ \overline{x}\ 0\ (f^\nu\ \langle \overline{x}, \ulcorner 0 \urcorner\rangle)).$$

It is routine work, although quite long, to verify that the translations have the correct computational behavior.

Theorem 4.2. *Let $f\colon \mathbb{N}^n \rightharpoonup \mathbb{N}$ be a partial recursive function. For every $\overline{x}\colon \mathbb{N}^n$ and $y\colon \mathbb{N}$,*

$$(f\ \overline{x}) = y \quad \Longleftrightarrow \quad (f^\nu\ \overline{\ulcorner x \urcorner}) \overset{\nu}{=} \ulcorner y \urcorner.$$

Proof. The proof is a lengthy routine use of the techniques illustrated in the previous section. For a formal proof we should first formalize part of recursion theory. First, we have to define an inductive type of codes of recursive functions generated by constructors corresponding to the base functions, composition, primitive recursion, and minimization. Then, we need to give an operational semantics that associates to every code a relation on natural numbers, the relation being the graph of the recursive function. Finally, we have to interpret the codes as functions on the type of partial elements, as shown above, and prove that this interpretation is sound with respect to the operational semantics.

There are no conceptual problems in doing this, but a lot of technical work. We rather refer to Section 6, where we prove that it is possible to construct fixed points of functional operators. Since it is a known fact that all recursive functions can be realized by such a fixpoint combinator, the present statement will automatically follow. □

Let me stress the advantage of this approach in comparison with other methods to formalize general recursion in type theory.

First of all, some of the techniques, for example that of Balaa and Bertot [3] and that of Barthe and others [7, 9], do not address the question of partiality but present ways of extending the definition schemes for total recursive functions. The method of Bove and Capretta [14, 15, 16, 17] allows the definition of partial functions by restricting them to their domain of convergence. However, it is still not possible to apply a function freely to an argument, but it is necessary first to prove that the argument satisfies the domain predicate.

With coinductively defined types of partial elements, functions can be freely applied to arguments without the need of extra logical information.

5. Nested recursion

Theorem 4.2 says that every computable function can be represented in type theory. In this section we look at some specific examples of nested recursive functions. In general

we represent a partial function from A to B as an element of $A \to B^\nu$, since the values on the diverging elements of A must, by strictness, diverge.

We start with the simplest example of nested recursion [15]:

$$\mathsf{nest} \colon \mathbb{N} \rightharpoonup \mathbb{N}$$
$$\mathsf{nest}\ 0 = 0$$
$$\mathsf{nest}\ (\mathsf{S}\ n) = (\mathsf{nest}\ (\mathsf{nest}\ n)).$$

It is clear that nest is constantly 0, so we can implement it in type theory as the constant 0. However, we are interested in the form in which the function is defined. More complex nested functions do not have a simple non-nested presentation. We see an example later. We represent nest in type theory by using the method of accumulation of results and tail recursion:

$$\mathsf{cnest} \colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}^\nu$$
$$\mathsf{cnest}\ \langle n, 0 \rangle = \ulcorner n \urcorner$$
$$\mathsf{cnest}\ \langle 0, (\mathsf{S}\ m) \rangle = \triangleright(\mathsf{cnest}\ \langle 0, m \rangle)$$
$$\mathsf{cnest}\ \langle (\mathsf{S}\ n), (\mathsf{S}\ m) \rangle = \triangleright(\mathsf{cnest}\ \langle n, m + 2 \rangle).$$

It is easy to check that $(\mathsf{cnest}\ \langle n, m \rangle)$ computes the value $(\mathsf{nest}^m\ n)$, so the extra parameter m keeps track of the number of nested iteration of nest. Now we can define in type theory

$$\mathsf{nest} \colon \mathbb{N} \to \mathbb{N}^\nu$$
$$\mathsf{nest}\ n = (\mathsf{cnest}\ \langle n, 1 \rangle).$$

Let us look at a more interesting example. We define a class of nested recursive functions, that we nickname *the devil's nest*. Let A be a type, T a decidable subset of A, representable in type theory by a predicate $P \colon A \to \mathsf{Prop}$ such that $\forall a : A.(P\ a) \vee \neg(P\ a)$. Let $i \colon A \to A$ and $g \colon T \to A$ (that is, $g \colon (x \colon A)(P\ x) \to A$). Then the function $\mathsf{dev}_{T,i,g}$ is defined as

$$\mathsf{dev}_{T,i,g} \colon A \rightharpoonup A$$
$$\mathsf{dev}_{T,i,g}\ a = \begin{cases} (g\ a) & \text{if } a \in T \\ (\mathsf{dev}_{T,i,g}\ (\mathsf{dev}_{T,i,g}\ (i\ a))) & \text{otherwise} \end{cases}$$

We formalize this function in type theory by using, as before, an extra parameter that keeps track of the number of nested calls to $\mathsf{dev}_{T,i,g}$:

$$\mathsf{dev}_{\mathsf{aux}} \colon A \times \mathbb{N} \to A^\nu$$
$$\mathsf{dev}_{\mathsf{aux}}\ \langle a, m \rangle$$
$$= \begin{cases} \left(\mathsf{Cases}\ m\ \mathsf{of} \begin{cases} 0 & \mapsto \ulcorner(g\ a)\urcorner \\ (\mathsf{S}\ m') & \mapsto \triangleright(\mathsf{dev}_{\mathsf{aux}}\ \langle (g\ a), m' \rangle) \end{cases} \right) & \text{if } (P\ a) \\ \triangleright(\mathsf{dev}_{\mathsf{aux}}\ \langle (i\ a), (\mathsf{S}\ m) \rangle) & \text{otherwise} \end{cases}$$

We have left the proof argument of g implicit to simplify notation: Rigorously, we should have written $(g\ a\ h)$ in place of $(g\ a)$, in the case where $(P\ a)$ is true, where h is the proof of $(P\ a)$ guaranteed by the branch case. Similarly to the previous case, the original function is recovered by specifying an initial value for the accumulation parameter, in this case 0 :

$$\mathsf{dev}_{T,i,g} \colon A \to A^\nu$$
$$\mathsf{dev}_{T,i,g}\ a = (\mathsf{dev}_{\mathsf{aux}}\ \langle a, 0 \rangle).$$

Another interesting example occurs when a recursive call is followed by a call to another function, as in the definition of primitive recursion but without a decreasing argument. Let

$h\colon A \to A$. We want to formalize the following function:

$$\mathsf{d}_h\ a = \begin{cases} (g\ a) & \text{if } x \in T \\ (h\ (\mathsf{d}_h\ (i\ a))) & \text{otherwise.} \end{cases}$$

We use a continuation-passing style translation, using an extra functional parameter:

$$\mathsf{d}_{\mathsf{aux}}\colon (A \to A^\nu) \times A \to A^\nu$$
$$\mathsf{d}_{\mathsf{aux}}\ \langle k, a \rangle = \begin{cases} k\ (g\ a) & \text{if } (P\ a) \\ \triangleright(\mathsf{d}_{\mathsf{aux}}\ \langle k \circ h, (i\ a) \rangle) & \text{otherwise} \end{cases}$$

and then

$$\mathsf{d}_h\colon A \to A^\nu$$
$$\mathsf{d}_h\ a = (\mathsf{d}_{\mathsf{aux}}\ \langle \lambda x.\ulcorner x \urcorner, a \rangle).$$

Finally, we put the two preceding examples together to obtain the most general version of the devil's nest:

$$\mathsf{devil}_{T,i,g,h}\colon A \rightharpoonup A$$
$$\mathsf{devil}_{T,i,g,h}\ a = \begin{cases} (g\ a) & \text{if } a \in T \\ (h\ (\mathsf{devil}_{T,i,g,h}\ (\mathsf{devil}_{T,i,g,h}\ (i\ a)))) & \text{otherwise} \end{cases}$$

This function is formalized by

$$\mathsf{devil}_{\mathsf{aux}}\colon (A \to A^\nu) \times \mathbb{N} \times A \to A^\nu$$
$$\mathsf{devil}_{\mathsf{aux}}\ \langle k, m, a \rangle$$
$$= \begin{cases} \left(\mathsf{Cases}\ m\ \mathsf{of}\ \begin{cases} 0 & \mapsto k\ (g\ a) \\ (\mathsf{S}\ m') & \mapsto \triangleright(\mathsf{devil}_{\mathsf{aux}}\ \langle k, m', (g\ a) \rangle) \end{cases} \right) & \text{if } (P\ a) \\ \triangleright(\mathsf{devil}_{\mathsf{aux}}\ \langle k \circ h, (\mathsf{S}\ m), (i\ a) \rangle) & \text{otherwise} \end{cases}$$

and then

$$\mathsf{devil}_{T,i,g,h}\colon A \to B^\nu$$
$$\mathsf{devil}_{T,i,g,h}\ a = (\mathsf{devil}_{\mathsf{aux}}\ \langle \lambda x.\ulcorner x \urcorner, 0, a \rangle).$$

There is another possible generalization: Instead of having just one nested recursive call, we could have a variable number of nestings. It is clear that then we just have to modify the definition by adding the nesting number to the accumulation parameter.

6. Fixed points of function operators

We want to construct functions defined as fixed points of function operators. Let $F\colon (A \to B^\nu) \to (A \to B^\nu)$ be such an operator. Our goal is to define a function $\mathsf{Y}(F)\colon A \to B^\nu$ such that $(F\ \mathsf{Y}(F))$ is extensionally equal to $\mathsf{Y}(F)$. Moreover $\mathsf{Y}(F)$ must be minimal with respect to convergence.

Some conditions must be imposed on F for the construction to be possible. Since F is supposed to represent a generic recursive scheme, one sensible condition is finitarity: We assume that, to compute a specific result, F uses its arguments only on a finite number of inputs.

Definition 6.1. We say that $F\colon (A \to B^\nu) \to (A \to B^\nu)$ is *finitary* if it satisfy the following condition: For every function $f\colon A \to B^\nu$ and every argument $a\colon A$ such that $(F\ f\ a)$ converges to some value, that is, $(F\ f\ a) \downarrow b$ for some $b\colon B$; there exists a finite number of arguments $a_1, \ldots, a_n\colon A$ such that f converges on each of them, $(f\ a_i) \downarrow b_i$, and

$$\forall g\colon A \to B^\nu.\ \bigwedge_{i=1}^{n} (g\ a_i) \downarrow b_i \to (F\ g\ a) \downarrow b$$

In words, F is finitary (or continuous) if its results depend only on the values of the function argument on a finite set of inputs.

Notice that all operators used in recursion theory, for example those needed for the constructions of Section 4, are finitary.

We will use three consequences of finitarity: first, a finitary operator preserves extensional equality; second, it preserves convergence order; third, its least prefixed point can be constructed in a countable number of steps.

In the rest of this section, let F be a finitary operator.

Definition 6.2. *Extensional equality* between two functions $f_1, f_2 \colon A \to B^\nu$ is defined as

$$f_1 \overset{\mathsf{ext}}{=} f_2 \iff \forall a \colon A.(f_1 \ a) \overset{\nu}{=} (f_2 \ a).$$

We say that F is *extensional* if

$$\forall f_1, f_2 \colon A \to B^\nu.f_1 \overset{\mathsf{ext}}{=} f_2 \to (F \ f_1) \overset{\mathsf{ext}}{=} (F \ f_2).$$

Lemma 6.3. *F is extensional.*

Proof. It follows trivially from the stronger Lemma 6.7. $\qquad\square$

A stronger property holds: F preserves the order on functions given by convergence.

Definition 6.4. The *convergence order* between partial elements is defined coinductively by

$$\mathsf{CoInductive}\ \mathsf{Le}_A^\nu\ [x, y \colon A^\nu] \colon \mathsf{Prop}\ :=$$
$$\left\{\begin{array}{l} \mathsf{le\_value} \colon (x, y \colon A^\nu; a \colon A)x \downarrow a \to y \downarrow a \to (\mathsf{Le}_A^\nu\ x\ y) \\ \mathsf{le\_steps} \colon (x, y \colon A^\nu)(\mathsf{Le}_A^\nu\ x\ y) \to (\mathsf{Le}_A^\nu\ \rhd x\ \rhd y) \\ \mathsf{le\_lstep} \colon (x, y \colon A^\nu)(\mathsf{Le}_A^\nu\ x\ y) \to (\mathsf{Le}_A^\nu\ \rhd x\ y) \end{array}\right.$$

We use the notation $x \sqsubseteq y$ for $\mathsf{Le}_A^\nu\ x\ y$.

Intuitively, $x \sqsubseteq y$ holds if x is obtained from y by adding some (potentially infinite) $\rhd$ steps.

The order between functions is defined pointwise on their values, that is, if $f_1, f_2 \colon A \to B^\nu$, then we define

$$f_1 \sqsubseteq f_2 \iff \forall a \colon A, (f_1 \ a) \sqsubseteq (f_2 \ a).$$

It is immediate that equality $\overset{\nu}{=}$ is a subrelation of the order $\sqsubseteq$, since the order is defined by three constructors of which the first two correspond to the constructors of equality. It is also easy to prove that $\sqsubseteq$ is a transitive relation and it is reflexive and antisymmetric with respect to $\overset{\nu}{=}$. The relation $\sqsubseteq$ is equivalent to implication between convergence statements.

Lemma 6.5. *The proposition $x \sqsubseteq y$ is equivalent to $\forall b \colon B.x \downarrow b \to y \downarrow b$.*

Proof. From left to right, by induction on the proof of $x \downarrow b$. From right to left, by coinduction and cases on x. $\qquad\square$

By antisymmetry we can conclude that the following characterization of equality holds.

Lemma 6.6. *The proposition $x \overset{\nu}{=} y$ is equivalent to $\forall b \colon B.x \downarrow b \leftrightarrow y \downarrow b$.*

Lemma 6.7. *The operator F preserves the convergence order, that is,*

$$\forall f_1, f_2 \colon A \to B^\nu.f_1 \sqsubseteq f_2 \to (F \ f_1) \sqsubseteq (F \ f_2).$$

Proof. It is a straightforward consequence of finitarity. $\qquad\square$

We want to define the least fixed point $\mathsf{Y}(F)\colon A \to B^\nu$. Intuitively, we run in parallel all the iterations of F, starting with the always undefined function, and we take as result the outcome of the first converging run. If we call $\perp$ the function $\lambda a.\triangleright^\infty$, then we have:

$$
\begin{array}{rll}
\mathsf{Y}(F)\ a = & (\perp\ a) & \text{if it converges; otherwise} \\
& ((F\ \perp)\ a) & \text{if it converges; otherwise} \\
& ((F^2\ \perp)\ a) & \text{if it converges; otherwise} \\
& \vdots &
\end{array}
$$

(This intuitive explanation is not precise: Since we cannot decide convergence, the actual function choses the first of those values that converges.) Formally, we start by defining a function that computes the first converging of two partial objects.

Definition 6.8. The function computing the first converging element of a pair is defined corecursively by

$$
\begin{array}{l}
\mathsf{fstconv}\colon B^\nu \to B^\nu \to B^\nu \\
\mathsf{fstconv}\ \ulcorner b \urcorner\ y = \ulcorner b \urcorner \\
\mathsf{fstconv}\ \triangleright x\ \ulcorner b \urcorner = \ulcorner b \urcorner \\
\mathsf{fstconv}\ \triangleright x\ \triangleright y = \triangleright(\mathsf{fstconv}\ x\ y).
\end{array}
$$

We use the notation $x \curlywedge y$ for $(\mathsf{fstconv}\ x\ y)$.

So $x \curlywedge y$ returns the first between x and y to converge. Note that this is the only function defined so far that does not preserve equality: The term $(\triangleright^n \ulcorner b_1 \urcorner) \curlywedge (\triangleright^m \ulcorner b_2 \urcorner)$ converges to b_1 if $n \leq m$, to b_2 otherwise, without b_1 and b_2 being necessarily equal. The result of the function depends sensibly on the number of $\triangleright$ steps in the arguments. However, our use of $\mathsf{fstconv}$ to obtain fixed points is such that the arguments are always *compatible*, that is, they never converge to different elements, although it may be possible that one of the two converges while the other diverges. In this case the result is equality preserving. (The notion of compatibility of partial elements is due to Tarmo Uustalu.)

Lemma 6.9. *For every* $y\colon B^\nu$, $(\triangleright^\infty \curlywedge y) \overset{\nu}{=} y$.

Proof. Proof by coinduction. Assume H is the proof of the statement:

$$
H\colon \forall y\colon B^\nu.(\triangleright^\infty \curlywedge y) \overset{\nu}{=} y.
$$

In the computation of $\triangleright^\infty \curlywedge y$ the first equation in the definition of $\mathsf{fstconv}$ is not used. The second or third equation is applied, according to the structure of y. If $y = \ulcorner b \urcorner$, then the second equation itself states the truth of the lemma, by reflexivity of $\overset{\nu}{=}$. If $y = \triangleright y'$ then, by the third equation, $(\triangleright^\infty \curlywedge y) = (\triangleright \triangleright^\infty \curlywedge \triangleright y') = \triangleright(\triangleright^\infty \curlywedge y')$. By the coinductive hypothesis H applied to y', $\triangleright^\infty \curlywedge y' \overset{\nu}{=} y'$. By $\mathsf{eq\_step}$ we then have that $\triangleright(\triangleright^\infty \curlywedge y') \overset{\nu}{=} \triangleright y'$, that is, $(\triangleright^\infty \curlywedge y) \overset{\nu}{=} y$, as desired. The recursive application of H is guarded by $\mathsf{eq\_step}$. $\square$

Obviously, $\mathsf{fstconv}$ converges only if one of its arguments converges.

Lemma 6.10. *For every* $x, y\colon B^\nu$ *and* $b\colon B$, $(x \curlywedge y) \downarrow b \to x \downarrow b \vee y \downarrow b$.

Proof. By induction on the proof of $(x \curlywedge y) \downarrow b$. $\square$

The vice-versa is also true, but we have to be careful to take into account the non-extensionality of fstconv: if, for example, x converges to b, it is not guaranteed that $x \curlywedge y$ also converges to b, because y may converge to a different b' in a shorter time. However, if x is lower that y in the convergence order, we know that they cannot converge to different values.

Lemma 6.11. *Let $x, y \colon B^\nu$ and $b \colon B$, assume that $x \sqsubseteq y$; then $x \downarrow b \to (x \curlywedge y) \downarrow b$.*

Proof. By induction of the proof of $x \downarrow b$. $\qquad\square$

We can recursively define an infinitary version of fstconv.

Definition 6.12. The operation of computing the first converging element of a sequence is defined corecursively by

$$\begin{aligned}
&\mathsf{parallel\_search\_aux} \colon (\mathbb{N} \to B^\nu) \to \mathbb{N} \to B^\nu \to B^\nu \\
&\mathsf{parallel\_search\_aux}\ f\ n\ \ulcorner b \urcorner = \ulcorner b \urcorner \\
&\mathsf{parallel\_search\_aux}\ f\ n\ \triangleright x = \triangleright\, \mathsf{parallel\_search\_aux}\ f\ (\mathsf{S}\ n)\ (x \curlywedge (f\ n))
\end{aligned}$$

$$\begin{aligned}
&\mathsf{parallel\_search} \colon (\mathbb{N} \to B^\nu) \to B^\nu \\
&\mathsf{parallel\_search}\ f = \mathsf{parallel\_search\_aux}\ f\ 0\ \triangleright^\infty.
\end{aligned}$$

We write $(x \overset{\infty}{\curlywedge}_n f)$ for $(\mathsf{parallel\_search\_aux}\ f\ n\ x)$ and $\overset{\infty}{\curlywedge} f$ for $(\mathsf{parallel\_search}\ f)$.

Infinitary versions of the lemmas that we proved for fstconv hold for parallel_search, with corresponding additional hypotheses to take non-extensionality into account.

Lemma 6.13. *For all $f \colon \mathbb{N} \to B^\nu$ and $b \colon B$; $(\overset{\infty}{\curlywedge} f) \downarrow b \to \exists n.(f\ n) \downarrow b$.*

Lemma 6.14. *Let $f \colon \mathbb{N} \to B^\nu$ be increasing, that is, $\forall n, m . n < m \to (f\ n) \sqsubseteq (f\ m)$; if $b \colon B$ and $n \colon \mathbb{N}$, then $(f\ n) \downarrow b \to (\overset{\infty}{\curlywedge} f) \downarrow b$.*

Lemma 6.15. *Let $f \colon \mathbb{N} \to B^\nu$ and $y \colon B^\nu$; if $\forall n \colon \mathbb{N} . (f\ n) \sqsubseteq y$, then $\overset{\infty}{\curlywedge} f \sqsubseteq y$.*

Let us call k_i the ith iteration of F on the function that always diverges:

$$\begin{aligned}
k_0 &\equiv \lambda x . \triangleright^\infty \\
k_{n+1} &\equiv (F\ k_n).
\end{aligned}$$

Convergence of the k_is is stable with respect to the index i.

Lemma 6.16. *For every $n, m \colon \mathbb{N}$ such that $n \le m$, $k_n \sqsubseteq k_m$; equivalently*

$$\forall a \colon A, b \colon B . (k_n\ a) \downarrow b \Rightarrow (k_m\ a) \downarrow b.$$

Proof. We prove by induction on n that $k_n \sqsubseteq k_{n+1}$, the statement follows by transitivity of $\sqsubseteq$. The base case is obvious since k_0 has the constant value $\triangleright^\infty$, the least element of the convergence order. The inductive step follows immediately from Lemma 6.7. $\qquad\square$

Finally, we can simply define the fixed point of F pointwise as the result of running all of the k_is in parallel.

Definition 6.17. $\mathsf{Y}(F) = \lambda a \colon A . \overset{\infty}{\curlywedge}(\lambda n.(k_n\ a))$.

If $\mathsf{Y}(F)$ converges on a certain element a, it must give the same result as one of the k_is.

Lemma 6.18. $(\mathsf{Y}(F)\ a) \downarrow b \Leftrightarrow \exists n.(k_n\ a) \downarrow b$.

Proof. From left to right it follows from Lemma 6.13. From right to left it follows from Lemma 6.14. $\qquad\square$

The combination of the previous lemmas provides a proof of the soundness of $\mathsf{Y}(F)$ as a least fixed point of F.

Theorem 6.19. $\mathsf{Y}(F)$ *is a fixed point of* F: $(F\ \mathsf{Y}(F)) \overset{\mathsf{ext}}{=} \mathsf{Y}(F)$.

Proof. We prove that $(F\ \mathsf{Y}(F)) \sqsubseteq \mathsf{Y}(F)$ and $\mathsf{Y}(F) \sqsubseteq (F\ \mathsf{Y}(F))$. The statement follows from antisymmetry of $\sqsubseteq$.

To prove that $(F\ \mathsf{Y}(F)) \sqsubseteq \mathsf{Y}(F)$ we just have to show, by Lemma 6.5, that for all $a\colon A$ and $b\colon B$, if $(F\ \mathsf{Y}(F)\ a) \downarrow b$ then also $(\mathsf{Y}(F)\ a) \downarrow b$. By finitarity of F, $(F\ \mathsf{Y}(F)\ a) \downarrow b$ implies that there exist $a_1, \ldots, a_k\colon A$ and $b_1, \ldots, b_k\colon B$ such that $(\mathsf{Y}(F)\ a_i) \downarrow b_i$ and the result of $(F\ \mathsf{Y}(F)\ a)$ depends only on these arguments. By Lemma 6.18, we have that $(k_{n_i}\ a_i) \downarrow b_i$ for some indexes n_i. Let n be the largest of the n_is. By Lemma 6.16, $(k_n\ a_i) \downarrow b_i$ for every i. Therefore, by finitarity of F, $(F\ k_n\ a) \downarrow b$. But $(F\ k_n) = k_{n+1}$, so $(k_{n+1}\ a) \downarrow b$. By Lemma 6.18, it follows that $(\mathsf{Y}(F)\ a) \downarrow b$, as desired.

In the other direction, we prove that $\mathsf{Y}(F) \sqsubseteq (F\ \mathsf{Y}(F))$. Let $a\colon A$, we show that $(k_n\ a) \sqsubseteq (F\ \mathsf{Y}(F)\ a)$ for every n; the statement follows by Lemma 6.15. For $n = 0$ it is trivial because $(k_0\ a) = \triangleright^\infty$. For non-zero values we have $(k_{n+1}\ a) = (F\ k_n\ a)$ and, by Lemma 6.7, we just need to prove $(k_n\ a) \sqsubseteq (\mathsf{Y}(F)\ a)$. But this follows easily from Lemmas 6.5 and 6.18. $\qquad\square$

Theorem 6.20. $\mathsf{Y}(F)$ *is the least prefixed point of* F: *if* f *is any prefixed point of* F, *that is*, $(F\ f) \sqsubseteq f$, *then* $\mathsf{Y}(F) \sqsubseteq f$.

Proof. We prove by induction on n that $k_n \sqsubseteq f$. For $n = 0$ it is obvious. Assume that $k_n \sqsubseteq f$. Then, by Lemma 6.7, $k_{n+1} = (F\ k_n) \sqsubseteq (F\ f) \sqsubseteq f$ and we are done. The statement then follows from Lemma 6.15. $\qquad\square$

A method to construct fixed points of function operators is described by Balaa and Bertot [3], but in their case it is necessary to prove that recursive calls are decreasing with respect to some wellfounded order. The advantage of our method is that it produces fixed points of every finitary operator, without requiring any additional logical information.

7. Lazy interpretation

The formalization of partial elements of A as terms of type A^ν gives a *strict* interpretation of functions. There cannot be a partial evaluation of a term: If $x\colon A^\nu$, then we can investigate its shape, that is either a result $\ulcorner a \urcorner$ or a step $\triangleright x'$. In the latter case, we can go on investigating the shape of x'. As long as we get step cases, we have no information about the result. When we get a result $\ulcorner a \urcorner$ we get all the information.

In functional programming, it is useful to be able to compute only partial information about a result. For example, we may need to know that a certain natural number result is a successor, without computing it completely. An instance in which this capacity can be

used in computation is the following example of two mutually defined recursive functions:

$$\mathsf{sloth}_1 \colon \mathbb{N} \rightharpoonup \mathbb{N}$$
$$\mathsf{sloth}_1\ 0 = 0$$
$$\mathsf{sloth}_1\ (\mathsf{S}\ n) = (\mathsf{sloth}_1\ (\mathsf{sloth}_2\ n)) + (\mathsf{sloth}_2\ n)$$

$$\mathsf{sloth}_2 \colon \mathbb{N} \rightharpoonup \mathbb{N}$$
$$\mathsf{sloth}_2\ 0 = 0$$
$$\mathsf{sloth}_2\ (\mathsf{S}\ m) = \begin{cases} (\mathsf{sloth}_2\ (\mathsf{sloth}_1\ m)) + m & \text{if } (\mathsf{sloth}_1\ m) \leq m \\ 0 & \text{otherwise} \end{cases}$$

It is easy to see that the computation of $(\mathsf{sloth}_1\ 13)$ diverges, independently of whether the evaluation is strict or lazy. However, if we evaluate it partially, we obtain

$$(\mathsf{sloth}_1\ 13) = (\mathsf{sloth}_1\ (\mathsf{sloth}_2\ 12)) + (\mathsf{sloth}_2\ 12) = (\mathsf{sloth}_1\ 19) + 19.$$

When computing $(\mathsf{sloth}_2\ 14)$, we need to decide whether $(\mathsf{sloth}_1\ 13) \leq 13$. With a strict evaluation strategy, the computation of $(\mathsf{sloth}_2\ 14)$ diverges because the evaluation of $(\mathsf{sloth}_1\ 13)$ diverges. However, with a lazy evaluation strategy, we can determine that $(\mathsf{sloth}_1\ 13) = (\mathsf{sloth}_1\ 19) + 19 > 13$ and therefore $(\mathsf{sloth}_2\ 14) = 0$. This shows that a lazy evaluation strategy may converge when a strict one diverges.

To formalize lazy evaluation we have to modify the formalization of partial elements for inductive types. For an inductive type T, we define the type of its partial elements as the coinductive type with the same constructors plus a constructor for a computation step that does not yield any information.

Definition 7.1. Let T be an inductive type, that is, T is defined in type theory by

$$\text{Inductive } \mathsf{T} : \text{Type} := \begin{cases} \mathsf{c}_1 \colon (\Theta_1)\mathsf{T} \\ \vdots \\ \mathsf{c}_n \colon (\Theta_n)\mathsf{T}. \end{cases}$$

The type of *lazy partial elements* of T is

$$\text{CoInductive } {}^\nu\mathsf{T} : \text{Type} := \begin{cases} \mathsf{c}_1 \colon (\Theta_1[\mathsf{T} := {}^\nu\mathsf{T}])^\nu\mathsf{T} \\ \vdots \\ \mathsf{c}_n \colon (\Theta_n[\mathsf{T} := {}^\nu\mathsf{T}])^\nu\mathsf{T} \\ \mathsf{step} \colon {}^\nu\mathsf{T} \to {}^\nu\mathsf{T}. \end{cases}$$

As before, we use the notation $\triangleright x$ for $(\mathsf{step}\ x)$. We make a slight abuse of notation by using the same constructor names for T and ${}^\nu\mathsf{T}$.

This was my original formalization of partial elements (see Chapter 7 of [19]). The version A^ν is a simplification suggested to me by Herman Geuvers and Peter Aczel.

As an example, let us see how this variant produces the right computation behavior for the sloth_2 function. First of all, the type of lazy natural numbers is

$$\text{CoInductive } {}^\nu\mathbb{N} : \text{Type} := \begin{cases} 0 \colon {}^\nu\mathbb{N} \\ \mathsf{S} \colon {}^\nu\mathbb{N} \to {}^\nu\mathbb{N} \\ \triangleright \colon {}^\nu\mathbb{N} \to {}^\nu\mathbb{N}. \end{cases}$$

Here are the lazy versions of addition and order, addition is defined by corecursion on its result and order is defined as an inductive relation:

$$^\nu\!\!+ : {}^\nu\mathbb{N} \to {}^\nu\mathbb{N} \to {}^\nu\mathbb{N} \qquad\qquad \mathsf{Inductive}\ {}^\nu\!\!\leq\ : {}^\nu\mathbb{N} \to {}^\nu\mathbb{N} \to \mathsf{Prop}\ :=$$

$$x\ {}^\nu\!\!+ 0 = x$$
$$x\ {}^\nu\!\!+ (\mathsf{S}\ y) = \mathsf{S}\ (x\ {}^\nu\!\!+ y)$$
$$x\ {}^\nu\!\!+ \rhd y = \rhd(x\ {}^\nu\!\!+ y)$$

$$\begin{cases} \mathsf{le}_0\colon (y\colon {}^\nu\mathbb{N})0\ {}^\nu\!\!\leq y \\ \mathsf{le}_\mathsf{S}\colon (x, y\colon {}^\nu\mathbb{N})x\ {}^\nu\!\!\leq y \to (\mathsf{S}\ x)\ {}^\nu\!\!\leq (\mathsf{S}\ y) \\ \mathsf{le}_{\mathsf{step,l}}\colon (x, y\colon {}^\nu\mathbb{N})x\ {}^\nu\!\!\leq y \to (\rhd x)\ {}^\nu\!\!\leq y \\ \mathsf{le}_{\mathsf{step,r}}\colon (x, y\colon {}^\nu\mathbb{N})x\ {}^\nu\!\!\leq y \to x\ {}^\nu\!\!\leq (\rhd y) \end{cases}$$

This definition of order is not reflexive, because it is not possible to prove $\rhd^\infty\ {}^\nu\!\!\leq \rhd^\infty$, but we adopt it for our example for simplicity. The reader should find it easy to modify it into a reflexive relation.

Returning to our example, the conditioned equation in the definition of sloth_2 produces the the statement $(\mathsf{sloth}_1\ 19)\ {}^\nu\!\!+ 19\ {}^\nu\!\!\leq 13$. Contrary to the strict case, we can now evaluate this statement to a truth value. By the definition of ${}^\nu\!\!+$ this becomes

$$\mathsf{S}^{19}(\mathsf{sloth}_1\ 19)\ {}^\nu\!\!\leq \mathsf{S}^{13}0.$$

By inversion, if this statement were provable, the only applicable constructor would be $\leq_\mathsf{S}$, so $\mathsf{S}^{18}(\mathsf{sloth}_1\ 19)\ {}^\nu\!\!\leq \mathsf{S}^{12}0$ should also be provable. Repeating this step 13 times, we get $\mathsf{S}^6(\mathsf{sloth}_1\ 19)\ {}^\nu\!\!\leq 0$. But this statement does not match the conclusion of any constructor of ${}^\nu\!\!\leq$, thus it is not provable. We conclude that our original statement was not provable.

8. PARTIALITY AS A MONAD

The categorical notion of (strong) monad is a useful abstract description of computation. Eugenio Moggi studied this relation in a series of works [45, 46, 11]. We will show that the operator mapping a type A to the type of partial elements A^ν gives a computational monad in Moggi's sense. We recall the notion of monad in extension form, or Kleisli triple. The definition is taken from [11], Definition 3, pg. 45.

Definition 8.1 (Kleisli triple/monad in extension form). A *Kleisli triple* over a category $\mathcal{C}$ is a triple $(T, \eta, \_^*)$, where $T\colon |\mathcal{C}| \to |\mathcal{C}|$, $\eta_A\colon A \to TA$ for $A \in |\mathcal{C}|$, $f^*\colon TA \to TB$ for $f\colon A \to TB$ and the following equations hold:

- $\eta_A^* = \mathsf{id}_{TA}$;
- $f^* \circ \eta_A = f$ for $f\colon A \to TB$;
- $g^* \circ f^* = (g^* \circ f)^*$ for $f\colon A \to TB$ and $g\colon B \to TC$.

The intuitive understanding of a Kleisli triple is that, for a type of values A, TA is the type of computations of elements of type A. The unit η_A maps a value a to the trivial computation that just returns a. A function $f\colon A \to TB$ that maps values of type A to computations of type B can be extended to computations: $f^*\colon TA \to TB$ is the program that, given a computation x of type A, first computes it and, if it gives a value a, applies f to it.

In the present case, $\mathcal{C}$ is the category of setoids in type theory [8]. The operator T maps a type A to the type of partial elements A^ν. To be precise, we must define this operation on setoids. This is not difficult, it is just necessary to identify elements of A^ν that give equal results according to the book equality of the setoid. The details of the definition are left to the reader. Also for the other components of the monad, we define them on types and

leave to the reader the routine extension to setoids. The unit of the monad η is simply the return constructor of A^ν. If $f \colon A \to B^\nu$, then $f^* \colon A^\nu \to B^\nu$ is defined by

$$(f^* \ x) = \mathsf{Cases} \ x \ \mathsf{of} \ \left\{ \begin{array}{l} \ulcorner a \urcorner \mapsto \ulcorner (f \ a) \urcorner \\ \rhd \, x' \mapsto \rhd (f^* \ x'). \end{array} \right.$$

The equations for a Kleisli triple are satisfied. The proof of this fact is routine.

Theorem 8.2. *The triple* $(\_^\nu, \mathsf{return}, \_^*)$ *is a Kleisli triple on the category of setoids.*

A more powerful notion is that of *strong monad*. This is a monad in which a pair value-computation can be turned into the computation of a pair. The definition is taken from [45].

Definition 8.3. A *strong monad* over a category $\mathcal{C}$ with finite products is a monad (T, η, μ) together with a natural transformation $t_{A,B} \colon A \times TB \to T(A \times B)$ satisfying the following equations:

- $T(r_A) \circ t_{1,A} = r_{TA}$;
- $T(\alpha_{A,B,C}) \circ t_{A \times B, C} = t_{A, B \times C} \circ (\mathsf{id}_A \times t_{B,C}) \circ \alpha_{A,B,TC}$;
- $t_{A,B} \circ (\mathsf{id}_A \times \eta_B) = \eta_{A \times B}$;
- $t_{A,B} \circ (\mathsf{id}_A \times \mu_B) = \mu_{A \times B} \circ T(t_{A,B}) \circ t_{A,TB}$.

where r and α are the natural isomorphisms

- $r_A \colon 1 \times A \to A$;
- $\alpha_{A,B,C} \colon (A \times B) \times C \to A \times (B \times C)$.

Moggi [45] shows that the pure λ-calculus can be interpreted inside strong monads. In the present case the natural transformation t can be defined by

$$\begin{array}{l} t_{A,B} \colon A \times B^\nu \to (A \times B)^\nu \\ t_{A,B} \ \langle a, \ulcorner b \urcorner \rangle := \ulcorner \langle a, b \rangle \urcorner \\ t_{A,B} \ \langle a, \rhd x \rangle := \rhd \, t_{A,B} \langle a, x \rangle. \end{array}$$

Theorem 8.4. *The quadruple* $(\_^\nu, \mathsf{return}, \_^*, t)$ *is a strong monad.*

A thorough study of this monad is the subject of a forthcoming article in collaboration with Thorsten Altenkirch and Tarmo Uustalu.

ACKNOWLEDGMENT

REFERENCES

[1] Peter Aczel. *Non-Well-Founded Sets*. Number 14 in CSLI Lecture Notes. Stanford University, 1988.
[2] Peter Aczel. Algebras and coalgebras. In Roland Backhouse, Roy Crole, and Jeremy Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *LNCS*, pages 79–88. Springer, 2002.

[3] Antonia Balaa and Yves Bertot. Fix-point equations for well-founded recursion in type theory. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 2000.

[4] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 2*, pages 117–309. Oxford University Press, 1992.

[5] Henk Barendregt and Herman Geuvers. Proof-assistants using dependent type systems. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 1151–1238. Elsevier Science Publishers, 2001.

[6] G. Barthe and T. Coquand. An introduction to dependent type theory. In G. Barthe, P. Dybjer, L. Pinto, and J. Saraiva, editors, *Proceedings of APPSEM'00*, volume 2395 of *LNCS*. Springer, 2002. To appear.

[7] G. Barthe, M. J. Frade, E. Gimnez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. *Mathematical Structures in Computer Science*, 14:97–141, 2002.

[8] Gilles Barthe, Venanzio Capretta, and Olivier Pons. Setoids in type theory. *Journal of Functional Programming*, 13(2):261–293, 2003.

[9] Gilles Barthe, Benjamin Grégoire, and Fernando Pastawski. Practical inference for type-based termination in a polymorphic setting. In Urzyczyn [58], pages 71–85.

[10] Jon Barwise and Lawrence Moss. *Vicious Circles*. Number 60 in CSLI Lecture Notes. CSLI, Stanford, California, 1996.

[11] Nick Benton, John Hughes, and Eugenio Moggi. Monads and effects. In Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saravia, editors, *Applied Semantics*, volume 2395 of *LNCS*, pages 42–122. Springer, 2002. International Summer School, APPSEM 2000, Caminha, Portugal, September 9–15, 2000, Advanced Lectures.

[12] Yves Bertot. Fileters on coinductive streams, an application to eratosthenes' sieve. In Urzyczyn [58], pages 102–115.

[13] Yves Bertot, Venanzio Capretta, and Kuntal Das Barman. Type-theoretic functional sematics. In V. A. Carreno, C. A. Munoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics*, volume 2410 of *LNCS*, pages 83–97. Springer, 2002. 15th International Conference, TPHOLs 2002, Hampton, VA, USA, August 20-23, 2002.

[14] A. Bove. Simple general recursion in type theory. *Nordic Journal of Computing*, 8(1):22–42, Spring 2001.

[15] Ana Bove and Venanzio Capretta. Nested general recursion and partiality in type theory. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics: 14th International Conference, TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*, pages 121–135. Springer-Verlag, 2001.

[16] Ana Bove and Venanzio Capretta. Modelling general recursion in type theory. To appear in *Mathematical Structures in Computer Science*. Available at `http://www.science.uottawa.ca/~vcapr396/`, 2004.

[17] Ana Bove and Venanzio Capretta. Recursive functions with higher order domains. In Urzyczyn [58], pages 116–130.

[18] Venanzio Capretta. Universal algebra in type theory. In Yves Bertot, Gilles Dowek, André Hirschowits, Christine Paulin, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs '99*, volume 1690 of *Lecture Notes in Computer Science*, pages 131–148. Springer-Verlag, 1999.

[19] Venanzio Capretta. *Abstraction and Computation*. PhD thesis, Computing Science Institute, University of Nijmegen, 2002.

[20] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.

[21] Robert L. Constable. Partial functions in constructive formal theories. In *Theoretical Computer Science, 6th GI-Conference, Dortmunt*, volume 145 of *LNCS*, pages 1–18, January 1983.

[22] Robert L. Constable and Scott Fraser Smith. Partial object in constructive type theory. In *Logic in Computer Science, Ithaca, New York*, pages 183–193, Washington, D.C., June 1987. IEEE.

[23] Thierry Coquand. Infinite objects in type theory. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs. International Workshop TYPES'93*, volume 806 of *Lecture Notes in Computer Science*, pages 62–78. Springer-Verlag, 1993.

[24] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76:95–120, 1988.

[25] Thierry Coquand and Christine Paulin. Inductively defined types. In P. Martin-Löf, editor, *Proceedings of Colog '88*, volume 417 of *LNCS*. Springer-Verlag, 1990.

[26] Catherine Dubois and Véronique Viguié Donzeau-Gouge. A setp towards the mechanization of partial functions: domains as inductive predicates. Presented at CADE-15, Workshop on Mechanization of Partial Functions, 1998.

[27] Herman Geuvers. Inductive and coinductive types with iteration and recursion. In Bengt Nord-ström, Kent Pettersson, and Gordon Plotkin, editors, *Proccedings of the 1992 Workshop on Types for Proofs and Programs, Båstad, Sweden, June 1992*, pages 193–217, 1992. `ftp://ftp.cs.chalmers.se/pub/cs-reports/baastad.92/proc.dvi.Z`.

[28] Herman Geuvers. *Logics and Type systems*. PhD thesis, University of Nijmegen, September 1993.

[29] Herman Geuvers, Erik Poll, and Jan Zwanenburg. Safe proof checking in type theory with Y. In F. Flum and M. Rodriguez-Artalejo, editors, *Computer Science Logic (CSL'99)*, volume 1683 of *LNCS*, pages 439–452. Stringer-Verlag, 1999.

[30] Eduardo Giménez. Codifying guarded definitions with recursive schemes. In Peter Dybjer, Bengt Nord-ström, and Jan Smith, editors, *Types for Proofs and Programs. International Workshop TYPES '94*, volume 996 of *Lecture Notes in Computer Science*, pages 39–59. Springer, 1994.

[31] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972. Thèse d'Etat.

[32] Jean-Yves Girard. *Proofs and Types*. Cambridge University Press, 1989. Translated and with appendices by Paul Taylor and Yves Lafont.

[33] Tatsuya Hagino. *A Categorical Programming Language*. PhD thesis, University of Edinburgh, 1987.

[34] Tatsuya Hagino. A typed lambda calculus with categorical type constructors. In D. H. Pitt, A. Poigné, and D. E. Rydeheard, editors, *Category Theory and Computer Science*, volume 283 of *LNCS*, pages 140–157. Springer, 1987.

[35] Michael Hedberg. A type-theoretic interpretation of constructive domain theory. *Journal of Automated Reasoning*, 16:369–425, 1996.

[36] Claudio Hermida and Bart Jacobs. Structural induction and coinduction in a fibrational setting. *Information and Computation*, 145:107–152, 1998.

[37] W. A. Howard. The formulae-as-types notion of construction. In J. P. Selding and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[38] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.

[39] Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science, VI, 1979*, pages 153–175. North-Holland, 1982.

[40] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984. Notes by Giovanni Sambin of a series of lectures given in Padua, June 1980.

[41] Conor McBride and James McKinna. The view from the left. draft paper, 2002.

[42] N. P. Mendler, P. Panangaden, and R. L. Constable. Infinite objects in type theory. In *Proceedings, Symposium on Logic in Computer Science*, pages 249–255, Cambridge, Massachussetts, 16–18 June 1986. IEEE Computer Society.

[43] Paul Francis Mendler. *Inductive Definition in Type Theory*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, 1987.

[44] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[45] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings, Fourth Annual Symposium on Logic in Computer Science (LICS)*, pages 14–23. IEEE Computer Society Press, 1989.

[46] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[47] Bengt Nordström. Terminating general recursion. *BIT*, 28(3):605–619, October 1988.

[48] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory. An Introduction*, volume 7 of *International Series of Monographs on Computer Scence*. Oxford University Press, 1990.

[49] C. Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In M. Bezem and J.-F. Groote, editors, *Proceedings of the conference Typed Lambda Calculi and Applications*, volume 664 of *LNCS*, 1993. LIP research report 92-49.

[50] Lawrence C. Paulson. Constructing recursion operators in intuitionistic type theory. *Journal of Symbolic Computation*, 2:325–355, 1986.

[51] F. Pfenning and C. Paulin-Mohring. Inductively defined types in the Calculus of Constructions. In *Proceedings of Mathematical Foundations of Programming Semantics*, volume 442 of *LNCS*. Springer-Verlag, 1990. technical report CMU-CS-89-209.

[52] J. C. C. Phillips. Recursion theory. In *Handbook of Logic in Computer Science. Volume 1. Background: Mathematical Structures*, pages 79–187. Oxford University Press, 1992.

[53] John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3–4):233–248, 1993.

[54] Dana Scott. Domains for denotational semantics. In M. Nielsen and E. M. Schmidt, editors, *Automata, Languages and Programming. Ninth Colloquium*, volume 140 of *Lecture Notes in Computer Science*, pages 577–613. Springer-Verlag, 1982.

[55] Morten Heine B. Sørensen and P. Urzyczyn. Lectures on the Curry-Howard isomorphism. Available as DIKU Rapport 98/14, 1998.

[56] The Coq Development Team. LogiCal Project. *The Coq Proof Assistant. Reference Manual. Version 7.3 1*. INRIA, 2002.

[57] The Coq Development Team. LogiCal Project. *The Coq Proof Assistant. Reference Manual. Version 8*. INRIA, 2004. Available at the web page `http://pauillac.inria.fr/coq/coq-eng.html`.

[58] Paweł Urzyczyn, editor. *Typed Lambda Calculi and Applications. 7th International Conference, TLCA 2005, Nara, Japan, April 2005*, volume 3461 of *LNCS*. Springer, 2005.

[59] Benjamin Werner. *Méta-théorie du Calcul des Constructions Inductives*. PhD thesis, Université Paris 7, 1994.

[60] Glynn Winskel. *The Formal Semantics of Programming Languages*. The MIT Press, 1993.

[61] G. C. Wraith. A note on categorical datatypes. In D. H. Pitt, D. E. Rydehead, P. Dybjer, A. M. Pitts, and A. Poigné, editors, *Category Theory and Computer Science*, volume 389 of *Lecture Notes in Computer Science*, pages 118–127. Springer-Verlag, 1989.