Some Remarks on Type Systems for Course-of-value Recursion

Favio Ezequiel Miranda-Perea[1,2]

Departamento de Matemáticas
Facultad de Ciencias UNAM
Circuito Exterior s/n, Cd. Universitaria,
México D.F. 04510, México

Abstract

Course-of-value recursion is a scheme which allows us to define the value of a function in some argument of an inductive structure by using not only the immediate, but arbitrary previously computed values. In the categorical approach to typed (total) functional programming, where datatypes (codatatypes) are initial algebras (final coalgebras), one models this principle by a construction called histomorphism. On the other hand, it is known that other categorical principles such as catamorphisms and anamorphisms representing definitions by iteration and coiteration have been successfully used to implement safe type systems extending the second-order polymorphic lambda calculus, system F. Hence it is natural to pursuit the definition and implementation of fold operators corresponding to course-of-value recursion as well. This paper proposes some new such extensions and states some important remarks emerged while verifying the correctness and safety properties of their operational semantics, relying not only on the categorical, but also on the logical approach based on fixed-point operators. Our observations should be considered as a starting point for a deeper study of the interrelation between these two approaches.

Keywords: schemes of (co)recursion, course-of-value, typed lambda calculi, category theory, histomorphism, (co)inductive types, least fixed points

1 Introduction

In the calculational approach to programming, an implementation of a problem is derived from a particular specification by means of algebraic manipulation of formulas or equations. This approach has been quite succesful in functional programing, since expressions in such a language behave as mathematical functions due to the property of referential transparency. However, in order to enhace the efectiveness of calculations, the use of unstructured general recursion should be abandoned in favor of a set of structured (co)recursion schemes encoding typical recursion patterns, just as the use of arbitrary goto statements was replaced by the use of primitive control structures in imperative programming.

[1] This research has been partially supported by Conacyt-UNAM Mexico postdoctoral grant number 050289

[2] Email: favio@matematicas.unam.mx

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

A usual basis for the calculation approach is category theory, where recursion patterns are modelled by categorical constructions such as catamorphisms and anamorphisms, implemented in functional programming by the higher-order functions `fold` and `unfold`,respectively.

This paper concentrates in another useful pattern known as course-of-value recursion, captured by a construction called histomorphism in [21]. However, we do not focus in the categorical concepts, but rather in their implementation as type systems, an issue which has been succesfully achieved for other schemes (see [13,17]). Our overall goal is to implement recursion principles in the setting of typed lambda calculus, taking the categorical approach discussed above, as well as a fixed-point approach, usual in logics with inductive definitions, as a foundation. Although the formalisms presented here are not new strictly speaking (see [21,22]), our contribution, apart from present them in the framework of Curry-style type systems and generalizing them with full monotonicity instead of positivity, is to state some important remarks about their static and dynamic semantics. Moreover this work intends to be a starting point for a deeper comparison of two distinct approaches to model course-of-value recursion.

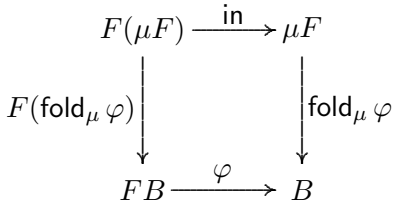
The paper is organised as follows: after this introduction we recall the categorical approach to (co)iteration in section 2. We settle a formal definition of course-of-value recursion and introduce the categorical construction modelling it in section 3. The migration from categories to types is explained in section 4 where we also present the formalization of basic fold and unfold operators. In section 5 we discuss a safe type system for course-of-value iteration and two different operational semantics corresponding to different implementations of the course-of-value fold operator, this is made clear in the case of natural numbers. After a brief discussion on termination (strong normalisation) and its high price, we adopt the point of view of [20], and still pursuit this property by abandoning the realm of categories in favor of the fixed-point approach in section 6, where we present an alternative system for course-of-value iteration which happens to be terminating. Here we realize that for natural numbers there are still some operational problems and propose full course-of-value primitive recursion as definitive solution in section 7. Finally in section 8 we provide some closing remarks and future work.

Through the paper we will be using Haskell notation, in particular $f \,.\, g$ denotes function composition. Nevertheless the translation to other functional language is straightforward.

2 The Categorical Approach

In the categorical approach to typed (total) functional programming, datatypes, like natural numbers, lists or trees, are modelled by initial algebras, whereas codatatypes, like streams, colists or infinite trees are modelled by final coalgebras. For this purpose a default category $\mathcal{C}$ is used, which usually has finite products $(\times, 1)$ and coproducts $(+, 0)$ such that products distribute over coproducts, for example the category Set. The basic function definition schemes are iteration and coiteration modelled by constructions named catamorphisms and anamorphisms. Let us briefly recall the basic concepts.

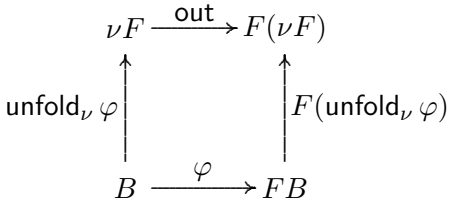
Definition 2.1 Let $F : \mathcal{C} \to \mathcal{C}$ be a functor. If it exists, the initial algebra of F is a pair $\langle \mu F, \mathsf{in} \rangle$ such that for every object B and arrow $\varphi : F\,B \to B$ there exists a unique arrow $\mathsf{fold}_\mu\,\varphi$ such that the following diagram commutes:



That is $(\mathsf{fold}_\mu\,\varphi)\,.\,\mathsf{in} = \varphi\,.\,F(\mathsf{fold}_\mu\,\varphi)$. In such case the arrow $\mathsf{fold}_\mu\,\varphi$, is called a catamorphism and the equation is called the principle of iteration on φ.

To get codatatypes we just dualize the above definition.

Definition 2.2 Let $F : \mathcal{C} \to \mathcal{C}$ be a functor. If it exists, the final coalgebra of F is a pair $\langle \nu F, \mathsf{out} \rangle$ such that for every object B and arrow $\varphi : B \to F\,B$ there exists a unique arrow $\mathsf{unfold}_\nu\,\varphi$ such that the following diagram commutes:



That is $\mathsf{out}\;.\;(\mathsf{unfold}_\nu\,\varphi) = F(\mathsf{unfold}_\nu\,\varphi)\,.\,\varphi$. In such case the arrow $\mathsf{unfold}_\nu\,\varphi$,is called an anamorphism and the equation is called the principle of coiteration on φ.

The existence of initial algebras and final coalgebras is guaranteed for a wide class of functors including all the functors built up from the identity and constant functors using products and coproducts. Let us recall some typical examples.

Example 2.3 The natural numbers are defined as $\mathsf{Nat} = \mu F$ where $FX = 1 + X$. In this case $\mathsf{in} : 1 + \mathsf{Nat} \to \mathsf{Nat}$ encodes the usual constructors $zero, suc$ by means of $zero = \mathsf{in}\,.\,\mathsf{inl}$ and $suc = \mathsf{in}\,.\,\mathsf{inr}$ where $\mathsf{inl}, \mathsf{inr}$ are the coproduct injections. As usual, in category theory, zero is a global element $zero : 1 \to \mathsf{Nat}$. Moreover, given functions $c : 1 \to C$ and $g : C \to C$ the catamorphism $f = \mathsf{fold}_\mu[c, g]$, where $[c, g] : 1 + B \to B$ is the usual copair of arrows, generates the following version of the principle of iteration: $f.zero = a,\ f.suc = g.f$. This corresponds to the following fold operator on natural numbers in Haskell

```
foldnat a g 0 = a
foldnat a g (n+1) = g (foldnat a g n)
```

Example 2.4 The codatatype of streams over a given type A is usually represented as $\mathsf{stream}\,A = \nu F$, where $F\,X = A \times X$. The $\mathsf{out} : \mathsf{stream}\,A \to A \times \mathsf{stream}\,A$ arrow encodes the usual destructors $head, tail$ defined by $head = \mathsf{fst}\,.\,\mathsf{out},\ tail = \mathsf{snd}\,.\,\mathsf{out}$. Given two functions $h : B \to A,\ t : B \to B$ the anamorphism $f = \mathsf{unfold}_\nu\langle h, t\rangle$, where $\langle h, t\rangle : B \to A \times B$ is the usual pair of arrows, generates the following version

of the coiteration principle: $head.f = h,\ tail.f = f.t$. This corresponds to the following unfold operator on streams in Haskell:

```
head (unfold h t x) = h x
tail (unfold h t x) = unfold h t (t x)
```

Example 2.5 The definition $\mathsf{Colist}\, C = \nu F$, where $F\, X = C \times (1 + X)$, generates the codatatype of non-empty and maybe infinite lists of elements of C. In this case we have $tail : \mathsf{Colist}\, C \to 1 + \mathsf{Colist}\, C, tail = \mathsf{snd} \,.\, \mathsf{out}$. Therefore this destructor can return an error (an inhabitant of 1) indicating that the tail does not exist, in this case the colist is finite. We will use colists in several ocassions later in this paper.

The following well-known result, due to Lambek, states that the arrows in and out are isomorphisms

Proposition 2.6 *The arrows* $\mathsf{in}, \mathsf{out}$ *are isomorphisms. Therefore there are inverse arrows* $\mathsf{in}^{-1} : \mu F \to F(\mu F)$ *and* $\mathsf{out}^{-1} : F(\nu F) \to \nu F$ *such that* $\mathsf{in}^{-1} \,.\, \mathsf{in} = \mathsf{Id}$ *and* $\mathsf{out} \,.\, \mathsf{out}^{-1} = \mathsf{Id}$.

The basic function definition schemes $\mathsf{fold}_\mu, \mathsf{unfold}_\nu$ are useful and well-known. However, there are other enhaced principles which allows us to define a wider class of functions in a more direct or elegant way. In this paper we are concerned with one of such principles called course-of-value recursion.

3 Course-of-Value Recursion

The scheme of course-of-value recursion (iteration) generalizes conventional iteration, given by catamorphisms, by allowing the result sought after to depend not only on the recursive result of applying the function to the immediate children of the current input, but also on the recursive result of any previous subterm. This idea is captured, for the case of natural numbers, by the following definition, which we consider to be simpler and more friendly than the usual mathematical definition.

Definition 3.1 Let Nat be the datatype of natural numbers, C any (co)datatype, $c_0, \ldots, c_k \in C$ and $g : \mathsf{Colist}\, C \to C$. A function $f : \mathsf{Nat} \to C$ is defined by course-of-value iteration with base cases $c_0, \ldots, c_k$ and step function g if and only if:

$$f(0) = c_0,\ f(1) = c_1,\ \ldots,\ f(k) = c_k$$
$$f(n+1) = g\big(\mathsf{rcd}\, f\, n\big),\ n \geq k$$

where $\mathsf{rcd} : (\mathsf{Nat} \to C) \to \mathsf{Nat} \to \mathsf{Colist}(C)$ is the function returning the history or record of f, namely $\mathsf{rcd}\ f\, n = [f\, n, f\, (n-1), \ldots, f\, 0]$. This function is coiteratively defined by:

$$\mathsf{head}\, (\mathsf{rcd}\, f\, n) = f\, n$$
$$\mathsf{tail}\, (\mathsf{rcd}\, f\, n) = \mathsf{if}\ n = 0\ \mathsf{then\ error\ else}\ \mathsf{rcd}\, f\, (n-1)$$

This definition generates, in particular, the following fold operator in Haskell, called a course-of-value fold `cvfold`:

```
cvfold g c 0 = c
```

```
cvfold g c (n+1) = g (rcd (cvfold g c) n)
                   where
                     rcd f n = (f n):
                                 (if n == 0 then error ‘‘No tail
                                   for rcd f 0’’ else (rcd f (n-1))
```

Observe that the step function g is required to receive the whole record of previous values. For efficiency instead, one could think of calculating only the values needed for the recursive call. However, a good implementation of course-of-value recursion does not calculate values repeatedly but relies on dynamic programming techniques like memoization which in our case can easily be achieved, see section 5.3. The reader can also realize that the record function rcd could also be defined by simple iteration, we will consider this option later. However we prefer the above definition, for one of the side goals of this paper is to show the nice interaction between recursion and corecursion, after all this latter principle comes almost free because of duality.

Example 3.2 Some functions definable with the above scheme are:

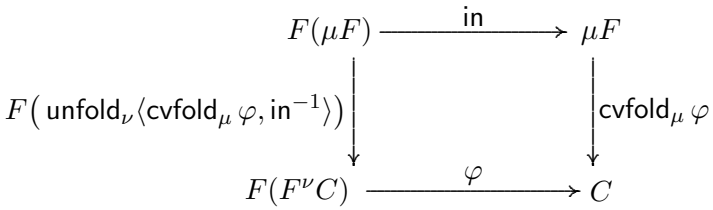
- The function fibo generates the elements of the sequence of Fibonacci numbers and is defined by taking $c_0 = c_1 = 1$ and $g\,\ell = sum\,(take\,2\,\ell)$ where sum adds the elements of a list of numbers and $take\,n\,\ell$ returns the list of first n elements of the list ℓ
- The function half returns the integer half of a natural number, specified by $\mathsf{half}\,0 = 0 = \mathsf{half}\,1$ and $\mathsf{half}\,(n+1) = \mathsf{half}(n-1) + 1$. It is defined by taking $c_0 = c_1 = 0$ and $g\,\ell = \ell\,!!\,1 + 1$ where $\ell\,!!\,n$ returns the nth element of ℓ counting from zero.
- A more interesting example is the function generating the Catalan numbers C_n given by $C_0 = 1$, $C_{n+1} = C_0C_n + C_1C_{n-1} \ldots + C_{n-1}C_1 + C_nC_0$. This shows the extreme case of course-of-value iteration where the whole record $[C_n, \ldots, C_0]$ is needed to define C_{n+1} . A generating function cat arises by taking $c_0 = 1$ and $g\,\ell = cnv\,\ell\,\ell$, where cnv is a function calculating the convolution of two lists, which in particular yields
  $cnv\,[a_0, \ldots, a_n]\,[b_0, \ldots, b_n] = a_0b_n + a_1b_{n-1} + \ldots + a_{n-1}b_1 + a_nb_0$.

The scheme of course-of-value iteration given in definition 3.1 is captured by the categorical combinator known as histomorphism (see [21]). We briefly recall it here.

Definition 3.3 [cv-algebra] Let $\langle \mu F, \mathsf{in}\rangle$ be the initial algebra of a functor $F : \mathcal{C} \to \mathcal{C}$. Given an object $C \in \mathcal{C}$ we define $F_C^{\times}\,X = C \times F\,X$. Assuming that for every C there is a final coalgebra $\langle \nu F_C^{\times}, \mathsf{out}\rangle$, we define a new functor F^{ν} such that $F^{\nu}\,C = \nu F_C^{\times}$. An F-cv-algebra is a pair $\langle C, \varphi\rangle$ where $\varphi : F(F^{\nu}C) \to C$.

Definition 3.4 [Histomorphism] Let $\langle \mu F, \mathsf{in}\rangle$ be the initial F-algebra. Given an F-cv-algebra $\langle C, \varphi\rangle$ the histomorphism $\mathsf{cvfold}_{\mu}\,\varphi : \mu F \to A$, is the unique arrow

making the following diagram commute:



where $\mathsf{unfold}_\nu\,\langle\mathsf{cvfold}_\mu\,\varphi,\mathsf{in}^{-1}\rangle$ denotes the anamorphism of the functor $F_C^\times$ with step arrow $\langle\mathsf{cvfold}_\mu\,\varphi,\mathsf{in}^{-1}\rangle$. The equation $(\mathsf{cvfold}_\mu\,\varphi).\ \mathsf{in} = \varphi\,.\,F(\mathsf{unfold}_\nu\langle\mathsf{cvfold}_\mu\,\varphi,\mathsf{in}^{-1}\rangle)$ is called the principle of course-of-value iteration.

Example 3.5 For the datatype of natural numbers Nat defined in example 2.3, cv-algebras are pairs $\langle C,\varphi\rangle$ where $\varphi : 1 + \mathsf{Colist}(C) \to C$. We will make clear later in section 5.1 that in this case, a histomorphism $f = \mathsf{cvfold}_\mu\ \varphi$ corresponds to a function $f : \mathsf{Nat} \to C$ defined by course-of-value iteration according to definition 3.1.

The existence and uniqueness of histomorphisms is ensured by the following property first obtained in [21].

Proposition 3.6 *Let $\langle\mu F,\mathsf{in}\rangle$ be the initial F-algebra, $h : \mu F \to C$ and $\varphi : F(F^\nu C) \to C$. Then the equation $h\,.\,\mathsf{in} = \varphi\,.\,F(\mathsf{unfold}_\nu\langle h,\mathsf{in}^{-1}\rangle)$ holds if and only if $h = \mathsf{fst}\ .\ \mathsf{out}\ .\ \mathsf{fold}_\mu(\mathsf{out}^{-1}\ .\langle\varphi,\mathsf{Id}\rangle)$. Therefore we can define a histomorphism as $\mathsf{cvfold}_\mu\,\varphi = \mathsf{fst}\ .\ \mathsf{out}\ .\ \mathsf{fold}_\mu(\mathsf{out}^{-1}\ .\langle\varphi,\mathsf{Id}\rangle)$.*

An interesting issue is to model course-of-value recursive types in Kleisli categories or to compare histomorphisms with Kleisli or Eilenberg-Moore constructions. However, it is not a goal of this paper to further develop the work on category theory (for this see [19,21]), but rather to implement recursion principles in the setting of typed lambda calculi, taking the categorical approach as a foundation. We discuss next the general idea to model later the principle of course-of-value iteration.

4 From Categories to Types

Our goal is to implement the course-of-value iteration scheme as defined by histomorphisms. This categorical approach is interesting and useful to obtain recursion combinators in typed lambda calculus. Recall that although the untyped lambda calculus has general recursion, the simple typed lambda calculus lacks recursion, for fixed-point operators like Y are not typable and, although simple iteration principles can be modelled in the second-order lambda calculus (system F), the encodings are impredicative and very hard to handle from a practical point of view.

To implement categorical combinators as constructors of typed lambda calculi, we need to see the type system as a category $\mathcal{T}$ where types are objects, arrows are transformations between types and arrow composition is the usual composition. Such categories of types and their features are well-known, see for example [4].
A functor $F : \mathcal{T} \to \mathcal{T}$ is then a transformation between types. In particular we use functors FX depending on a type variable X, which map a type B to a type

FB. Such functors are defined, more accurately, by expressions of the form λXF or $F = \lambda XG$ abstracting the type variable X. Note that the systems developed in this paper are not higher-order and therefore abstractions like λXF are only a useful notation. In particular an application $(\lambda XF)B$ of a functor to a type B should be understood as the capture-avoiding substitution $F[X := B]$.
Our aim is to define type systems with (co)inductive types: given a functor λXF we model its initial algebra with an inductive type $\mu(\lambda XF)$ or its final coalgebra with a coinductive type $\nu(\lambda XF)$. Observe that we cannot consider λXF to be immediately a functor for we only know its action on objects (types) but not on arrows (transformations between types). To allow the construction of the types $\mu(\lambda XF), \nu(\lambda XF)$ for a functor, type systems handling some kind of recursive or (co)inductive types, usually require the syntactical condition of X ocurring only on positive positions in F, that is, not to the left of an odd number of the $\to$ type constructor (see, for example [7]). Such condition guarantees the functoriality or monotonicity of λXF on function types. In our treatment we prefer to follow [13] and use full monotonicity instead: the functoriality of λXF on arrows is represented internally by means of a term $\mathsf{map} : (\lambda XF)\,\mathsf{mon}$ in a given context. The type $(\lambda XF)\,\mathsf{mon}$, defined as $\forall X \forall Y.(X \to Y) \to F \to FY$, represents the fact that the functor λXF is monotone (covariant) with respect to its argument X. Such terms are called *monotonicity witnesses*. For the merits of using full-monotonicity instead of positivity we point to [2]. In conclusion, a functor in our framework is a pair $\langle \lambda XF, \mathsf{map} \rangle$ where map is a term of type $(\lambda XF)\,\mathsf{mon}$ in the needed context. Thus, our functors are just like instances of the class `Functor` in Haskell. Next we define the basic (co)iteration schemes following the above ideas.

4.1 The Basic System

The definition of the type $(\lambda XF)\,\mathsf{mon}$ implies the use of polymorphic types. Therefore the systems in this paper are all extensions of the second-order polymorphic lambda calculus, system F or $\lambda 2$ (for a complete definition see appendix A). Our basic system will model initial algebras and catamorphisms as well as final coalgebras and anamorphisms as follows:

- Initial algebra: given a functor $F = \lambda XG$ we model its initial algebra $\langle \mu F, \mathsf{in} \rangle$ by means of a new type μF, called an inductive type, and a type constructor in defined by the following rule:

$$\frac{\Gamma \vdash t : F(\mu F)}{\Gamma \vdash \mathsf{in}\, t : \mu F}\ (\mu I)$$

- Catamorphisms: we introduce a new ternary term constructor fold_μ typed as follows:

$$\frac{\Gamma \vdash t : \mu F \qquad \Gamma \vdash m : \mathsf{mon}\, F \qquad \Gamma \vdash \varphi : F\, B \to B}{\Gamma \vdash \mathsf{fold}_\mu\, m\ \varphi\ \ t : B}\ (\mu E)$$

Finally the operational semantics is given by the following reduction rule, representing the principle of iteration:

$$\mathsf{fold}_\mu \ m\ \varphi\ (\mathsf{in}\, t) \to \varphi(m\ (\mathsf{fold}_\mu\ m\ \varphi)\, t)$$

where $\mathsf{fold}_\mu\, m\ \varphi$ means $\lambda x.\, \mathsf{fold}_\mu\, m\ \varphi\ x$. From now on we will be using similar conventions without further notice, for they simplify the reading of reduction rules. The origin of this reduction rule is clear, it corresponds to the principle of iteration of definition 2.1. It is also worth noting that the operational semantics models only a weak initial algebra, for we are not modelling uniqueness. However, this suffices our purposes.

- Final coalgebra: given a functor $F = \lambda X G$ we model its final coalgebra $\langle \nu F, \mathsf{out} \rangle$ by means of a new type νF, called a coinductive type, and a type constructor out with the typing rule:
$$\frac{\Gamma \vdash t : \nu F}{\Gamma \vdash \mathsf{out}\, t : F(\nu F)}\ (\nu E)$$
- Anamorphisms: we introduce a new ternary term constructor unfold_ν, typed as follows:
$$\frac{\Gamma \vdash t : B \qquad \Gamma \vdash m : F\,\mathsf{mon} \qquad \Gamma \vdash \varphi : B \to FB}{\Gamma \vdash \mathsf{unfold}_\nu\, m\ \varphi\ \ t : \nu F}\ (\nu I)$$
The operational semantics is given by the following reduction rule, representing the principle of coiteration:
$$\mathsf{out}(\mathsf{unfold}_\nu\ m\ \varphi\ t) \to m(\mathsf{unfold}_\nu\ m\ \varphi)(\varphi\, t)$$

Systems with iteration and coiteration constructors are well-known, in particular the one just described here is safe and terminating, for simple (co)iteration is definable as syntactic sugar in F, see for example [6,13,17].

4.2 On Inverses

Proposition 2.6 states that the arrows $\mathsf{in}, \mathsf{out}$ are isomorphisms. The inverses $\mathsf{in}^{-1}, \mathsf{out}^{-1}$ play usually the role of inductive destructors and coinductive constructors, respectively. In the case of Nat the inverse $\mathsf{in}^{-1} : \mathsf{Nat} \to 1 + \mathsf{Nat}$ represents the predecessor function whereas for $\mathsf{stream}\, A$ the inverse $\mathsf{out}^{-1} : A \times \mathsf{stream}\, A \to \mathsf{stream}\, A$ represents the stream constructor cons. These operators are modelled in a type system by the following rules:

$$\frac{\Gamma \vdash t : \mu F \qquad \Gamma \vdash m : F\,\mathsf{mon}}{\Gamma \vdash \mathsf{in}^{-1}\ m\ t : F(\mu F)} \qquad \frac{\Gamma \vdash t : F(\nu F) \qquad \Gamma \vdash m : F\,\mathsf{mon}}{\Gamma \vdash \mathsf{out}^{-1}\ m\ t : \nu F}$$

with operational semantics

$$\mathsf{in}^{-1}\ m\ (\mathsf{in}\, t) \to t \qquad \mathsf{out}(\mathsf{out}^{-1}\ m\ t) \to t$$

Observe that we are only modelling one half of the isomorphism, namely that $\mathsf{in}^{-1} \,.\, \mathsf{in} = \mathsf{Id}_{F(\mu F)}, \mathsf{out} \,.\, \mathsf{out}^{-1} = \mathsf{Id}_{\nu F}$.

5 A System with Course-of-Value Iteration

Next, we model course-of-value iteration obtaining an extension of system F called CCVT, a system with Categorical Course-of-Value (co)inductive Types. The new features are:

- Cv-algebras. Given a functor $F = \lambda X G$ the associated functors $F_C^{\times}$ and F^{ν} are defined as $F_C^{\times} = \lambda X.C \times F\,X$ and $F^{\nu} = \lambda Z.\nu(F_Z^{\times}) = \lambda Z.\nu(\lambda X.Z \times F\,X)$
- Course-of-value Iteration:

$$\frac{\Gamma \vdash t : \mu F \qquad \Gamma \vdash m : F\,\mathsf{mon} \qquad \Gamma \vdash \varphi : F(F^{\nu}\,C) \to C}{\Gamma \vdash \mathsf{cvfold}_{\mu}\, m\, \varphi\, t : C}\ (\mu E^{cv})$$

The dynamic semantics is given by the following reduction rule:

$$\mathsf{cvfold}_{\mu}\, m\, \varphi\, (\mathsf{in}\, t) \to \varphi\Big(m\, \big(\, \mathsf{unfold}_{\nu}\, m^{\times}\, \langle \mathsf{cvfold}_{\mu}\, m\, \varphi, \mathsf{in}^{-1} \rangle\big)\, t\Big)$$

where $m^{\times} = \lambda f \lambda p.(\mathsf{fst}\, p, m f(\mathsf{snd}\, p))$. Therefore, this reduction corresponds to the principle of course-of-vale iteration of definition 3.4

Let us see now the meaning of these definitions on natural numbers.

5.1 Course-of-Value on Usual Natural Numbers

The type of natural numbers is defined as $\mathsf{Nat} = \mu F$ with $F = \lambda X.1 + X$ and canonical monotonicity witness $\mathsf{mapN} = \lambda f.[\mathsf{inl}, \mathsf{inr}\,.f]$, where $[f_1\, f_2] = \lambda x.case(x, y.f_1 y, z.f_2 y)$ is the copair of functions. The constructors $0 : \mathsf{Nat}$, $\mathsf{suc} : \mathsf{Nat} \to \mathsf{Nat}$ are encoded in the in arrow by $0 = \mathsf{in}(\mathsf{inl}\,\star)$ and $\mathsf{suc}\, n = \mathsf{in}(\mathsf{inr}\, n)$. In this case a histomorphism is a function $f : \mathsf{Nat} \to C$ defined by $f = \mathsf{cvfold}_{\mu}\, \mathsf{mapN}\, \varphi$ with step function $\varphi : 1 + \mathsf{Colist}(C) \to C$, where $\mathsf{Colist}(C) = F^{\nu}(C) = \nu(\lambda X.C \times (1 + X))$ is the type of potentially infinite non-empty lists. This coinductive type has coded destructors head and tail, denoted here as $\mathsf{cur} : \mathsf{Colist}(C) \to C$, $\mathsf{cur} = \mathsf{fst}\,.\,\mathsf{out}$, $\mathsf{prev} : \mathsf{Colist}(C) \to 1 + \mathsf{Colist}(C)$, $\mathsf{prev} = \mathsf{snd}\,.\,\mathsf{out}$, which return the current value and the colist of previous values of a given colist, respectively. The operational semantics becomes

$$f(\mathsf{in}\, t) \to \varphi\Big(\mathsf{mapN}\, \big(\, \mathsf{unfold}_{\nu}\, \mathsf{mapN}^{\times}\, \langle f, \mathsf{pred} \rangle\big)\, t\Big)$$

where $\mathsf{pred} : \mathsf{Nat} \to 1 + \mathsf{Nat}$ is the predecessor function on naturals defined as $\mathsf{pred} = \mathsf{in}^{-1}\, \mathsf{mapN}$. The anamorphism $g = \mathsf{unfold}_{\nu}\, \mathsf{mapN}^{\times}\, \langle f, \mathsf{pred} \rangle$ behaves as follows: $\mathsf{out}(g\ 0) \to^{\star} (f0, \mathsf{error})$, $\mathsf{out}(g\ (\mathsf{suc}\, x)) \to^{\star} \big(f(\mathsf{suc}\, x), \mathsf{inr}(g\ x)\big)$.

Therefore g is esentially the history $\mathsf{rcd}\, f$ given in definition 3.1. A histomorphism f, is easier to understand if the step function φ is decomposed as the copair

$\varphi = [z, s]$ with $z : 1 \to C$, $s : \mathsf{Colist}(C) \to C$. In this case f behaves as follows: $f\ 0 \to z\star$, $f\ (\mathsf{suc}\, n) \to s(g\, n)$. Thus, f corresponds to the function defined by course-of-value iteration with basis $z\star$ and step function s according to definition 3.1. The record function $\mathsf{rcd}\, f$ is now faithfully implemented and the scheme of course-of-value iteration on naturals is fully available.

Example 5.1 The Fibonacci function $\mathsf{fibo} : \mathsf{Nat} \to \mathsf{Nat}$ such that $\mathsf{fibo}\, 0 = 1$, $\mathsf{fibo}\, 1 = 1$, $\mathsf{fibo}(n+2) = \mathsf{fibo}(n) + \mathsf{fibo}(n+1)$ is programmed as $\mathsf{fibo}\, n = \mathsf{cvfold}_\mu\, \mathsf{mapN}\, [z, s], n$ where $z = \lambda\_.1$ is the constant function returning 1 and $s = \lambda x.\mathsf{case}\big(\mathsf{prev}\, x, y.1, z.(\mathsf{cur}\, x)+(\mathsf{cur}\, z)\big)$. It can be verified that this is a sound definition, for its behaviour is $\mathsf{fibo}\, 0 \to 1$, $\mathsf{fibo}\, 1 \to 1$, $\mathsf{fibo}(n+2) \to \mathsf{fibo}(n)+\mathsf{fibo}(n+1)$.

5.2 Course-of-Value Iteration as Syntactic Sugar

As we have seen in the previous section, the principle of course-of-value iteration can be faithfully modelled in a type system. However a natural question when proposing a new language is if the new constructors could be defined, as syntactic sugar, in a basis language, which usually already has some important properties like safety or termination. It turns out that this holds in our case. Consider the following alternative reduction rule for course-of-value iteration:

$$\mathsf{cvfold}_\mu\, m\ \varphi\ (\mathsf{in}\, t) \to \mathsf{cur}\,.\,\mathsf{fold}_\mu\, m\, (\mathsf{out}^{-1}\ \ m^\times\,.\,\langle \varphi, \mathsf{Id}\rangle)\, (\mathsf{in}\, t)$$

The system obtained from CCVT by abandoning its reduction rule in favor of the above rule is called CCVT2.
A nice explanation for this reduction in the case of Nat is the following: Given $f : \mathsf{Nat} \to C$, its record $\mathsf{rcd}\, f : \mathsf{Nat} \to \mathsf{Colist}\, C$ can be defined either by coiteration as in definition 3.1, or iteratively as: $\mathsf{rcd}\ f\, 0 = [f\, 0]$, $\mathsf{rcd}\, f\, (n+1) = (f\, (n+1) : (\mathsf{rcd}\, f\, n))$. This last choice corresponds to the above rule, since if we put $f : \mathsf{Nat} \to C$ as before, with $\varphi = [z, s] : 1 + \mathsf{Colist}\, C \to \mathsf{Colist}\, C$ and $\mathsf{rcd}\, f = \mathsf{fold}_\mu\, \mathsf{mapN}\, (\mathsf{cons}\,.\langle \varphi, \mathsf{Id}\rangle) : \mathsf{Nat} \to \mathsf{Colist}\, C$, where $\mathsf{cons} = \mathsf{out}^{-1}\, \mathsf{mapN}^\times$ is the colist constructor, then we get the following behaviour: $\mathsf{rcd}\ f\, 0 \to \mathsf{cons}\, (z\star, \mathsf{error})$, which yields the single colist $[z\star]$, corresponding to $[f\, 0]$. Similarly, $\mathsf{rcd}\ f\, (\mathsf{suc}\, n) \to \mathsf{cons}\, (s(\mathsf{rcd}\ f\, n), \mathsf{inr}(\mathsf{rcd}\ f\, n))$ yields essentially the colist $(f\, (\mathsf{suc}\, n) : \mathsf{rcd}\ f\, n)$. Therefore we can restate the reduction rule as $f(\mathsf{in}\, t) \to \mathsf{cur}(\mathsf{rcd}\ f\ (\mathsf{in}\, t))$ which implies $f\, n \to \mathsf{cur}(\mathsf{rcd}\ f\, n)$ for $n : \mathsf{Nat}$. This looks really inefficient, for it reduces the problem of constructing the value of f at n to the construction of the whole record up to n and only then taking its head, which is the needed current value.

5.3 Memoizing Histomorphisms

The above inefficiency can be avoided by a dynamic programming technique called memoization which consists in storing previously computed values of a function instead of recalculating them. This can be achieved by simple coiteration as follows, according to the treatment in [8]. The function $\mathsf{tabN} : (\mathsf{Nat} \to C) \to \mathsf{stream}\, C$, defined coiteratively as $\mathsf{tabN}\, f = (f\, 0 : \mathsf{tabN}(f.\mathsf{suc}))$, constructs the whole record of the function f, namely $\mathsf{tabN}\, f = [f\, 0, f\, 1, f\, 2, \ldots]$. To obtain a particular value $f\, n$ we need a consulting function $\mathsf{consN} : \mathsf{Nat} \to \mathsf{stream}\, C \to C$ such that $\mathsf{consN}\, n\, s$

returns the nth element of the stream s. This function is defined iteratively as $\mathsf{consN}\,0 = \mathsf{head}$, $\mathsf{consN}\,(n+1) = (\mathsf{consN}\,n).\,\mathsf{tail}$. Finally we define $\mathsf{memo} : (\mathsf{Nat} \to C) \to \mathsf{Nat} \to C$ by $\mathsf{memo} = (\mathsf{flip}\,\mathsf{consN}).\,\mathsf{tabN}$, where $\mathsf{flip} = \lambda f \lambda x \lambda y. f\,y\,x$. Thus, $\mathsf{memo}\,f$ is a memoized efficient version of f.

Remark 5.2 The alternative dynamic semantics for course-of-value iteration in CCVT2 is not directly acceptable from a practical point of view but, as the above discussion shows, efficiency can be gained by implementing memoization. This can also be done with the previous semantics which is also inefficient.
From the theoretical point of view this semantics is an improvement over the former, for now the histomorphisms are just syntactic sugar definable on the basic system of section 4.1. We can observe that the reduction rule of CCVT2 consists in desugaring the cvfold_μ constructor exactly as in proposition 3.6.

5.4 On Safety and Termination

Type safety is a desirable property for a language, usually composed by subject reduction (type preservation) and evaluation progress.

Proposition 5.3 (Subject reduction) *Systems* CCVT, CCVT2 *enjoy subject reduction. That is, if* $\Gamma \vdash e : A$ *and* $e \to e'$ *according to* CCVT *or* CCVT2 *then* $\Gamma \vdash e' : A$

Proof This can be easily achieved in a similar way to the proof for the systems in [17], which is based in the one for system F given in [11]. Observe that the property is not trivial as it would be for a Church-style system. The systems are presented here in Curry-style and therefore the rules for polymorphic typing are not syntax-directed. □

Proposition 5.4 (Progress) *Systems* CCVT, CCVT2 *enjoy progress of the evaluation relation. That is, if* $\vdash e : A$ *then either* e *is a value or there exists* e' *such that* $e \to e'$.

Proof Straightforward induction on $\vdash$. □

On the other hand the termination or strong normalisation property for a language generates some interesting remarks. From the point of view of language design, this is not a decisive issue although its merits are also defendable, see [20]. From the point of view of program analysis the undecidability of the halting problem leaves no hope for an automated universal termination checker. We have to put up with a termination checker working for some restricted class of programs together with human reasoning. For example the prominent system $\mathcal{T}$ of Gödel captures the class of primitive recursive functions on natural numbers, and the basic system of section 4.1 guarantees termination for the class of (co)iteratively defined functions on every monotone (co)inductive type. Termination checkers are either syntax-directed using a well founded order on terms for instance, or type-based where termination is ensured by a type system; that is, if a program passes the type-checker, then it will terminate on all inputs. This is the kind of termination we pursuit here. For a deep discussion on termination we point to [1].

Next, we make some remarks related to termination of our systems.

- *The languages* $\mathsf{CCVT}, \mathsf{CCVT2}$ *are not terminating.* This is exclusively due to the reduction rules for (co)inductive inversion. Define $\mathsf{T} = \nu(\lambda X.X \to 1), m = \lambda f \lambda x \lambda y.\star$, $\omega = \lambda x.(\mathsf{out}\, x)x$, $\Omega = \omega(\mathsf{out}^{-1}\ m\ \omega)$. We have the typings $\vdash m : (\lambda X.X \to 1)\,\mathsf{mon}$, $\vdash \omega : \mathsf{T} \to 1, \vdash \mathsf{out}^{-1}\ m\ \omega : \mathsf{T}$ and $\vdash \Omega : 1$. With the rule $\mathsf{out}(\mathsf{out}^{-1}\ m\ t) \to t$ we get $\Omega \to^{+} \Omega$ as follows: $\Omega \to (\mathsf{out}(\mathsf{out}^{-1}\ m\ \omega))(\mathsf{out}^{-1}\ m\ \omega) \to \omega(\mathsf{out}^{-1}(m, \omega)) \equiv \Omega$. The same happens with the rule $\mathsf{in}^{-1}\ m\ (\mathsf{in}\, t) \to t$. This phenomenon was first noticed in [14] for fixed-point-types. It is worth nothing that the type T is non-positive and trivial. A natural question is asking if there would be a positive type causing no termination, but we are not aware of such example. However we can mention that, for the case of positive types the out^{-1} constructor and its operational semantics can easily be defined in an extension of F with primitive corecursion, which we prove to be terminating in [16]. For a further discussion on this see appendix B of [12].
- *The languages* $\mathsf{CCVT}, \mathsf{CCVT2}$ *could be terminating.* This is achieved by changing the operational semantics for inversion as follows:

$$\mathsf{out}(\mathsf{out}^{-1}\ m\ t) \to m(\lambda zz)t \qquad \mathsf{in}^{-1}\ m\ (\mathsf{in}\, t) \to m(\lambda zz)t$$

  Observe that categorically the expressions $m(\lambda zz)t$ and t are equivalent as m plays the role of a functor on arrows, and λzz is the identity arrow. Therefore, $m(\lambda zz)t = t$ is a consequence of the first functor law $F(\mathsf{Id}) = \mathsf{Id}$.
  The termination of these new systems is carried out by a type-respecting and reduction-preserving embedding into a terminal system developed in [17], which is basically the one described in section 4.1 plus the above operational rules for inversion. The essential part of this embedding can be read from the definition of histomorphisms by catamorphisms (simple iteration) of proposition 3.6.
- *The price of termination.* In our type systems the reduction $m(\lambda zz)t \to t$, needed for a correct operational semantics, does not hold in general. An interesting task is to assert when it holds, for which we can give the following answer: If $F = \lambda X G$ and X occurs only positively in G, then the first functor law can be ensured for so-called canonical witnesses defined recursively according to the shape of G (see [13,16]). However, some extensionality principles are needed. That is, if m is canonical then $m(\lambda zz)t \to_{\beta\eta} t$. For instance, for natural numbers the witness $\mathsf{mapN} = \lambda f.[\mathsf{inl}, \mathsf{inr}\, .f]$ is canonical and we have $\mathsf{mapN}(\lambda zz)t \to^{\star}_{\beta} \mathsf{case}(t, y.\, \mathsf{inl}\, y, z.\, \mathsf{inr}\, z) \to_{\eta} t$. The price is high, for it is well-known that subject reduction fails already for system F (see [16], page 65) when adding the rule $\lambda x.fx \to_{\eta} f,\ x \notin FV(f)$.

From the above remarks it is clear that, for the current extensions, termination should not be pursuit. However, there is another way to define course-of-value iteration, which is based on a fixed-point logic, is terminating and does not require inversion principles.

6 The Logical Approach

From the logical point of view, (co)inductive types correspond to fixed-points of monotone operators $F = \lambda X.G$. In particular the course-of-value least-fixed point,

denoted by $\mu^\star F$ is algebraically defined as the least element of the set of all $R's$ such that $F(\nu(\lambda X.R \wedge FX))$ is less than R. Observe that this condition is analogous to the definition of a *cv*-algebra. Based on this idea, we describe another type system.

6.1 The Iterative System LCVT

This extension is essentially the natural deduction system involving the course-of-value logical constant $\mu^\star$ in [22], gained by using the Curry-Howard correspondence. We call this a system with Logical Course-of-Value (co)inductive Types. In comparison to our previous system CCVT we will have a new kind of inductive type denoted $\mu^\star F$, which is characterized by a different way of constructing its inhabitants. Moreover, there will be no need to use the inverse constructors $\mathsf{in}^{-1}, \mathsf{out}^{-1}$ anymore, which are the source of non-termination in the previous systems.

LCVT is obtained by modifying the rule for course-of-value-iteration (μE^{cv}) of page 103, replacing μF by $\mu^\star F$ and adding a new unary constructor cvin with the below mentioned typing rule corresponding to folding of the course-of-value least fixed point:

$$\frac{\Gamma \vdash t : F(F^\nu(\mu^\star F))}{\Gamma \vdash \mathsf{cvin}\, t : \mu^\star F}\ (\mu I^{cv})$$

where $F^\nu = \lambda Z.\nu(\lambda X.Z \times F\, X)$ as before. Observe that we are using a conventional coinductive type ν to define the new course-of-value fixed point $\mu^\star$.

The operational semantics is given by the following rule:

$$\mathsf{cvfold}_\mu\, m\, \varphi\, (\mathsf{cvin}\, t) \to \varphi\Big(m\, \big(\mathsf{unfold}_\nu\, m^\times\, \langle(\mathsf{cvfold}_\mu\, m\, \varphi)\,.\,\mathsf{cur}, \mathsf{prev}\rangle\big)\, t\Big)$$

where $\mathsf{cur}, \mathsf{prev}, m^\times$ are defined as before. This rule is easier to understand if we set $f = \mathsf{cvfold}_\mu\, m\, \varphi$, becoming $f(\mathsf{cvin}\, t) \mapsto_\beta \varphi\Big(m\, (\mathsf{unfold}_\nu\, m^\times \langle f.\,\mathsf{cur}, \mathsf{prev}\rangle)\, t\Big)$. Let us see now its meaning on course-of-value natural numbers.

6.2 Course-of-Value Natural Numbers

The inductive type of course-of-value natural numbers is defined as $\mathsf{Nat}^\star = \mu^\star(\lambda X.1{+}X)$. This is not the usual type of natural numbers. In particular, the usual succesor function is not directly encoded by cvin, whose native encoded constructors are $0 : \mathsf{Nat}^\star$, $0 = \mathsf{cvin}(\mathsf{inl}\star)$ and $\mathsf{suc}^\star : \mathsf{Colist}(\mathsf{Nat}^\star) \to \mathsf{Nat}^\star$, $\mathsf{suc}^\star = \mathsf{cvin}\ .\ \mathsf{inr}$. This last function defines a course-of-value succesor receiving a colist of natural numbers and encapsulating it to form a new natural number. The operational semantics captures yet another scheme of course-of-value recursion based on a function $\mathsf{rcd}^\star : (\mathsf{Nat}^\star \to C) \to \mathsf{Colist}\,\mathsf{Nat}^\star \to \mathsf{Colist}\,C$ which constructs the record of a function on a given colist of arguments, namely $\mathsf{rcd}^\star\, f\, [a_1, \ldots, a_k] = [f\, a_1, \ldots, f\, a_k]$. Thus, $\mathsf{rcd}^\star$ is a map function on colists. Given $f = \mathsf{cvfold}_\mu\, \mathsf{mapN}\, \varphi : \mathsf{Nat}^\star \to C$ the function $\mathsf{rcd}^\star\ f$ is implemented as $\mathsf{rcd}^\star\ f = \mathsf{unfold}_\nu\, \mathsf{mapN}^\times \langle f.\,\mathsf{cur}, \mathsf{prev}\rangle$ and if $\varphi = [z, s]$ the reduction rule yields $f\, 0 \to z\star$ and $f(\mathsf{suc}^\star\ \ell) \to s(\mathsf{rcd}^\star\ f\, \ell)$. To obtain a scheme working on numbers and not colists we need to implement the usual succesor function. This can be done as in [22], by defining $\mathsf{suc} = \mathsf{suc}^\star\,.\,\mathsf{tcl}$ where $\mathsf{tcl} : \mathsf{Nat}^\star \to \mathsf{Colist}(\mathsf{Nat}^\star)$ receives a number n and constructs the colist $[n, n-1, \ldots, 0]$, therefore the number $\mathsf{suc}\, n$ encapsulates its colist of predecessors

just as a set-theoretic ordinal is simply the set of its predecessors. Thus, we can implement a course-of-value iterative function corresponding to the following scheme working with numbers directly: Given $a : \mathsf{Nat}^\star$, $g : \mathsf{Colist}\, C \to C$ there is a function $f : \mathsf{Nat}^\star \to C$ such that $f\, 0 = a$, $f\,(n+1) = g\,(\mathsf{rcd}^\star\ f\,[n, \ldots, 0])$. A corresponding `cvfold` operator can be easily defined in Haskell. In particular, if the scheme is well implemented in the type system, the Fibonacci function definition of example 3.2 still works. But once again the operational correctness of the whole scheme depends on a good predecessor function.

6.2.1 The Iterative Predecessor

The predecessor function is constructed similarly to the succesor by first defining, through course-of-value iteration, a function $\mathsf{pred}^\star : \mathsf{Nat}^\star \to 1 + \mathsf{Colist}(\mathsf{Nat}^\star)$ which destructs a non-zero natural by returning the colist of all its predecessors. The usual predecessor can then be defined as in [22], by composing $\mathsf{pred}^\star$ with the application of the map function to the cur colist destructor. Formally we have $\mathsf{pred} = (\mathsf{mapN}\ \mathsf{cur})\,.\,\mathsf{pred}^\star$, $\mathsf{pred}^\star = \mathsf{cvfold}_\mu\, \mathsf{mapN}\, \varphi$, $\varphi = \mathsf{mapN}\,(\mathsf{unfold}_\nu\, \mathsf{mapN}^\times\, \langle \mathsf{cvin}\,.\,\mathsf{cur}, \mathsf{prev}\rangle)$. Unfortunately this predecessor definition does not have the required behaviour, for we only get $\mathsf{pred}(\mathsf{suc}\, n) \to^\star \mathsf{inr}\,(\mathsf{cvin}(\mathsf{pred}^\star\, n))$ but the term $\mathsf{cvin}(\mathsf{pred}^\star\, n)$ cannot be further reduced to n as desired. However the reduction $\mathsf{cvin}(\mathsf{pred}^\star\, n) \to n$ seems correct, since the left hand side is constructing a new inhabitant of $\mathsf{Nat}^\star$ from the ancestral $\mathsf{pred}^\star\, n$, which should be just n. This reduction seems to be an η rule modelling some kind of uniqueness. However, adding it to the system, apart from being a tailor-made solution, contradicts the belief that η-rules are not rules of computation, which up to our knowledge is still a piece of folklore, although it seems to be confirmed in the case of conventional (co)inductive types (see [9]).

7 Towards a System with Course-of-Value Primitive Recursion

It is well-known that the inefficiency of an iterative predecessor vanishes using primitive recursion instead of simple iteration to define it (see [18]). Therefore, we think that the needed well-behaved predecessor for course-of-value natural numbers can be obtained using primitive course-of-value recursion. Moreover, this would solve the problem not only for natural numbers, but also for the destructor of any course-of-value monotone inductive type.

Conjecture 7.1 *There is a safe and terminating type system modelling course-of-value primitive recursion on monotone inductive types. Furthermore, this principle would allow us to define operationally well-behaved inductive destructors without requiring the use of η-rules.*

A system including course-of-value primitive recursion has been mentioned in [23,22] but, up to our knowledge, it was never developed. We are currently designing

such a system. For the time being we have the following rule for the static semantics.

$$\frac{\Gamma \vdash t : \mu^{\star} F \qquad \Gamma \vdash m : F\,\mathsf{mon} \qquad \Gamma \vdash \varphi : F\big(\mu^{\star} F \times F^{\nu}(C)\big) \to C}{\Gamma \vdash \mathsf{rcvfold}_{\mu}\, m\, \varphi\, t : C}\ (\mu E^{cvr})$$

The difference with iteration arises on the signature of the function φ which now requires a pair whose first element belongs to the inductive type, over which, we are recursing. For the case of $\mathsf{Nat}^{\star}$ we get $\varphi : 1 + (\mathsf{Nat}^{\star} \times \mathsf{Colist}\, C) \to C$, this function encodes the value c and the step function g of the following principle: given a value $c : C$, and a function $g : \mathsf{Nat}^{\star} \times \mathsf{Colist}\, C \to C$ there is a function $f : \mathsf{Nat}^{\star} \to C$ such that $f\, 0 = c,\ f\,(n+1) = g\,(n, \mathsf{rcd}^{\star}\, f\,[n, .., 0])$.

Example 7.2 An interesting example fitting this last scheme is the function $\mathsf{bp} : \mathsf{Nat}^{\star} \to \mathsf{Nat}^{\star}$ returning the number of binary partitions of a number n, that is, the number of ways n can be decomposed as a sum of powers of two, ignoring the order. The usual definition is $\mathsf{bp}\ 0 = 1,\ \mathsf{bp}\,(n+1) = \mathsf{if}\ \mathsf{even}\, n\ \mathsf{then}\ \mathsf{bp}\, n\ \mathsf{else}\ (\mathsf{bp}\, n) + (\mathsf{bp}\,(n\, div\, 2))$ and can be implemented by course-of-value primitive recursion taking $c = 1$ and $g\,(n, \ell) = \mathsf{if}\ (\mathsf{even}\, n)\ \mathsf{then}\ \mathsf{head}\ \ell\ \mathsf{else}\ (\mathsf{head}\ \ell) + \ell\,!!\,(n\, div\, 2)$, where $\ell\,!!\, n$ returns the nth element of ℓ counting from zero.

Regarding the operational semantics it is clear that it should be analogous to the one for the iterative cvfold_{μ}, but instead of the simple coiterator unfold_{ν} we would need a primitive corecursor (apomorphism). Further research on this subject is being done.

8 Final Remarks

We have presented three different safe type systems comprising course-of-value iteration schemes. The first two are based on the categorical constructor modelling course-of-value iteration called histomorphism, and do not change the definition of datatypes as initial algebras of functors, whereas the last one comes from a logical approach based on fixed-points and changes the representation of datatypes. As all systems are safe, they can be used as prototypes for a programming language with an explicit higher-order course-of-value fold operator. Regarding termination (strong normalisation), though we think this must not be a decisive issue in the design and use of a language, we adopt the point of view of type-based termination and pursuit this property finding some interesting remarks: systems $\mathsf{CCVT}, \mathsf{CCVT2}$ are not terminating though this property can be gained by a small modification to the operational semantics, yet at a high price, including the need for some extensionality η-rules which would jeopardise the safety of the languages, as mentioned in section 5.4. On the other hand, system LCVT is terminating, for it can be embedded in the basic terminating (co)iterative system of section 4.1. For the case of positive (co)inductive types this result appears in [22]. Nevertheless, the predecessor function needed to ensure the correctness of the course-of-value scheme on

naturals, is faulty. We could overcome this defect by adding an η-rule at the price of destructing the safety of the language. A definitive solution is conjectured by the use of primitive course-of-value recursion, a useful principle that we are currently studying. Furthermore, the corecursive counterpart of the systems discussed here is straightforward definable by duality (for the logical approach this has been done in [22]) and implements the scheme of course-of-value corecursion captured by a categorical construction called futumorphism in [21]. This principle allows us to specify arguments for the construction of remaining parts of a coinductive structure at arbitrary stages instead of being forced to specify for the following stage already. Concerning future work, apart from developing more examples and a Haskell implementation, we want to settle the exact relation between the categorical and logical approaches, as well as investigate other recursion schemes such as hylomorphisms (see [15,3]), a more recent scheme called dynamorphism, developed in [10], and the very efficient TABA-pattern of [5], not to mention Mendler-style course-of-value recursion schemes. Another line of research is the use of clausular (co)inductive types (see [7,16,17]), which improves the style of programming as well as the definition of witnesses of monotonicity by avoiding the use of injections and projections.

Acknowledgements

We are thankful to our anonymous referees for the helpful comments and suggestions regarding the contents and presentation of this paper.

References

[1] Abel, A.,"A Polymorphic Lambda-Calculus with Sized Higher-Order Types". Dissertation. Universität München. Germany 2006. URL: http://www.tcs.informatik.uni-muenchen.de/~abel/diss.pdf

[2] Abel, A., R. Matthes, T. Uustalu. *Iteration and Coiteration Schemes for Higher-Order and Nested Datatypes*. Theoretical Computer Science 333(2005). 3-66.

[3] Cunha, M.A., *Recursion Patterns as Hylomorphisms*. Technical report DI-PURe-03.11.01, Department of Informatics, University of Minho. November 2003.

[4] Crole, R.L., "Categories for Types". Cambridge Mathematical Textbooks. Cambridge University Press, 1993.

[5] Danvy, O., M. Goldberg, *There and Back Again*. BRICS Report Series RS-05-3. University of Aarhus. January 2005.

[6] Geuvers, H., *Inductive and coinductive types with iteration and recursion*. In Nordström B., K. Petterson, G. Plotkin, Eds. Proceedings of the 1992 Workshop on Types for Proofs and Programs. Båstad, Sweden June 1992, pp. 183-207.

[7] Hagino, T., *A Typed Lambda Calculus with Categorical Type Constructors*. In Pitt D.H., Poigné, A., Rydeheard, D.E. (eds). Category Theory and Computer Science. LNCS, vol 283. pp. 140–157. Springer, Heidelberg (1987).

[8] Hinze, R., *Memo functions, polytipically!*. In Johan Jeuring, editor, Proceedings of the Second Workshop on Generic Programming, WGP 2000, Ponte de Lima, Portugal, 6th July 2000.

[9] Howard, B.T., "Fixed Points and Extensionality in Typed Functional Programming Languages". Ph. D. Thesis, Stanford University 1992.

[10] Kabanov, J., V. Vene, *Recursion Schemes for Dynamic Programming*. In: Uustalu T., (ed) Proc. of 8th Int. Conf. on Mathematics of Program Construction, MPC 06. LNCS, vol. 4014, pp. 235-252. Springer, Heidelberg (2006).

[11] Krivine, J.L.. "Lambda-Calculus, Types and Models". Ellis Horwood Series in Computers and their Applications. Ellis Horwood, Masson 1993.

[12] Matthes, R., "Extensions of System F by Iteration and Primitive Recursion on Monotone Inductive Types", Dissertation Universität München, 1999. URL: http://www.tcs.informatik.uni-muenchen.de/~matthes/dissertation/matthesdiss.ps.gz

[13] Matthes, R., *Monotone (co)inductive types and positive fixed-point types.* Theoretical Informatics and Applications 33(4-5) pp. 309-328 (1999).

[14] Matthes, R., *Monotone fixed-point types and strong normalization.* In Georg Gottlob, Etienne Grandjean, and Katrin Seyr, editors, Computer Science Logic, 12th International Workshop, Brno, Czech Republic, August 24-28, 1998, Proceedings, volume 1584 of Lecture Notes in Computer Science, pages 298-312. Springer Verlag, 1999.

[15] Meijer, E., M. Fokkinga, R. Paterson, *Functional programming with bananas, lenses, envelopes and barbed wire.* In: Hughes, J. (ed) FPCA 91. LNCS, vol. 523, pp 124–144. Springer, Heidelberg (1991).

[16] Miranda-Perea, F.E., "On Extensions of AF2 with Monotone and Clausular (Co)inductive Definitions". Dissertation, Ludwig-Maximilians-Universität München. Germany 2004. URL: http://edoc.ub.uni-muenchen.de/2855/

[17] Miranda-Perea, F.E., *Two Extensions of System F with (Co)iteration and Primitive (Co)recursion Principles.* Submitted to Theoretical Informatics and Applications. February 2007. A preliminary draft is available on http://www.matematicas.unam.mx/favio/drafts/stmunu.pdf

[18] Parigot, M., *On the Representation of Data in Lambda-Calculus.* In Egon Börger, Hans Kleine Büning, Michael M. Richter (Eds.): CSL '89, 3rd Workshop on Computer Science Logic, Kaiserslautern, Germany, October 2-6, 1989, Proceedings. Lecture Notes in Computer Science 440 Springer Verlag 1990.

[19] Simpson, A., *Recursive Types in Kleisli Categories.* Unpublished draft. URL: http://homepages.inf.ed.ac.uk/als/Research/kleisli.ps.gz

[20] Turner, D., *Total Functional Programming.* Journal of Universal Computer Science, v. **10**, n. **7**. pp 751–768. 2004.

[21] Uustalu, T., V. Vene, *Primitive (co)recursion and course-of-value (co)iteration, categorically.* INFORMATICA, v. **10**, n.**1**, pp. 5-26. 1999.

[22] Uustalu, T., V. Vene, *Least and greatest fixed-points in intuitionistic natural deduction.* Theoretical Computer Science, v. **272**, n. **1-2**, pp. 315-339, 2002.

[23] Uustalu, T., "Natural deduction for intuitionistic least and greatest fixedpoint logics, with an application to program construction" (PhD thesis). Dissertation TRITA-IT AVH 98:03, Dept. of Teleinformatics, Royal Inst of Technology (KTH), Stockholm, 1998.

[24] Vene, V., "Categorical programming with inductive and coinductive types". Diss. Math. Uni. Tartuensis, v. 23, Uni. of Tartu, Aug. 2000.

A System F

We describe here system F of Girard and Reynolds in a Curry-style presentation. For convenience we add sums, products and unit type as primitives.

- *Types.* Built from an infinite set of type variables denoted by metavariable X.

$$A, B, C, F, G ::= X \mid A \to B \mid \forall X A \mid A + B \mid A \times B \mid 1$$

- *Terms.* Built from an infinite set of term variables denoted by metavariable x.

$$t, r, s ::= x \mid \lambda x r \mid r s \mid \mathsf{inl}\, r \mid \mathsf{inr}\, s \mid \mathsf{case}(r, x.s, y.t) \mid (r, s) \mid \mathsf{fst}\, r \mid \mathsf{snd}\, r \mid \star$$

- *Contexts.* Finite sets Γ of pairs of the form $x : A$. The expression $\Gamma, x : A$ denotes the context $\Gamma \cup \{x : A\}$ always assuming that a pair $x : B$ was not previously declared in Γ.

- *Typing rules.* Inference rules of the form $\Gamma \vdash t : A$ denoting that t is a well-formed term of type A in context Γ.

$$\frac{}{\Gamma, x : A \vdash x : A}\ (Var) \qquad \frac{}{\Gamma \vdash \star : 1}\ (Unit)$$

$$\frac{\Gamma, x : A \vdash r : B}{\Gamma \vdash \lambda x r : A \to B}\ (\to I) \qquad \frac{\Gamma \vdash r : A \to B \quad \Gamma \vdash s : A}{\Gamma \vdash rs : B}\ (\to E)$$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t : \forall X A}\ (\forall I) \qquad \frac{\Gamma \vdash t : \forall X A}{\Gamma \vdash t : A[X := F]}\ (\forall E)$$

$$\frac{\Gamma \vdash r : A}{\Gamma \vdash \mathsf{inl}\, r : A + B}\ (+I_L) \qquad \frac{\Gamma \vdash r : B}{\Gamma \vdash \mathsf{inr}\, r : A + B}\ (+I_R)$$

$$\frac{\Gamma \vdash r : A + B \quad \Gamma, x : A \vdash s : C \quad \Gamma, y : B \vdash t : C}{\Gamma \vdash \mathsf{case}(r, x.s, y.t) : C}\ (+E)$$

$$\frac{\Gamma \vdash r : A \quad \Gamma \vdash s : B}{\Gamma \vdash (r, s) : A \times B}\ (\times I) \qquad \frac{\Gamma \vdash s : A \times B}{\Gamma \vdash \mathsf{fst}\, s : A}\ (\times E_L) \qquad \frac{\Gamma \vdash s : A \times B}{\Gamma \vdash \mathsf{snd}\, s : B}\ (\times E_R)$$

- *Reduction.* The operational semantics is given by the one-step β-reduction relation $t \to t'$ defined as the closure of the following axioms under all term formers.

$$\begin{aligned}
(\lambda x r)s &\to r[x := s] \\
\mathsf{case}(\mathsf{inl}\, r, x.s, y.t) &\to s[x := r] \\
\mathsf{case}(\mathsf{inr}\, r, x.s, y.t) &\to t[y := r] \\
\mathsf{fst}\, (r, s) &\to r \\
\mathsf{snd}\, (r, s) &\to s
\end{aligned}$$