Language Constructs for Non-Well-Founded Computation

Jean-Baptiste Jeannin[1], Dexter Kozen[1], and Alexandra Silva[2]

[1] Cornell University, Ithaca, NY 14853-7501, USA
{jeannin,kozen}@cs.cornell.edu

[2] Institute for Computing and Information Sciences, Radboud University Nijmegen, Postbus 9010, 6500 GL Nijmegen, The Netherlands
alexandra@cs.ru.nl

Abstract. Recursive functions defined on a coalgebraic datatype C may not converge if there are cycles in the input, that is, if the input object is not well-founded. Even so, there is often a useful solution. Unfortunately, current functional programming languages provide no support for specifying alternative solution methods. In this paper we give numerous examples in which it would be useful to do so: free variables, α-conversion, and substitution in infinitary λ-terms; halting probabilities and expected running times of probabilistic protocols; abstract interpretation; and constructions involving finite automata. In each case the function would diverge under the standard semantics of recursion. We propose programming language constructs that would allow the specification of alternative solutions and methods to compute them.

Keywords: coalgebraic types, functional programming, recursion.

1 Introduction

Coalgebraic datatypes have become popular in recent years in the study of infinite behaviors and non-terminating computation. One would like to define functions on coinductive datatypes by structural recursion, but such functions may not converge if there are cycles in the input; that is, if the input object is not well-founded. Even so, there is often a useful solution that we would like to compute.

For example, consider the problem of computing the set of free variables of a λ-term. In pseudo-ML, we might write

```
type term =                        let rec fv = function
  | Var of string                    | Var v -> {v}
  | App of term * term               | App (t1,t2) -> (fv t1) ∪ (fv t2)
  | Lam of string * term             | Lam (x,t) -> (fv t) − {x}
```

and this works provided the argument is an ordinary (well-founded) λ-term. However, if we call the function on an infinitary term (λ-coterm), say

```
 let rec t = App (Var "x", App (Var "y", t))
```

$$\begin{tikzcd}
 & \bullet \arrow[dl] \arrow[dr] & \\
x & & \bullet \arrow[dl] \arrow[ul, bend right=90, looseness=2] \\
 & y &
\end{tikzcd} \tag{1}$$

then the function will diverge, even though it is clear the answer should be $\{x, y\}$. Note that this is not a corecursive definition: we are not asking for a greatest solution or a unique solution in a final coalgebra, but rather a least solution in a different ordered domain from the one provided by the standard semantics of recursive functions. The standard semantics gives us the least solution in the flat Scott domain $(\mathcal{P}(\mathbf{string})_\perp, \sqsubseteq)$ with bottom element $\perp$ representing nontermination, whereas we would like the least solution in a different CPO, namely $(\mathcal{P}(\mathbf{string}), \subseteq)$ with bottom element $\varnothing$.

The coinductive elements we consider are always *regular*, that is, they have a finite but possibly cyclic representation. This is different from a setting in which infinite elements are represented lazily. A few of our examples, like substitution, could be computed by lazy evaluation, but most of them, for example free variables, could not.

Theoretically, the situation is governed by diagrams of the form

$$\begin{tikzcd}
C \arrow[r, "h"] \arrow[d, "\gamma"'] & A \\
FC \arrow[r, "Fh"'] & FA \arrow[u, "\alpha"']
\end{tikzcd} \tag{2}$$

describing a recursive definition of a function $h : C \to A$. Here F is a functor describing the structure of the recursion. To apply h to an input x, the function $\gamma : C \to FC$ identifies the base cases, and in the recursive case prepares the arguments for the recursive calls; the function $Fh : FC \to FA$ performs the recursive calls; and the function $\alpha : FA \to A$ assembles the return values from the recursive calls into final value $h(x)$.

A canonical example is the usual factorial function

```
let rec factorial = function
  | 0 -> 1
  | n -> n * factorial (n-1)
```

Here the abstract diagram (2) becomes

$$\begin{tikzcd}
\mathbb{N} \arrow[r, "h"] \arrow[d, "\gamma"'] & \mathbb{N} \\
\mathbb{1} + \mathbb{N} \times \mathbb{N} \arrow[r, "\mathsf{id}_{\mathbb{1}} + \mathsf{id}_{\mathbb{N}} \times h"'] & \mathbb{1} + \mathbb{N} \times \mathbb{N} \arrow[u, "\alpha"']
\end{tikzcd} \tag{3}$$

where the functor is $FX = \mathbb{1} + \mathbb{N} \times X$ and γ and α are given by:

$$\begin{aligned} \gamma(0) &= \iota_0() & \qquad \alpha(\iota_0()) &= 1 \\ \gamma(n+1) &= \iota_1(n+1, n) & \qquad \alpha(\iota_1(c, d)) &= cd \end{aligned}$$

where ι_0 and ι_1 are injectors into the coproduct. The fact that there is one recursive call is reflected in the functor by the single X occurring on the right-hand side. The function γ determines whether the argument is the base case 0 or the inductive case $n+1$, and in the latter case prepares the recursive call. The function α combines the result of the recursive call with the input value by multiplication. In this case we have a unique solution, which is precisely the factorial function.

Theoretical accounts of this general idea have been well studied [1,2,3,9]. Most of this work is focused on conditions ensuring unique solutions, primarily when C is well-founded or when A is a final coalgebra. The account most relevant to this study is the work of Adámek et al. [2], in which a canonical solution can be specified even when it is not unique, provided various desirable conditions are met; for example, when A is a complete CPO and α is continuous, or when A is a complete metric space and α is contractive. Also closely related are the work of Widemann [10] on coalgebraic semantics of recursion and cycle detection algorithms and the work of Simon et al. [7,8] on coinductive logic programming, which addresses many of the same issues in the context of logic programming.

Ordinary recursion over inductive datatypes corresponds to the case in which C is well-founded. In this case, the solution h exists and is unique: it is the least solution in the standard flat Scott domain. For example, the factorial function is uniquely defined by (3) in this sense. If C is not well-founded, there can be multiple solutions, and the one provided by the standard semantics of recursion is typically not be the one we want. Nevertheless, the diagram (2) can still serve as a valid definitional scheme, provided we are allowed to specify a desired solution. In the free variables example, the codomain of the function (sets of variables) is indeed a complete CPO under the usual set inclusion order, and the constructor α is continuous, thus the desired solution can be obtained by a least fixpoint computation.

The example (1) involving free variables of a λ-coterm fits this scheme with the diagram

$$\begin{tikzcd}
\texttt{Term} \arrow[r, "\texttt{fv}"] \arrow[d, "\gamma"'] & \mathcal{P}(\texttt{Var}) \\
F(\texttt{Term}) \arrow[r, "\mathsf{id}_{\texttt{Var}} + \texttt{fv}^2 + \mathsf{id}_{\texttt{Var}} \times \texttt{fv}"'] & F(\mathcal{P}(\texttt{Var})) \arrow[u, "\alpha"']
\end{tikzcd}$$

where $FX = \texttt{Var} + X^2 + \texttt{Var} \times X$ and

$$\begin{aligned}
\gamma(\texttt{Var}\ x) &= \iota_0(x) & \alpha(\iota_0(x)) &= \{x\}\\
\gamma(\texttt{App}\ (t_1,t_2)) &= \iota_1(t_1,t_2) & \alpha(\iota_1(u,v)) &= u \cup v\\
\gamma(\texttt{Lam}\ (x,t)) &= \iota_2(x,t) & \alpha(\iota_2(x,v)) &= v \setminus \{x\}.
\end{aligned}$$

Here the domain of $\texttt{fv}$ (regular λ-coterms) is not well-founded and the codomain (sets of variables) is not a final coalgebra, but the codomain is a complete CPO under the usual set inclusion order with bottom element $\varnothing$, and the desired solution is the least solution in this order; it is just not the one that would be computed by the standard semantics of recursive functions.

Unfortunately, current programming languages provide little support for specifying alternative solutions. One must be able to specify a canonical method for solving systems of equations over an F-algebra (the codomain) obtained from the function definition and the input. We will demonstrate through several examples that such a feature would be extremely useful in a programming language and would bring coinduction and coinductive datatypes to a new level of usability in accordance with the elegance already present for algebraic datatypes. Our examples include free variables, α-conversion, and substitution in infinitary terms; halting probabilities, expected running times, and outcome functions of probabilistic protocols; and abstract interpretation. In each case, the function would diverge under the standard semantics of recursion.

In this paper we propose programming language constructs that would allow the specification of alternative solutions and methods to compute them. These examples require different solution methods: iterative least fixpoint computation, Gaussian elimination, structural coinduction. We describe how this feature might be implemented in a functional language and give mock-up implementations of all our examples. In our implementation, we show how the function definition specifies a system of equations and indicate how that system of equations might be extracted automatically and then passed to an equation solver. In many cases, we suspect that the process can be largely automated, requiring little extra work on the part of the programmer.

Current functional languages are not particularly well suited to the manipulation of coinductive datatypes. For example, in OCaml one can form coinductive objects with **let rec** as in (1), but due to the absence of mutable variables, such objects can only be created and not dynamically manipulated, which severely limits their usefulness. One can simulate them with references, but this negates the elegance of algebraic manipulation of inductively defined datatypes, for which the ML family of languages is so well known. It would be of benefit to be able to treat coinductive types the same way.

Our mock-up implementation with all examples and solvers is available from [5].

2 Motivating Examples

In this section we present a number of motivating examples that illustrate the usefulness of the problem. Several examples of well-founded definitions that fit the scheme (2) can be found in the cited literature, including the Fibonacci function and various divide-and-conquer algorithms such as quicksort and mergesort, so we focus on non-well-founded examples: free variables and substitution in λ-coterms, probabilistic protocols, and abstract interpretation.

2.1 Substitution

We now describe another function on infinitary λ-terms: substitution. A typical implementation for well-founded terms would be

```
let rec subst t y = function
  | Var x -> if x = y then t else Var x
  | App (t1,t2) -> App (subst t y t1, subst t y t2)
  | Lam (x,s) -> if x = y then Lam (x,s)
                 else if x ∈ fv t then
                   let w = fresh ()
                   in Lam (w, subst t y (rename w x s))
                 else Lam (x, subst t y s)
```

where fv is the free variable function defined above and rename w x s is a function that substitutes a fresh variable w for x in a term s.

```
let rec rename w x = function
  | Var z -> Var (if z = x then w else z)
  | App (t1,t2) -> App (rename w x t1, rename w x t2)
  | Lam (z,s) -> if z = x then Lam (z,s)
                 else Lam (z, rename w x s)
```

Applied to a λ-coterm with a cycle, for example attempting to substitute a term for y in (1), the computation would never finish. Nevertheless, this computation fits the scheme (2) with $C = A = \texttt{term}$ (the set of λ-coterms), functor

$$FX = \texttt{term} + X^2 + \texttt{string} \times X \qquad Fh = \mathsf{id}_{\texttt{term}} + h^2 + \mathsf{id}_{\texttt{string}} \times h$$

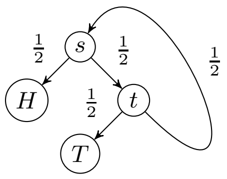
and γ and α defined by

$$\begin{aligned}
\gamma(\texttt{Var}\ x) &= \begin{cases} \iota_0(t) & \text{if } x = y \\ \iota_0(\texttt{Var}\ x) & \text{otherwise} \end{cases} \\
\gamma(\texttt{App}\ (t_1, t_2)) &= \iota_1(t_1, t_2) \\
\gamma(\texttt{Lam}\ (x, s)) &= \begin{cases} \iota_0(\texttt{Lam}\ (x, s)) & \text{if } x = y \\ \iota_2(w, \texttt{rename}\ w\ x\ s) & \text{if } x \neq y \text{ and } x \in \texttt{fv}\ t, \text{ where } w \text{ is fresh} \\ \iota_2(x, s) & \text{otherwise} \end{cases} \\
\alpha(\iota_0(s)) &= s \\
\alpha(\iota_1(s_1, s_2)) &= \texttt{App}\ (s_1, s_2) \\
\alpha(\iota_2(x, s)) &= \texttt{Lam}\ (x, s)
\end{aligned}$$

In this case, even though the domain is not well-founded, the solution nevertheless exists and is unique up to observational equivalence. This is because the definition of the function is corecursive and takes values in a final coalgebra.

2.2 Probabilistic Protocols

In this section, we present a few examples in the realm of probabilistic protocols. Imagine one wants to simulate a biased coin, say a coin with probability 2/3 of heads, with a fair coin. Here is a possible solution: flip the fair coin. If it comes up heads, output heads, otherwise flip again. If the second flip is tails, output tails, otherwise repeat from the start. This protocol can be represented succinctly by the following probabilistic automaton:



(4)

Operationally, starting from states s and t, the protocol generates series that converge to 2/3 and 1/3, respectively.

$$\mathsf{Pr}_H(s) = \tfrac{1}{2} + \tfrac{1}{8} + \tfrac{1}{32} + \tfrac{1}{128} + \cdots = \tfrac{2}{3}$$
$$\mathsf{Pr}_H(t) = \tfrac{1}{4} + \tfrac{1}{16} + \tfrac{1}{64} + \tfrac{1}{256} + \cdots = \tfrac{1}{3}.$$

However, these values can also be seen to satisfy a pair of mutually recursive equations:

$$\mathsf{Pr}_H(s) = \tfrac{1}{2} + \tfrac{1}{2} \cdot \mathsf{Pr}_H(t) \qquad\qquad \mathsf{Pr}_H(t) = \tfrac{1}{2} \cdot \mathsf{Pr}_H(s).$$

This gives rise to a contractive map on the unit interval, which has a unique solution. It is also monotone and continuous with respect to the natural order on the unit interval, therefore has a unique least solution.

One would like to define the probabilistic automaton (4) by

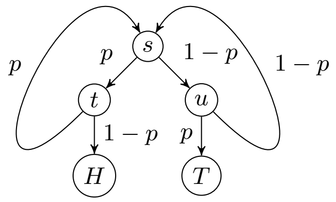
```
type pa = H | T | Flip of float * pa * pa
let rec s = Flip (0.5,H,t) and t = Flip (0.5,T,s)
```

and write a recursive program, say something like

```
let rec pr_heads = function
  | H -> 1.
  | T -> 0.
  | Flip (p,u,v) -> p *. (pr_heads u) +. (1 -. p) *. (pr_heads v)
```

and specify that the extracted equations should be solved exactly by Gaussian elimination, or by iteration until achieving a fixpoint to within a sufficiently small error tolerance ε. We give implementations using both methods.

The *von Neumann trick* for simulating a fair coin with a coin of arbitrary bias is a similar example. In this protocol, we flip the coin twice. If the outcome is HT, we output heads. If the outcome is TH, we output tails. These outcomes occur with equal probability. If the outcome is HH or TT, we repeat.



Here we would define

```
let rec s = Flip (p,t,u) and t = Flip (p,s,H) and u = Flip (p,T,s)
```

but the typing and recursive function `pr_heads` are the same. Markov chains and Markov decision processes can be modeled the same way.

Other functions on probabilistic automata can be computed as well. The expected number of steps starting from state s is the least solution of the equation

$$E(s) = \begin{cases} 0 & \text{if } s \in \{\mathtt{H}, \mathtt{T}\} \\ 1 + p \cdot E(u) + (1-p) \cdot E(v) & \text{if } s = \mathtt{Flip}(p, u, v). \end{cases}$$

We would like to write simply

```
let rec ex = function
  | H -> 0.
  | T -> 0.
  | Flip (p,u,v) -> 1. +. p *. (ex u) +. (1 -. p) *. (ex v)
```

and specify that the extracted equations should be solved by Gaussian elimination or least fixpoint iteration from 0.

The coinflip protocols we have discussed all fit the abstract definitional scheme (2) in the form

$$\begin{tikzcd} S \arrow[r, "h"] \arrow[d, "\gamma"'] & \mathbb{R} \\ FS \arrow[r, "Fh"'] & F\mathbb{R} \arrow[u, "\alpha"'] \end{tikzcd}$$

where S is the set of states (a state can be either $\mathtt{H}$, $\mathtt{T}$, or a triple (p, u, v), where $p \in \mathbb{R}$ and $u, v \in S$, the last indicating that it flips a p-biased coin and moves to state u with probability p and v with probability $1 - p$), and F is the functor

$$FX = \mathbb{1} + \mathbb{1} + \mathbb{R} \times X^2 \qquad\qquad Fh = \mathsf{id}_{\mathbb{1}} + \mathsf{id}_{\mathbb{1}} + \mathsf{id}_{\mathbb{R}} \times h^2.$$

For both the probability of heads and expected running times examples, we can take

$$\gamma(s) = \begin{cases} \iota_0() & \text{if } s = \mathtt{H} \\ \iota_1() & \text{if } s = \mathtt{T} \\ \iota_2(p, u, v) & \text{if } s = (p, u, v). \end{cases}$$

For the probability of heads, we can take

$$\alpha(\iota_0()) = 1 \qquad \alpha(\iota_1()) = 0 \qquad \alpha(\iota_2(p, a, b)) = pa + (1-p)b.$$

For the expected running time, we can take

$$\alpha(\iota_0()) = \alpha(\iota_1()) = 0 \qquad \alpha(\iota_2(p, a, b)) = 1 + pa + (1-p)b.$$

The desired solution in all cases is a least fixpoint in an appropriate ordered domain.

2.3 Abstract Interpretation

In this section we present our most involved example: abstract interpretation of a simple imperative language. Our example follows Cousot and Cousot [6] as inspired by lecture notes of Stephen Chong [4].

Consider a simple imperative language of while programs with integer expressions a and commands c. Let Var be a countable set of variables.

$$\begin{aligned} a &::= n \in \mathbb{Z} \mid x \in \mathsf{Var} \mid a_1 + a_2 \\ c &::= \mathsf{skip} \mid x := a \mid c_1 \,;\, c_2 \mid \mathsf{if}\ a\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid \mathsf{while}\ a\ \mathsf{do}\ c \end{aligned}$$

For the purpose of tests in the conditional and while loop, an integer is considered true if and only if it is nonzero. Otherwise, the operational semantics is standard, in the style of [11]. A store is a partial function from variables to integers, an arithmetic expression is interpreted relative to a store and returns an integer, and a command is interpreted relative to a store and returns an updated store.

Abstract interpretation defines an abstract domain that approximates the values manipulated by the program. We define an abstract domain for integers that abstracts an integer by its sign. The set of abstract values is $\mathsf{AbsInt} = \{\mathsf{neg}, \mathsf{zero}, \mathsf{pos}, \top\}$, where neg, zero, and pos represent negative, zero, and positive integers, repectively, and $\top$ represents an integer of unknown sign. The abstract values form a join semilattice with join $\sqcup$ defined by the following diagram:

$$\begin{tikzcd} & \top \arrow[dl, no head] \arrow[d, no head] \arrow[dr, no head] & \\ \mathsf{neg} & \mathsf{zero} & \mathsf{pos} \end{tikzcd} \tag{5}$$

The abstract interpretation of an arithmetic expression is defined relative to an abstract store $\sigma : \mathsf{Var} \rightharpoonup \mathsf{AbsInt}$, used to interpret the abstract values of variables. We write $\mathsf{AS} = \mathsf{Var} \rightharpoonup \mathsf{AbsInt}$ for the set of abstract stores. The abstract interpretation of arithmetic expressions is given by:

$$\begin{aligned} \mathcal{A}[\![n]\!]\sigma &= \begin{cases} \mathsf{pos} & \text{if } n > 0 \\ \mathsf{zero} & \text{if } n = 0 \\ \mathsf{neg} & \text{if } n < 0 \end{cases} \\ \mathcal{A}[\![x]\!]\sigma &= \sigma(x) \\ \mathcal{A}[\![a_1 + a_2]\!] &= \begin{cases} \mathcal{A}[\![a_1]\!]\sigma & \text{if } \mathcal{A}[\![a_2]\!]\sigma = \mathsf{zero} \\ \mathcal{A}[\![a_2]\!]\sigma & \text{if } \mathcal{A}[\![a_1]\!]\sigma = \mathsf{zero} \\ \mathcal{A}[\![a_1]\!]\sigma \sqcup \mathcal{A}[\![a_2]\!]\sigma & \text{otherwise.} \end{cases} \end{aligned}$$

The abstract interpretation of commands returns an abstract store, which is an abstraction of the concrete store returned by the commands. Abstract stores form a join semilattice, where the join $\sqcup$ of two abstract stores just takes the join of each variable: $(\sigma_1 \sqcup \sigma_2)(x) = \sigma_1(x) \sqcup \sigma_2(x)$. Commands other than the while loop are interpreted as follows:

$$\mathcal{C}[\![\mathsf{skip}]\!]\sigma = \sigma \qquad \mathcal{C}[\![x := a]\!]\sigma = \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma] \qquad \mathcal{C}[\![c_1 \,;\, c_2]\!]\sigma = \mathcal{C}[\![c_2]\!](\mathcal{C}[\![c_1]\!]\sigma)$$

$$\mathcal{C}[\![\mathsf{if}\ a\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!]\sigma = \begin{cases} \mathcal{C}[\![c_1]\!]\sigma & \text{if } \mathcal{A}[\![a]\!]\sigma \in \{\mathsf{pos}, \mathsf{neg}\} \\ \mathcal{C}[\![c_2]\!]\sigma & \text{if } \mathcal{A}[\![a]\!]\sigma = \mathsf{zero} \\ \mathcal{C}[\![c_1]\!]\sigma \sqcup \mathcal{C}[\![c_2]\!]\sigma & \text{otherwise.} \end{cases}$$

We would ideally like to define

$$\mathcal{C}[\![\mathsf{while}\ a\ \mathsf{do}\ c]\!]\sigma = \begin{cases} \sigma & \text{if } \mathcal{A}[\![a]\!]\sigma = \mathsf{zero} \\ \sigma \sqcup \mathcal{C}[\![\mathsf{while}\ a\ \mathsf{do}\ c]\!](\mathcal{C}[\![c]\!]\sigma) & \text{otherwise.} \end{cases}$$

Unfortunately, when $\mathcal{A}[\![a]\!]\sigma \neq \mathsf{zero}$, the definition is not well-founded, because it is possible for σ and $\mathcal{C}[\![c]\!]\sigma$ to be equal. However, it is a correct definition of $\mathcal{C}[\![\mathsf{while}\ a\ \mathsf{do}\ c]\!]$ as a least fixpoint in the join semilattice of abstract stores. The existence of the least fixpoint can be obtained in a finite time by iteration because the join semilattice of abstract stores satisfies the ascending chain condition (ACC), that is, it does not contain any infinite ascending chains.

Given $\mathcal{A}[\![a]\!]$ and $\mathcal{C}[\![c]\!]$ previously defined, $\mathcal{C}[\![\mathsf{while}\ a\ \mathsf{do}\ c]\!]$ satisfies the following instantiation of (2):

$$\begin{tikzcd} \mathsf{AS} \arrow[r, "\mathcal{C}[\![\mathsf{while}\ a\ \mathsf{do}\ c]\!]"] \arrow[d, "\gamma"'] & \mathsf{AS} \\ \mathsf{AS} + \mathsf{AS} \times \mathsf{AS} \arrow[r, "\mathsf{id}_{\mathsf{AS}} + \mathsf{id}_{\mathsf{AS}} \times \mathcal{C}[\![\mathsf{while}\ a\ \mathsf{do}\ c]\!]"'] & \mathsf{AS} + \mathsf{AS} \times \mathsf{AS} \arrow[u, "\alpha"'] \end{tikzcd}$$

where the functor is $FX = \mathsf{AS} + \mathsf{AS} \times X$ and

$$\gamma(\sigma) = \begin{cases} \iota_1(\sigma) & \text{if } \mathcal{A}[\![a]\!]\sigma = \mathsf{zero} \\ \iota_2(\sigma, \mathcal{C}[\![c]\!]\sigma) & \text{otherwise} \end{cases} \qquad \begin{aligned} \alpha(\iota_1(\sigma)) &= \sigma \\ \alpha(\iota_2(\sigma, \tau)) &= \sigma \sqcup \tau \end{aligned}$$

The function $\mathcal{C}[\![\mathsf{while}\ a\ \mathsf{do}\ c]\!]$ is the least function in the pointwise order that makes the above diagram commute.

This technique allows us to define $\mathcal{C}[\![c]\!]$ inductively on the structure of c. An inductive definition can be used here because the set of abstract syntax trees is well-founded.

The literature on abstract interpretation explains how to compute the least fixpoint, and much research has been done on techniques for accelerating convergence to the least fixpoint. This body of research can inform compiler optimization techniques for computation with coalgebraic types.

2.4 Finite Automata

We conclude this section with a brief example involving finite automata. Suppose we want to construct a deterministic finite automaton (DFA) over a two-letter alphabet accepting the intersection of two regular sets given by two other DFAs over the same alphabet. We might define states coalgebraically by

```
type state = State of bool * state * state
```

where the first component specifies whether the state is an accepting state and the last two components give the states to move to under the two input symbols. The standard product construction is defined coalgebraically simply by

```
let rec product (s : state) (t : state) : state =
  match s, t with
    | State (b1,s1,t1), State (b2,s2,t2) ->
        State (b1 && b2, product s1 t1, product s2 t2)
```

and we can compute it, provided we can solve the generated equations.

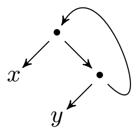
3 A Framework for Non-Well-Founded Computation

In this section we discuss our proposed framework for incorporating language constructs to support non-well-founded computation. At a high level, we wish to specify a function h uniquely using a finite set E of structural recursive equations. The function is defined in much the same way as an ordinary recursive function on an inductive datatype. However, the value $h(x)$ of the function on a particular input x is computed not by calling the function in the usual sense, but by generating a system of equations from the function definition and then passing the equations to a specified equation solver to find a solution. The equation solver is either a standard library function or programmed by the user according to an explicit interface.

The process is partitioned into several tasks as follows.

1. The left-hand sides of the clauses in the function definition determine syntactic terms representing equation schemes. These schemes are extracted by the compiler from the abstract syntax tree of the left-hand side expressions. This determines (more or less, subject to optimizations) the function γ in the diagram (2).
2. The right-hand sides of the clauses in the function definition determine the function α in the diagram (2) (again, more or less, subject to optimizations). These expressions essentially tell how to evaluate terms extracted in step 1 in the codomain. As in 1, these are determined by the compiler from the abstract syntax trees of the right-hand sides.
3. At runtime, when the function is called with a coalgebraic element c, a finite system of equations is generated from the schemes extracted in steps 1 and 2, one equation for each element of the coalgebra reachable from c. In fact, we can take the elements reachable from c as the variables in our equations. Each such element matches exactly one clause of the function body, and this determines the right-hand side of the equation that is generated.
4. The equations are passed to a solver that is specified by the user. This will presumably be a module that is programmed separately according to a fixed interface and available as a library function. There should be a simple syntactic mechanism for specifying an alternative solution method (although we do not specify here what that should look like).

Let us illustrate this using our initial example of the free variables. Recall the infinitary λ-term below and the definition of the free variables function from the introduction:



```
let rec fv = function
  | Var v -> {v}
  | App (t1,t2) -> (fv t1) ∪ (fv t2)                    (6)
  | Lam (x,t) -> (fv t) − {x}
```

Steps 1 and 2 would analyze the left-and right-hand sides of the three clauses in the body at compile time to determine the equation schemes. Then at runtime, if the function were called on the coalgebraic element pictured, the runtime system would generate four equations, one for each node reachable from the top node:

```
fv t = (fv x) ∪ (fv u)    fv u = (fv y) ∪ (fv t)    fv x = {x}    fv y = {y}
```

where `t` and `u` are the unlabeled top and right nodes of the term above.

As noted, these equations have many solutions. In fact, any set containing the variables `x` and `y` will be a solution. However, we are interested in the least solution in the ordered domain $(\mathcal{P}(\mathsf{Var}), \subseteq)$ with bottom element $\varnothing$. In this case, the least solution would assign {`x`} to the leftmost node, {`y`} to the lowest node, and {`x,y`} to the other two nodes.

With this in mind, we would pass the generated equations to an iterative equation solver, which would produce the desired solution. In many cases, such as this example, the codomain is a complete partial order and we have default solvers to compute least fixpoints, leaving to the programmer the simple task of indicating that this is the desired solution method. That would be an ideal situation: the defining equations of (6) plus a simple tag would be enough to obtain the desired solution.

3.1 Generating Equations

The equations are generated from the recursive function definition and the input c, a coalgebraic element, in accordance with the abstract definitional scheme (2). The variables can be taken to be the elements of the coalgebraic object reachable from c. There are finitely many of these, as no infinite object can ever exist in a running program. More accurately stated, the objects of the final coalgebra represented by coalgebraic elements during program execution are all *regular* in the sense that they have a finite representation. These elements are first collected into a data structure (in our implementation, simply a list) and the right-hand sides of the equations are determined by the structure of the object using pattern matching. The object matches exactly one of the terms extracted in step 1.

4 Implementation

The examples of §2 show the need for new program constructs that would allow the user to manipulate corecursive types with the same ease and elegance as we are used to for algebraic datatypes. It is the goal of this section to provide language constructs that allow us to provide the intended semantics to the examples above in a functional language like OCaml.

The general idea behind the implementation is as follows. We want to keep the overhead for the programmer to a minimum. We want the programmer to specify the function in the usual way, then at runtime, when the function is evaluated on a given argument, a set of equations is generated and passed on to a solver, which will find a solution according to the specification. In an ideal situation, the programmer only has to specify the solver. For the examples where a CPO structure is present in the codomain, such as the free variables example, or when we have a complete metric space and a contractive map, we provide the typical solution methods (least and unique fixpoint) and the programmer only needs to tag the codomain with the intended solver. In other cases, the programmer needs to implement the solver.

4.1 Equations and Solvers

Our mock-up implementation aims to allow the programmer to encode a particular instantiation of the general diagram (2) as an OCaml module. This module can then be passed to an OCaml functor, `Corecursive`, that builds the desired function. We discuss the structure of `Corecursive` later in this section.

The functor F is represented by a parameterized type `'b f`. The structures (C, γ) and (A, α), which form a coalgebra and an algebra, respectively, for the functor F, are defined by types `coalgebra` and `algebra`, respectively. This allows us to specify γ naturally as a function from `coalgebra` to `coalgebra f` and α as a function from `algebra f` to `algebra`. In the free variables example, if `VarSet` is a module implementing sets of strings, this is done as:

```
type 'b f = I1 of string | I2 of 'b * 'b | I3 of string * 'b
type coalgebra = Var of string
               | App of coalgebra * coalgebra
               | Lam of string * coalgebra
type algebra = VarSet.t

let gamma (c:coalgebra) : coalgebra f =
  match c with
    | Var v -> I1 v
    | App(c1, c2) -> I2(c1, c2)
    | Lam(x, c) -> I3(x, c)
let alpha (s:algebra f) : algebra =
  match s with
    | I1 v -> VarSet.singleton v
    | I2(s1, s2) -> VarSet.union s1 s2
    | I3(x, s) -> VarSet.remove x s
```

Variables are represented by strings and fresh variables are generated with a counter. Equations are of the form `variable = t`, where the variables on the left-hand side are elements of the domain and the terms on the right side are built up from the constructors of the datatype, constants and variables.

In the `fv` example, the domain was specified by the following datatype:

```
type term =
  | Var of string
  | App of term * term
  | Lam of string * term
```

Recall the four equations above defining the free variables of the λ-term (1) from the introduction:

```
fv t = (fv x) ∪ (fv u)   fv u = (fv y) ∪ (fv t)   fv x = {x}   fv y = {y}
```

A variable name is generated for each element of the coalgebra encountered. For example, here we write `v1` for the unknown corresponding to the value of `fv t`; `v2` for `x`; `v3` for `u`; and `v4` for `y`. An equation is represented as a pair of a variable and an element of type `f variable`. The intuitive meaning of a pair $(\mathtt{v}, \mathtt{w})$ is the equation $\mathtt{v} = \alpha(\mathtt{w})$. In the example above, we would have

`("v1", I2("v2", "v3"))`	representing	`v1 = v2 ∪ v3`
`("v2", I1("x"))`	representing	`v2 = {x}`
`("v3", I2("v4", "v1"))`	representing	`v3 = v4 ∪ v1`
`("v4", I1("y"))`	representing	`v4 = {y}`

The function `solve` can now be described. Its arguments are a variable `v` for which we want a solution and a system of equations in which `v` appears. It returns a value for `v` that satisfies the equations. In most cases the solution is not unique, and the `solve` method determines which solution is returned.

For technical reasons, two more functions need to be provided. The function `equal` provides an equality test on the coalgebra, which allows the equation generator to know when it has encountered a loop. In most cases, this equality is just the OCaml physical equality `==`; this is necessary because the OCaml equality `=` on coinductive objects does not terminate. In some other cases the function `equal` is an equality function built from both `=` and `==`.

The function `fh` can be seen either as an iterator on the functor `f` in the style of folding and mapping on lists or as a monadic operator on the functor `f`. It allows the lifting of a function from `'c` (typically `coalgebra`) to `'a` (typically `algebra`) to a function from `'c f` to `'a f`, while folding on an element of type `'e`. It works by destructing the element of type `'c f` to get some number (perhaps zero) elements of type `'c`, successively applying the function on each of them while passing through the element of type `'e`, and reconstructing an element of type `'a f` with the same constructor used in `'c f`, returned with the final value of the element of type `'e`. In the example on free variables, the function `fh` is defined as:

```
let fh (h: 'c * 'e -> 'a * 'e) : 'c f * 'e -> 'a f * 'e = function
  | I1 v, e -> I1 v, e
  | I2(c1, c2), e -> let a1, e1 = h (c1, e) in
                     let a2, e2 = h (c2, e1) in
                     I2(a1, a2), e2
  | I3(x, c), e -> let a, e1 = h (c, e) in
                   I3(x, a), e1
```

If we had access to an abstract representation of the functor `f`, analyzing it allows to automatically generate the function `fh`. This is what we do in §5.

All this is summarized in the signature of a type `SOLVER`, used to specify one of those functions:

```
module type SOLVER = sig
  type 'b f
  type coalgebra
  type algebra

  val gamma : coalgebra -> coalgebra f
  val alpha : algebra f -> algebra

  type variable = string
  type equation = variable * (variable f)

  val solve : variable -> equation list -> algebra

  val equal : coalgebra -> coalgebra -> bool
  val fh : ('c * 'e -> 'a * 'e) -> 'c f * 'e -> 'a f * 'e
end
```

Let us now define the OCaml functor `Corecursive`. From a specification of a function as a module `S` of type `SOLVER`, it generates the equations to be solved and sends them to `S.solve`. Here is how it generates the equations: starting from an element `c` of the coalgebra, it gathers all the elements of the coalgebra that are reachable from `c`, recursively descending with `gamma` and `fh`, and stopping when reaching an element that is equal—in the sense of the function `equal`—to an element that has already been seen. For each of those elements, it generates an associated fresh variable and an associated equation based on applying `gamma` to that element.

From an element `c`, generating the equations and solving them with `solve` returns an element `a` in the coalgebra, the result of applying the function we defined to `c`.

```
module Corecursive :
  functor (S: SOLVER) -> sig
    val main : S.coalgebra -> S.algebra
  end
```

We will now explain the default solvers we have implemented and which are available for the programmer to use. These solvers cover the examples we have shown before: a least fixpoint solver, a solver that generates coinductive elements and is used for substitution, and a Gaussian elimination solver.

4.2 Least Fixpoints

If the algebra A is a CPO, then every monotone function f on A has a least fixpoint, by the Knaster–Tarski theorem. Moreover, if the CPO satisfies the *ascending chain condition* (ACC), that is, if there does not exist an infinite ascending chain, then this least fixpoint can be computed in finite time by iteration, starting from $\bot_A$. Even if the ACC is not satisfied, an approximate least fixpoint may suffice.

In the free variables example, the codomain $(\mathcal{P}(\mathsf{Var}), \subseteq)$ is a CPO, and its bottom element is $\bot_A = \varnothing$. It satisfies the ACC as long as we restrict ourselves to the total set of variables appearing in the term. This set is finite because the term is regular and thus has a finite representation.

To implement this, first consider the set of equations: each variable is defined by one equation relating it to the other variables. We keep a guess for each variable, initially set at $\bot_A$, and compute a next guess based on the equation for each variable. This eventually converges and we can return the value of the desired variable. Note that to implement this, the programmer needs to know that A is a CPO satisfying the ACC, and needs to provide two things: a bottom element $\bot_A$, and an equality relation on A that determines when a fixpoint is achieved.

The same technique can be used to implement the solver for the abstract interpretation example, as it is also a least fixpoint in a CPO. This CPO is the subset of the join semilattice of abstract domains containing only the elements greater than or equal to the initial abstract domain. The ACC is ensured by the fact that the abstract domain is always of finite height. The bottom element is the initial abstract domain. Much of the code is shared with the free variables example. As pointed out before, only the bottom element of A and the equality on A change.

More suprisingly, this technique can also be used in the probability examples. Here the system of equations looks more like a linear system of equation on $\mathbb{R}$. Except in trivial extreme cases, the equations are contracting, thus we can solve them by iterative approximation until getting close enough to a fixpoint. The initial element $\bot_A$ is 0. The equality test on A is the interesting part: since it determines when to stop iterating, two elements of A are considered equal if and only if they differ by less than ε, the precision of the approximation. This is specified by the programmer in the definition of equality on A. Of course, such a linear system could also be solved with Gaussian elimination, as presented below in §4.4.

It can be seen from these examples that the least fixpoint solver is quite generic and works for a large class of problems. We need only parameterize with a bottom element to use as an initial guess and an equality test.

4.3 Generating Coinductive Elements and Substitution

Let us return to the substitution example. Suppose we wanted to replace y in Fig. 1(b) by the term of Fig. 1(a) to obtain Fig. 1(c). The extracted equations would be

```
v1 = App(v2, v3)
v2 = Var("x")
v3 = App(v4, v1)
```

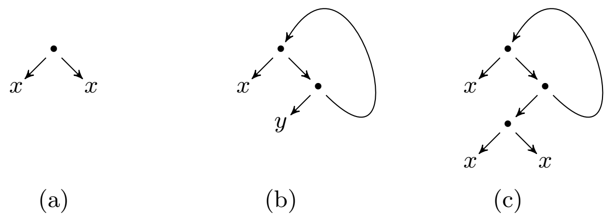


Fig. 1. A substitution example

```
v4 = App(Var "x", Var "x")
```

and we are interested in the value of `v1`. Finding such a `v1` is easily done by executing the following code in OCaml:

```
let rec v1 = App(v2, v3)
    and v2 = Var("x")
    and v3 = App(v4, v1)
    and v4 = App(Var "x", Var "x")
in v1
```

This code can be easily generated (as a string of text) from the equations. Unfortunately, there is no direct way of generating the element that this code would produce. One workaround is to use the module `Toploop` of OCaml that provides the ability to dynamically execute code from a string, like `eval` in Javascript. But that is not a satisfying solution.

Another solution is to allow the program to manipulate terms by making all subterms mutable using references:

```
type term =
  | Var of string
  | App of term ref * term ref
  | Lam of string * term ref
```

This type allows the creation of the desired term by going down the equations and building the terms progressively, backpatching if necessary when encountering a loop. But this is also unsatisfactory, as we had to change the type of `term` to allow references.

The missing piece is mutable variables, which are currently not supported in the ML family of languages. A variable is mutable if it can be dynamically rebound, as with the Scheme `set!` feature or ordinary assignment in imperative languages. In ML, variables are only bound once when they are declared and cannot be rebound.

References can simulate mutable variables, but this corrupts the typing and forces the programmer to work at a lower pointer-based level. Moreover, there are subtle differences in the aliasing behavior of references and mutable variables. The language constructs we propose should ideally be created in a programming language with mutable variables.

4.4 Gaussian Elimination

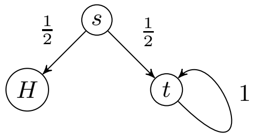
In many of the examples on probabilities and streams, a set of linear equations is generated. One of the examples on probabilistic protocols of §2.2 requires us to find a float `var1` such that

```
var1 = 0.5 + 0.5 * var2
var2 = 0.5 * var1
```

In the case where the equations are contractive, we have already seen that the solution is unique and we can approximate it by iteration. We have also implemented a Gaussian elimination solver that can be used to get a more precise answer or when the map is not contractive but the solution is still unique.

But what happens when the linear system has no solution or an infinite number of solutions? If the system does not have a solution, then there is no fixpoint for the function, and the function is undefined on that input. If there are an infinite number of solutions, it depends on the application. For example, in the case of computing the probability of heads in a probabilistic protocol, we want the least such solution such that all variables take values between 0 and 1.

For example, let us consider the following probabilistic protocol: Flip a fair coin. If it comes up heads, output heads, otherwise flip again. Ignore the result and come back to this last state, effectively flipping again forever. This protocol can be represented by the following probabilistic automaton:



The probability of heads starting from s and t, respectively, is given by:

$$\mathsf{Pr}_H(s) = \tfrac{1}{2} + \tfrac{1}{2} \cdot \mathsf{Pr}_H(t) \qquad\qquad \mathsf{Pr}_H(t) = 1 \cdot \mathsf{Pr}_H(t).$$

The set of solutions for these equations for $\mathsf{Pr}_H(t)$ is the interval $[0, 1]$, thus the set of solutions for $\mathsf{Pr}_H(s)$ is the interval $[\frac{1}{2}, 1]$. The desired result, however, is the least of those solutions, namely 1/2 for $\mathsf{Pr}_H(s)$, because the protocol halts with result heads only with probability 1/2.

Again, the Gaussian solver is quite generic and would be applicable to a large class of problems involving linear equations.

5 Future Work: Automatic Partitioning

In §4, we described a mock-up implementation that demonstrates the feasibility of our approach. In this implementation, the programmer needs to provide the elements of the `SOLVER` module. We now describe our ideas for future work, and in particular, ideas to make the task of the programmer easier by automatically generating some of those elements.

Providing all the elements to a `SOLVER` module requires from the programmer a good understanding of the concepts explained in this paper and a method to solve equations. On the other hand, examples show that the same solving techniques arise again and again. Ideally, we would like the programmer to have to write only:

```
type term =
  | Var of string
  | App of term * term
  | Lam of string * term
```

```
let rec[...] fv = function
  | Var v -> {v}
  | App (t1,t2) -> (fv t1) ∪ (fv t2)
  | Lam (x,t) -> (fv t) − {x}
```

where the keyword `rec` has been parameterized by the name of a module implementing the `SOLVER` interface for a particular codomain, such as a generic iteration solver for CPOs or contractive maps or a Gaussian elimination solver for linear equations.

This definition is almost enough to generate the `SOLVER` module. Only three more things need to be specified by the programmer:

– the function `equal` on coalgebras, which is just `==` in most cases; and
– the two elements needed in the least fixpoint algorithm: a bottom element $\bot_A$ and an equality test $=_A$ on the algebra A, written `algebra` in the code.

The other elements can be directly computed from a careful analysis of the function definition:

– The function can be typed with the usual typing rules for recursive functions. Then `algebra` is defined as its input type and `coalgebra` as its output type.
– An analysis of the abstract syntax trees of the clauses of the function definition can determine what is executed before the recursive calls, which comprises γ, and what is executed after the recursive calls, which comprises α. An analysis of the arguments that are passed to the recursive calls, as well as the variables that are still alive across the boundary between `gamma` and `alpha`, determine the functor `f`.
– The function `fh` can be defined by induction on the structure of the abstract syntax tree defining `'a f`. The only difficult case is the product, where we apply `h` to every element of type `'a` in the product, passing through the element of type `'e`, and returning a reconstructed product of the results.
– The type `equation` is always defined in the same way.
– Finally, the `solve` function is generic for all functions solved as a least fixpoint by iteration, just depending on the bottom element and the equality on the algebra.

6 Conclusion

Coalgebraic (coinductive) datatypes and algebraic (inductive) datatypes are similar in many ways. Nevertheless, there are some important distinctions. Algebraic types have a long history, are very well known, and are heavily used in modern applications, especially in the ML family of languages. Coalgebraic types, on the other hand, are the subject of more recent research and are less well known. Not all modern languages support coalgebraic types—for example, Standard ML and F# do not—and even those that do may not do so adequately.

The most important distinction is that coalgebraic objects can be cyclic, whereas algebraic objects are always well-founded. Functions defined by structural recursion on well-founded data always terminate and yield a value under the standard semantics of recursion, but not so on coalgebraic data. A more subtle distinction is that constructors can be interpreted as functions under the algebraic interpretation, as they are in Standard ML, but not under the coalgebraic interpretation as in OCaml.

Despite these differences, there are some strong similarities. They are defined in the same way by recursive type equations, algebraic types as initial solutions and coalgebraic types as final solutions. Because of this similarity, we would like to program with them in the same way, using constructors and destructors and writing recursive definitions using pattern matching.

In this paper we have shown through several examples that this approach to computing with coalgebraic types is not only useful but viable. For this to be possible, it is necessary to circumvent the standard semantics of recursion, and we have demonstrated that this obstacle is not insurmountable. We have proposed new programming language features that would allow the specification of alternative solutions and methods to compute them, and we have given mock-up implementations that demonstrate that this approach is feasible.

The chief features of our approach are the interpretation of a recursive function definition as a scheme for the specification of equations, a means for extracting a finite such system from the function definition and its (cyclic) argument, a means for specifying an equation solver, and an interface between the two. In many cases, such as an iterative fixpoint on a codomain satisfying the ascending chain condition, the process can be largely automated, requiring little extra work on the part of the programmer.

We have mentioned that mutable variables are essential for manipulating coalgebraic data. Current functional languages in the ML family do not support mutable variables; thus true coalgebraic data can only be constructed explicitly using **let rec**, not programmatically. Moreover, once constructed, a coalgebraic object cannot be changed dynamically. These restrictions currently constitute a severe restriction the use of coalgebraic datatypes. One workaround is to simulate mutable variables with references, but this is a grossly unsatisfactory alternative, because it confounds algebraic elegance and forces the programmer to work at a lower pointer-based level. A future endeavor is to provide a smoother and more realistic implementation of these ideas in an ML-like language with mutable variables.

Acknowledgments. We are grateful to Bob Constable, Edgar Friendly, Nate Foster, Helle Hvid Hansen, Bart Jacobs, Jonathan Kimmitt, Xavier Leroy, Andrew Myers, Stefan Milius, Ross Tate, and the anonymous referees for helpful comments. Part of this work was done while the first two authors were visiting Radboud University Nijmegen and the CWI Amsterdam.

References

1. Adámek, J., Lücke, D., Milius, S.: Recursive coalgebras of finitary functors. Theoretical Informatics and Applications 41, 447–462 (2007)
2. Adámek, J., Milius, S., Velebil, J.: Elgot algebras. Log. Methods Comput. Sci. 2(5:4), 1–31 (2006)
3. Capretta, V., Uustalu, T., Vene, V.: Corecursive Algebras: A Study of General Structured Corecursion. In: Oliveira, M.V.M., Woodcock, J. (eds.) SBMF 2009. LNCS, vol. 5902, pp. 84–100. Springer, Heidelberg (2009)
4. Chong, S.: Lecture notes on abstract interpretation. Harvard University (2010), `http://www.seas.harvard.edu/courses/cs152/2010sp/lectures/lec20.pdf`
5. CoCaml project (December 2012), `http://www.cs.cornell.edu/Projects/CoCaml/`
6. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: 4th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages, pp. 238–252. ACM Press, New York (1977)
7. Simon, L., Mallya, A., Bansal, A., Gupta, G.: Coinductive Logic Programming. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 330–345. Springer, Heidelberg (2006)
8. Simon, L., Bansal, A., Mallya, A., Gupta, G.: Co-Logic Programming: Extending Logic Programming with Coinduction. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 472–483. Springer, Heidelberg (2007)
9. Taylor, P.: Practical Foundations of Mathematics. Cambridge Studies in Advanced Mathematics, vol. 59. Cambridge University Press (1999)
10. y Widemann, B.T.: Coalgebraic semantics of recursion on circular data structures. In: Cirstea, C., Seisenberger, M., Wilkinson, T. (eds.) CALCO Young Researchers Workshop (CALCO-jnr 2011), pp. 28–42 (August 2011)
11. Winskel, G.: The Formal Semantics of Programming Languages: An Introduction. MIT Press, Cambridge (1993)