A Cofixpoint Combinator for Mixed Recursive-Corecursive Functions

Jannis Limperg

University of Freiburg
Department of Computer Science
Chair of Programming Languages
Adviser: Peter Thiemann

Submitted 10 May 2017

Contents

1 Introduction

Consider the following problem: Define a function, runs, which takes as input an infinite stream of numbers. Its output should be the stream of all strictly decreasing runs of numbers in the input, such that for example

$$\texttt{runs}\ [0,3,2,1,3,3,\ldots] = [[0],[3,2,1],[3],\ldots].$$

In partial functional languages, say Haskell [34, 22], this is easily accomplished via an auxiliary function runs':

```
runs' :: [Nat] -> Stream Nat -> Stream [Nat]
runs' run (x : y : xs)
  = if y < x
      then runs' (run ++ [x]) (y : xs)
      else (run ++ [x]) : runs' [] (y : xs)
```

runs' takes the current run as its first argument. It then checks whether the first two elements of the input stream are part of the same run. If so, it adds the first element to the current run and recurses; otherwise it finishes up the current run, predending it to the output stream, and recurses with a new empty run. Appending single elements to the end of a list (++ [_]) is inefficient, but this does not concern us. Given this definition, runs is simply runs' [ ].

Unfortunately, our definition of runs' is unacceptable in current total functional languages. Since runs' produces an infinite data structure, it must be defined by corecursion, and such definitions must be productive. Informally, this means that the evaluation of any finite prefix of runs' xs must terminate (for all xs) or, in other words, that we only ever have to wait a finite amount of time until the next element of the output stream is available [36].

Languages which strive for totality – usually in order to maintain consistency as a logic under the Curry-Howard-de Bruijn isomorphism [17, 20, 27] – must check productivity statically. And since productivity checking is undecidable, languages rely upon heuristics which reject some productive programmes. The most common of these heuristics is the guardedness condition [16, 24], which essentially requires that corecursive calls in a function body appear only as direct arguments to a constructor of a coinductive data type.

runs' is a victim of the incompleteness of the guardedness condition because in the then branch, the recursive call does not appear under a constructor, and therefore is not guarded. In contrast, the else branch is fine because the recursive call is a direct argument of the : stream constructor.

Yet, despite its inadmissibility, runs' is indeed productive: For any input stream, we can only take the then branch a finite number of times, at most until we reach zero. Then we must take the else branch, which produces an element of the output stream before recursing. Therefore, the next element of a stream starting with the number n is always produced after at most $n + 1$ recursion steps, and thus in finite time.

Hence, the question is: How do we define runs' and functions like it in such a way that the language can recognise them as productive? We call these functions *mixed*

recursive-corecursive because they contain both a recursive and a corecursive component. In the recursive component, self-referential calls are justified by a termination argument; in the case of runs', we count down towards zero while recursing over the stream in the then branch. In the corecursive component, self-referential calls are justified by guardedness like in the else branch. ([12] makes these notions more precise.)

In this paper, we develop a technique that allows mixed recursive-corecursive functions to be defined with relative ease, without altering the productivity checker or type system. Our main contribution is a cofixpoint combinator that forms the greatest fixed point of an open recursive function while allowing that function to recurse in two different ways: By guarded corecursion, as in the else branch of runs', and by well-founded recursion [8, 31, 10].[1] The latter generalises the idea that the then branch can only be taken a finite number of times because the head of the stream passed to runs' always decreases.

In the end (Sec. 3), our definition of runs' will look very roughly like this:

```
runs'F
  :: ([Nat] -> Stream Nat -> Stream [Nat])
  -> ([Nat] -> Stream Nat -> Stream [Nat])
  -> [Nat] -> Stream Nat -> Stream [Nat]
runs'F rec corec run (x : y : xs)
  = if y < x
      then rec (run :' x) (y : xs)
      else (run :' x) : corec [] (y : xs)

runs' :: [Nat] -> Stream Nat -> Stream [Nat]
runs' = cofixWf runs'F
```

runs'F is an open recursive functional corresponding to runs', so the recursive calls in both branches have been replaced by arguments rec and corec. The first stands for a well-founded recursive call, the second for a guarded corecursive call. cofixWf then "ties the knot", forming the greatest fixed point of runs'F, which is precisely runs'. And since cofixWf will be a regular function, it will ensure that all functions defined in terms of it are productive.

We implement our cofixWf in Agda (version 2.5.2) [32], a total functional language with dependent types. The remainder of this paper is generated from a literate Agda file, so all coloured code listings are typechecked. However, we take some liberties reusing names and hiding incidental details (with all omitted definitions and proofs available in this paper's source code). Familiarity with Agda and the commonly used parts of its standard library (version 0.13) [39] is assumed, but we will explain some of the more esoteric details. This paper is accompanied by an Agda library called wellfounded-coind [28] which fully elaborates the ideas presented herein. For simplicity, we specialise all definitions to the universes Set and Set_1 in the paper; the library contains fully universe-polymorphic definitions.

[1] We will call greatest fixed points 'cofixpoint' to distinguish them from least fixed points. 'recursive' will refer to either (a) any self-referential definition, whether it is justified by structural and well-founded recursion or corecursion; or (b) more specifically only structural and well-founded recursion as opposed to corecursion. The context will make it clear which meaning is intended.

In the remainder of this text, we will first review the necessary background on coinduction and corecursion in Agda (2.1), sized types (2.2), well-founded recursion (2.3) and M-types (2.4). Then we will proceed to define cofixWf for non-indexed data types (3) and prove its fixpoint equation (3.3). After that, a generalisation of cofixWf to indexed families of data types (4) and an extended example concerning the generation of trace languages from session types (5) are presented. Finally, we discuss the limitations of our approach (6) and compare it with related work (7).

2 Background

2.1 Coinduction and Corecursion

In the context of programming languages and type theory, coinductive data types are traditionally thought of as potentially-infinite constructor trees [16, 24]. They can be used to represent a variety of useful data structures: guaranteed-infinite streams, potentially-infinite ("lazy") lists, potentially-infinite trees, and so on.

In this paper, however, we use a subtly different characterisation of coinductive types, discovered by Abel and Pientka [5] and implemented in recent versions of Agda. It emphasises the destructors, or observations, that a coinductive type allows. For example, observing a stream yields an element (the stream's 'head') and another stream (its 'tail'). As such, coinductive types resemble processes which can be queried for information. Consequently, coinductive types are not defined by constructors, but rather by how each value responds to all possible observations.

For example, consider the type of streams:

```
record Stream (A : Set) : Set where
  coinductive
  field
    head : A
    tail : Stream A
```

A record is a one-constructor data type, and the constructor has the given fields (here: head and tail). Declaring the record as coinductive enables the construction of infinite values by corecursion, which we will see momentarily. It also forbids pattern matching on values of the record, rendering Streams's constructor positively useless.

Observations of record types are defined by copatterns [7]. A copattern (technically: destructor copattern) has the form p .proj $x_1 \dots x_n$, where p is a pattern (potentially containing other copatterns) and proj is the name of a record field, which may be applied to arguments x_1 through x_n ($n \geq 0$). An equation with left hand side p .proj $x_1 \dots x_n$ then defines the value of proj $y\ x_1\ \dots\ x_n$ whenever y matches p.

To illustrate, let us define the stream of natural numbers with a given starting point:

```
naturalsFrom : ℕ → Stream ℕ
naturalsFrom n .head  = n
naturalsFrom n .tail  = naturalsFrom (suc n)
```

The first equation defines the head of naturalsFrom n (for any n), the second the tail.

naturalsFrom is defined *corecursively*: A self-referential call appears in the right-hand side of the second clause, but in contrast with definitions by structural recursion, integersFrom need not be applied only to structurally smaller arguments in its own body. In Sec. 2.2, we explore the conditions under which a corecursive call is admissible in more detail.

One question remains: How do we define coinductive sum types, like trees, when records only have one constructor? In this paper, we do so by forming the greatest fixed point of a functor, with the latter implemented as a non-recursive sum type. For example, the following type Tree represents unlabeled potentially-infinite binary trees:

```
data TreeF (Tree : Set) : Set where
  tip    :                    TreeF Tree
  branch : Tree → Tree →      TreeF Tree

record Tree : Set where
  coinductive
  field force : TreeF Tree
```

Observing a Tree via force yields a TreeF which we can analyse to determine whether it is a tip or a branch, and to obtain the subtrees in the latter case. An alternative way to construct coinductive types, which we do not use, is by mutually defining a coinductive record and an inductive data type; for an example, see [4].

2.2 Sized Types

As mentioned in the introduction, Agda must enforce that all recursive functions terminate and all corecursive functions are productive. Intuitively, a corecursive definition is productive if observing it only ever takes a finite amount of time (or evaluation steps) [36]. In other words, every observation terminates.

The traditional way to ensure this is the guardedness condition [16, 24]. It essentially stipulates that in a function defined by corecursion, recursive calls may only appear directly under a constructor of a coinductive data type. But this is obviously not applicable to the system of coinduction discussed above, which effectively gets rid of coinductive constructors entirely.

For this reason, among others which will become apparent later, we use *sized types* [6, 2], another recent addition to Agda.

Agda's implementation of sized types first defines a few new primitives. Size : SizeUniv (where SizeUniv is a new universe) is the type of sizes and may for most purposes be thought of as a natural number or infinity. ↑s : Size is the successor of size s. Size< s : SizeUniv is the type of sizes less than s, and ∞ : Size is a size such that s : Size< ∞ holds for every s. Sizes cannot be pattern matched on, so for a given size s, it is impossible to determine whether or not it is of the form ↑t for some t; this restriction is required for consistency [6].

Sizes are not treated specially by Agda's type checker except for some intuitive subtyping rules. In particular, s : Size< t implies both s : Size< (↑t) and s : Size. Indeed, Size may be regarded as an abbreviation for Size< ∞.

Given this vocabulary for sizes, Agda's termination checker is extended in a straightforward manner: A function which takes a size argument s is recognised as terminating if recursive calls take only smaller sizes $t : \mathsf{Size}< s$ as arguments.

We must thus design our inductive data types such that recursive function definitions gain access to sizes which justify termination. This can be done by adding a size parameter (or index) to a regular data type, with the understanding that this parameter represents an upper bound on the height of a particular value's constructor tree. For example, sized lists are defined as follows:

```
data List′ (s : Size) (A : Set) : Set where
  nil  :                                  List′ s A
  cons : ∀ {t : Size< s} → A → List′ t A → List′ s A
```

The tree height of nil is zero, so any size is an (inclusive) upper bound. cons x xs adds one constructor on top of xs, so s is an upper bound on its tree height if $t < s$ is an upper bound on xs's.

Given this definition, recursion on the size of a list emulates structural recursion. For example, map can be defined as follows:

```
map−List′ : ∀ {s A B} → (A → B) → List′ s A → List′ s B
map−List′ {s} f nil                = nil
map−List′ {s} f (cons {t} x xs) = cons (f x) (map−List′ {t} f xs)
```

This definition's termination is justified independently by the familiar structural recursion and a size-based termination argument. For the latter, consider the second clause: map−List′ is called recursively only at size t, and by the type of cons, we know that $t : \mathsf{Size}< s$; thus, the recursive call is acceptable. Here and in the following, we only give explicit size arguments for clarity; Agda would be able to infer them.

So far, we have not gained much by using size-based termination. To demonstrate a situation where it comes in handy, consider the type of finitely branching trees:

```
data Tree (s : Size) (A : Set) : Set where
  tip  :                                          Tree s A
  node : ∀ {t : Size< s} → A → List′ ∞ (Tree t A) → Tree s A
```

Again, we define the mapping operation:

```
map−Tree : ∀ {s A B} → (A → B) → Tree s A → Tree s B
map−Tree {s} f tip = tip
map−Tree {s} f (node {t} x ts)
  = node (f x) (map−List′ (map−Tree {t} f) ts)
```

This definition is accepted although map−Tree is not structurally recursive and would be rejected without the size typing: In the right-hand side of the second clause, map−Tree is not applied to a subterm of the input tree. Yet we know that the list ts only contains trees which are strictly smaller than the input (via the typing of node), so we can use map−Tree at size t, justifying termination. This is a typical example of how higher-order functions (here map−List′) become usable due to size typing.

Let us now turn our attention towards coinductive types. Here, we identify the size of a sized type with an upper bound on its observation depth, which is the number of projections that may be applied to it. For example, sized streams are defined as follows:

```
record Stream (s : Size) (A : Set) : Set where
  coinductive
  field
    head : A
    tail : ∀ {t : Size< s} → Stream t A
```

If a stream has observation depth $s = 0$, we cannot give a t : Size< s to tail, so the tail cannot be observed. For $s \geq 1$, we get back a tail which permits a strictly smaller number of observations than the original stream. A stream with finite observation depth may thus be regarded as partially defined.

This setup has a remarkable consequence for productivity checking. We had already mentioned that productivity may be understood as the ability to evaluate every observation in finite time. The combination of coinductive records and sized types realises this intuition by reducing productivity checking to termination checking: In definitions by corecursion, arguments to recursive calls must be strictly decreasing, just as in definitions by structural or size-based recursion. This guarantees that every observation terminates. Consider the familiar stream of naturals:

```
naturalsFrom : ∀ {s} → ℕ → Stream s ℕ
naturalsFrom {s} n .head     = n
naturalsFrom {s} n .tail {t} = naturalsFrom {t} (suc n)
```

In the second clause, the typing of tail ensures that t : Size< s, so recursive calls only happen on smaller sizes, convincing the termination checker that the definition as a whole terminates.

2.3 Well-founded Recursion

With termination checking being an undecidable problem, Agda's termination checker must rely on heuristics in its assessment of whether any particular function terminates, rejecting some valid programmes. The basic principle is that functions must be structurally recursive, meaning that recursive calls must only be performed on subterms of the original function arguments. Even though various extensions to the checker relax this restriction [3], and sized types permit abstraction over syntactic details, it is still challenging to define many, even "obviously" terminating, functions.

Well-founded recursion [8, 31, 10] is an established technique to overcome these difficulties, and is available in Agda's standard library under the Induction namespace. It is an abstraction defined within the language, and therefore does not expand the set of functions expressible in Agda. Still, defining non-structurally recursive functions in terms of well-founded recursion is often more convenient than fighting the termination checker on foot.

Technically, the main ingredient of well-founded recursion is an accessibility predicate Acc on elements of a carrier set A, parameterised by a relation _<_.[2] An element x of A is accessible wrt. _<_ iff any $y < x$ is also accessible: [3]

```
data Acc {A} (_<_ : Rel A) : A → Set where
  acc : ∀ {x} → (∀ y → y < x → Acc _<_ y) → Acc _<_ x
```

Since Acc is inductively defined, a proof of Acc x implies that there can be no infinite decreasing chain $x > x_1 > x_2 > \ldots$ starting at x; for otherwise there would have to be an infinite chain of acc constructors.

A relation _<_ : Rel A is well-founded iff every element of A is accessible wrt. _<_:

```
Well-founded : ∀ {A} → Rel A → Set
Well-founded _<_ = ∀ x → Acc _<_ x
```

Now consider a function f : $A \to B$ such that within the body of f x, recursive calls to f only take arguments $y < x$. Then f must terminate since otherwise, we would have an infinite descending chain $x > y > \ldots$ of arguments to recursive calls, which well-foundedness prohibits. Usefully, we can formalise this insight in Agda, defining the fixpoint combinator fixWf:[4]

```
module _ {A B : Set} {_<_ : Rel A}
  (F : (x : A) → ((y : A) → y < x → B) → B)
  where

  fixWf′ : (x : A) → Acc _<_ x → B
  fixWf′ x (acc rs) = F x (λ y y<x → fixWf′ y (rs y y<x))

  fixWf : Well-founded _<_ → A → B
  fixWf <-wf x = fixWf′ x (<-wf x)
```

F is an open recursive functional whose second argument stands in for recursive calls. Its type ensures that arguments y to these calls are strictly smaller than F's input, x. Operationally, fixWf′ implements the fixpoint equation fixWf′ $F\ x = F\ x$ (fixWf′ F), modulo the bookkepping required for the accessibility predicate. fixWf′ is recognised as terminating because it is structurally recursive in its second argument.

With this combinator in hand, we are able to define non-structurally recursive functions by writing them in open recursive form and proving that arguments to recursive calls always decrease. Which relation to use usually follows directly from an informal termination argument, though it can sometimes be tricky to prove well-foundedness. Luckily, there are combinators which allow for modular well-foundedness proofs [33]. We will use one of them momentarily.

[2] Despite the choice of name, _<_ need not be an order.

[3] Rel A is an abbreviation for $A \to A \to$Set.

[4] The form module _ … where introduces an anonymous module. Its definitions are added to the enclosing scope when leaving the module block. In effect, all definitions within the anonymous module are parameterised by the module parameters. This is similar to Coq's sections.

As an example, consider a close relative to runs from the introduction: firstRun, which extracts the first run of decreasing numbers from a stream (using the sized streams from Sec. 2.2). A direct formulation of this function is rejected by the termination checker:

```
{−# NON_TERMINATING #−}
firstRun′ : List ℕ → Stream ∞ ℕ → List ℕ
firstRun′ run xs with head (tail xs) <? head xs
... | yes _ = firstRun′ (run ::′ head xs) (tail xs)
... | no  _ = run ::′ head xs

firstRun : Stream ∞ ℕ → List ℕ
firstRun = firstRun′ []
```

_<?_ is a decision procedure for the _<_ relation, meaning that x <? y returns either yes p with $p : x < y$ or no np with $np : x \not< y$. ::′ appends a single element to a list. firstRun′ is obviously not structurally recursive: Recursing over the infinite stream does no good for termination.

In order to convince the termination checker, we must first identify a well-founded relation on streams which decreases on recursive calls of firstRun′. The following relation _<<_, which compares the heads of two streams, fits the bill:

```
_on_ : ∀ {A B} → (Rel B) → (A → B) → Rel A
R on f = λ x y → R (f x) (f y)

_<<_ : Rel (Stream ∞ ℕ)
_<<_ = _<_ on head
```

To prove that _<<_ is well-founded, we first note that the less-than relation on natural numbers is trivially well-founded. Using one of Paulson's combinators [33], here called wf−inverse−image, it is then easy to prove _<<_ well-founded.

```
<−wf : Well−founded _<_

wf−inverse−image : ∀ {A B} {_<_ : Rel B} {f : A → B}
  → Well−founded _<_
  → Well−founded (_<_ on f)

<<−wf : Well−founded _<<_
<<−wf = wf−inverse−image <−wf
```

Now we need to define the open recursive functional corresponding to firstRun′. The functional can take only one argument while firstRun′ requires two, so we need to use a pair combining the two arguments. (Alternatively, one could define a fixpoint combinator for functions with two arguments.) Lifting our well-founded relation to these pairs requires another application of wf−inverse−image:[5]

[5] We do not know why the f for wf−inverse−image can sometimes be inferred and sometimes not.

```
In : Set
In = List ℕ × Stream ∞ ℕ

_<<<_ : Rel In
_<<<_ = _<<_ on proj₂

<<<-wf : Well-founded _<<<_
<<<-wf = wf-inverse-image {f = proj₂} <<-wf
```

The functional is then constructed by replacing all recursive calls in the body of (an uncurried version of) firstRun′ with calls to a new argument *rec*:

```
firstRun′F : (x : In) → ((y : In) → y <<< x → List ℕ) → List ℕ
firstRun′F (run , xs) rec with head (tail xs) <? head xs
... | yes p = rec (run ::′ head xs , tail xs) p
... | no  _ = run ::′ head xs
```

For the second argument of *rec*, we need a proof that tail *xs* << *xs*, i.e. head (tail *xs*) < head *xs*; but this is precisely what the decision procedure for _<_ gives us in the first branch. Having thus defined the functional, we can take its least fixed point to obtain a function that is extensionally equal to our first formulation of firstRun′.

```
firstRun′ : List ℕ → Stream ∞ ℕ → List ℕ
firstRun′ run xs = fixWf firstRun′F <<<-wf (run , xs)
```

2.4 Containers and M-Types

The last preliminary question we must address is how to abstract over coinductive data types. Our fixpoint combinator will be similar to fixWf from Sec. 2.3, but it will be defined by corecursion. Therefore, it must construct a coinductive type; and we must know *which* type because we need access to its destructors. Hence, we need a data type which can represent any coinductive type definable in Agda.

To obtain a data type which can represent all coinductive types, we view coinductive types as greatest fixed points over functors. We may then define a coinductive type CoFix : (Set → Set) → Set which, given a functor F : Set → Set provided by the user, yields a type whose destructors we know and which we can therefore construct corecursively.

However, Agda throws a (well-justified) spanner in the works and does not accept the following simple definition, popularised among functional programmers by *Data types à la carte* [37]:

```
{-# NO_POSITIVITY_CHECK #-}
record CoFix (F : Set → Set) : Set where
  coinductive
  field unfold : F (CoFix F)
```

This definition is rejected because, for consistency, recursive data types must be fixed points of *strictly positive* functors [24], and there is no way to ensure that a particular F chosen by the user would be strictly positive. MiniAgda's polarity annotations [2] would let us require this, but Agda does not currently support them.

Therefore, we need an encoding of the strictly positive functors. Abbott et al.'s *containers* [1, 9] (also known as interaction structures [25]) provide precisely that.

A container consists of a set of shapes Shape and for each shape s a set of positions Pos s:

```
record Container : Set₁ where
  constructor _◁_
  field
    Shape : Set
    Pos : Shape → Set
```

For a container c, we define the extension $[\![c]\!]$:

```
⟦_⟧ : Container → Set → Set
⟦ Shape ◁ Pos ⟧ = λ X → Σ Shape (λ s → Pos s → X)
```

$[\![c]\!]$ is a functor, as witnessed by the following mapping operation:

```
map : ∀ {A B} c → (A → B) → ⟦ c ⟧ A → ⟦ c ⟧ B
map (_ ◁ _) f (shape , xs) = shape , f ∘ xs
```

map's definition reflects the common intuition that mapping over a functor changes its "contents" but not its "shape". It is easy to see that map fulfils the functor laws.

Remarkably, any strictly positive functor is isomorphic to the extension of some container [1, Corollary 6.1]. For simple functorial data types, say F : (A : Set) → Set, the translation is quite mechanical: Shape indexes F's constructors, including any data they contain apart from occurrences of A. Pos s indexes, for a constructor s, the occurences of A. For example, consider the functor ListF A whose least fixed point are the lists over A:

```
data ListF (A : Set) (List : Set) : Set where
  []   :                 ListF A List
  _::_ : A → List →      ListF A List
```

There is a bijection (modulo functional extensionality) between ListF and the following container functor. The bijection's implementation illustrates how shape and positions correspond to constructors and their arguments.[6]

```
ListC : Set → Container
ListC A = Shape ◁ Pos
```

[6] In the definition of ListC, the form module ListC where acts like a where clause which additionally defines the module ListC containing Shape and Pos. The form $\lambda()$ in ListF→ListF′ is an anonymous function which eliminates the absurd type $\bot$.

```
  module ListC where
    data Shape : Set where
      nil  :       Shape
      cons : A →  Shape

    Pos : Shape → Set
    Pos nil       = ⊥
    Pos (cons _) = ⊤

ListF′ : Set → Set → Set
ListF′ A = ⟦ ListC A ⟧


ListF→ListF′ : ∀ {A List} → ListF A List → ListF′ A List
ListF→ListF′ []       = ListC.nil     , λ()
ListF→ListF′ (x :: xs) = ListC.cons x , λ _ → xs

ListF′→ListF : ∀ {A List} → ListF′ A List → ListF A List
ListF′→ListF (ListC.nil     , _)  = []
ListF′→ListF (ListC.cons x , xs) = x :: xs tt
```

Now we can define the sized greatest fixed point over arbitrary containers, and thus a uniform representation of every non-indexed coinductive data type expressible in Agda. This fixed point is called an M-type after the related category-theoretic notion [41].

```
record M (c : Container) (s : Size) : Set where
  constructor m
  coinductive
  field inf : ∀ {t : Size< s} → ⟦ c ⟧ (M c t)

open M public
```

M is almost identical to the coinductive record from our earlier attempt at defining a universal cofixpoint type that ran afoul of the strict positivity condition. But in this case, we may expand the definition of ⟦_⟧, yielding the following type of M's single constructor:

```
m : (∀ {t : Size< s} → Σ (Shape c) (λ s → Pos c s → M c t)) → M c s
```

This makes it obvious that M does not appear to the left of any arrow in the type of m's argument – regardless of which container we choose – and so M is strictly positive.

We may now construct coinductive data structures, like possibly-infinite lists, as greatest fixed points over container functors:

```
List∞ : Set → Size → Set
List∞ A = M (ListC A)
```

As an example of how to work with them, we implement the usual mapping operator:

```
map-Listω : ∀ {s A B} → (A → B) → Listω A s → Listω B s
map-Listω f xs .inf with inf xs
... | ListC.nil       , _   = ListC.nil            , λ()
... | ListC.cons x , xs′  = ListC.cons (f x) , λ _ → map-Listω f (xs′ tt)
```

Apart from some syntactic overhead, the definition is exceedingly similar to the usual formulation of mapping on (lazy) lists.

In the following, we occasionally use a "weak head normal form" of M-types:

```
Whnf : Container → Size → Set
Whnf c s = ⟦ c ⟧ (M c s)
```

One may picture this as a potentially infinite structure whose first part – a layer of the $[\![c]\!]$ functor – is immediately accessible. For streams, this is a pair of an element and a stream. The terminology is due to Danielsson [18].

As noted above, our M-type cannot be used to represent coinductive indexed families such as coinductive relations. This requires an update to indexed containers, which we postpone until Sec. 4.

3 Cofixpoint Construction

3.1 Simple Cofixpoint

Before considering how to integrate well-founded recursion, it will be instructive to implement a regular cofixpoint combinator which we can later extend. Our solution, which is explained in detail below, reads as follows:

```
cofix : ∀ {s c} {A : Set}
  → (F  : ∀ {t}
        → (A → M c t)
        → A
        → Whnf c t)
  → A
  → M c s
cofix {s} F x .inf {t} = F {t} (cofix {t} F) x
```

Operationally, cofix implements the usual fixpoint equation, "replacing" the open recursion in the functional F with a recursive call to cofix F.

More interesting is the typing that makes Agda's termination checker accept this definition, and in particular the type of F. Receiving as input an element of A and the ability to recurse, producing an M c t, it must return a Whnf c t. Thus, it must add one observation – one layer of the $[\![c]\!]$ functor – to the recursive call if it performs one. This guarantee is achieved by abstracting over the size t of the participating data types, and it reflects precisely the restrictive form of guarded corecursion.

The size typing also makes the termination checker accept this definition: Via the type of inf, we know that t : Size< s, and cofix is only called recursively at size t. Hence, sized types are crucial to our technique.

As an example of how to use cofix, we reimplement map−Listω. Compared to the direct implementation in Sec. 2.4, almost no changes are required.[7]

```
module _ {A B} (f : A → B) where

  map−Listω′F : ∀ {t}
    → (Listω A ω → Listω B t)
    → Listω A ω → Whnf (ListC B) t
  map−Listω′F map−Listω′ xs with inf xs
  ... | ListC.nil       , _   = ListC.nil            , λ()
  ... | ListC.cons x , xs′  = ListC.cons (f x) , λ _ → map−Listω′ (xs′ tt)

  map−Listω′ : ∀ {s} → Listω A ω → Listω B s
  map−Listω′ = cofix map−Listω′F
```

3.2 Cofixpoint with Well-founded Recursion

We can now extend our cofixpoint combinator with well-founded recursion. The central idea is to give the functional F two different ways to recurse: either corecursively, as above, or by decreasing a well-founded relation. Again, we show the construction before explaining it:

```
module _
  {A : Set} {c : Container}
  {_<_ : Rel A} (<−wf : Well−founded _<_)
  (F : ∀ {t}
    → (x : A)
    → (A → M c t)
    → ((y : A) → y < x → Whnf c t)
    → Whnf c t)
  where

  cofixWf′ : ∀ {s} → (x : A) → Acc _<_ x → M c s
  cofixWf′ {s} x (acc rs) .inf {t}
    = F {t} x
        (λ y → cofixWf′ {t} y (<−wf y))
        (λ y y<x → inf (cofixWf′ {s} y (rs y y<x)) {t})

  cofixWf : ∀ {s} → A → M c s
  cofixWf x = cofixWf′ x (<−wf x)
```

The type of F has changed to accomodate a new option for recursion: In addition to a function of type $A \to \mathsf{M}\ c\ t$ – the corecursive part, which we already discussed above

[7] However, the input list has size ω, and so the fact that map−Listω′ is size-preserving is not reflected in its type. There is a variant of cofix which allows us to keep the more precise type, but since a cofixpoint combinator which integrates well-founded recursion – the target of this development – cannot be size-preserving anyway, we will not go into more detail on this.

–, it receives a function of type $(y : A) \to y < x \to \mathsf{Whnf}\ c\ t$. This function represents a well-founded recursive call and thus requires its argument to be less than F's input wrt. _<_. It returns an unfolded M-type which F can return directly, so the functional does not need to specify the returned value's next observation.

Operationally, the corecursive part remains the same, except for the proof of accessibility of y which we thread through. The size typing still works out because we only call cofixWf′ at size t.

For the recursive part, we add a new recursive call on the last line of cofixWf′. It works like the recursive call in our fixpoint combinator for well-founded recursion: Since we know y to be less than the original input x, we can recurse structurally on the accessibility predicate acc rs.

Since we need to unroll the result in order to return the unrolled M-type, we call cofixWf′ at size s. The existence of a t : Size< s proves that s is at least one, which justifies observing the M-type via inf. But this implies that we call cofixWf′ recursively at the same size, s, it was originally called with, which does not, by itself, justify termination.

Still, the function terminates because at each recursive call, either the size argument decreases (in the corecursive case) or it remains the same and we recurse structurally into the accessibility predicate. This is a lexicographic termination measure, which Agda's termination checker [3] is able to recognise.

With our new cofixpoint combinator in hand, we return to the example from the introduction. Recall that the function runs is supposed to extract a stream of decreasing runs from a stream of numbers. It is defined in terms of a helper function runs′ whose naïve definition would not satisfy Agda's termination checker:

```
runs' :: [Nat] -> Stream Nat -> Stream [Nat]
runs' run (x : y : xs)
  = if y < x
      then runs' (run ++ [x]) (y : xs)
      else (run ++ [x]) : runs' [] (y : xs)
```

To tackle the problem, we first define sized streams in terms of M-types, along with some utility functions.

```
StreamC : Set → Container
StreamC A = A ◁ (λ _ → ⊤)

Stream : Set → Size → Set
Stream A = M (StreamC A)

StreamWhnf : Set → Size → Set
StreamWhnf A = Whnf (StreamC A)

head : ∀ {A s} {t : Size< s} → Stream A s → A
head xs = proj₁ (inf xs)
```

```
tail : ∀ {A s} {t : Size< s} → Stream A s → Stream A t
tail xs = proj₂ (inf xs) tt

consWhnf : ∀ {A s} → A → Stream A s → StreamWhnf A s
consWhnf x xs = x , λ _ → xs

cons : ∀ {A s} → A → Stream A s → Stream A (↑ s)
cons x xs .inf = consWhnf x xs
```

Like firstRun′F from Sec. 2.3, runs′ takes two arguments which we must combine into a pair. The well-founded relation which recursive calls decrease is the same as for firstRun′F.

```
In : Set
In = List ℕ × Stream ℕ ∞

_<<<_ : Rel In
_<<<_ = _<_ on (head ∘ proj₂)

<<<-wf : Well-founded _<<<_
<<<-wf = wf-inverse-image <-wf
```

Now we can define a functional which has the form required by cofixWf:

```
runs′F : ∀ {t}
  → (x : In)
  → (In → Stream (List ℕ) t)
  → ((y : In) → y <<< x → StreamWhnf (List ℕ) t)
  → StreamWhnf (List ℕ) t
runs′F (run , xs) corec rec with head (tail xs) <? head xs
... | yes p  = rec (run ::′ head xs , tail xs) p
... | no np  = consWhnf (run ::′ head xs) (corec ([] , tail xs))
```

The recursive call in the yes branch has been replaced by a call to the argument *rec*, signifying a call justified by well-founded recursion. Like in firstRun′F, the proof that its argument is less than the original input according to _<<<_ follows directly from the conditional. Meanwhile, the recursive call in the no branch has been replaced by *corec*, which can only be used because it is "guarded" by consWhnf.

runs′ is then simply the greatest fixed point of runs′F, and runs is defined in terms of it.

```
runs′ : In → Stream (List ℕ) ∞
runs′ = cofixWf <<<-wf runs′F

runs : Stream ℕ ∞ → Stream (List ℕ) ∞
runs xs = runs′ ([] , xs)
```

3.3 Fixpoint Equation

Proving facts about functions which use our cofixpoint combinator does not, for the most part, require any special infrastructure: Proofs by coinduction generally construct coinductive relations (for example bisimulations), and the indexed version of our combinator can be used to help with these constructions if required (see Sec. 4).

However, we can provide one ingredient which simplifies proofs about cofixWf-based definitions. The characteristic fixpoint equation for cofixWf holds almost by definition, but not quite: Reducing cofixWf replaces the recursive calls in the functional F with calls to cofixWf′ rather than cofixWf. This is a significant obstacle for recursive or corecursive proofs about any cofixWf-based function, since these proofs have to incorporate a lemma about cofixWf′.

For this reason, we wish to prove that cofixWf is a proper fixed point combinator. The following fixpoint equation expresses this:

```
cofixWf−unfold : ∀ (x : A)
  → inf (cofixWf <−wf F x)
  ≡ F x (cofixWf <−wf F) (λ y _ → inf (cofixWf <−wf F y))
```

The proof of cofixWf−unfold is quite similar to that of the corresponding fixpoint equation for fixWf-based well-founded recursion [10]. Like the latter, it requires an additional extensionality property of F:

```
module _
  (F−ext : ∀ x {f f′ g g′}
    → (∀ y → f y ≡ f′ y)
    → (∀ y y<x → g y y<x ≡ g′ y y<x)
    → F x f g ≡ F x f′ g′)
  where
```

The two hypotheses state that f is extensionally equal to f' and g to g'. If that is the case, $F\ x\ f\ g$ should be equal to $F\ x\ f'\ g'$. This follows immediately from functional extensionality, and hence every F has this property; but since functional extensionality cannot be proven in Agda, we must either postulate it or prove $F{-}ext$ manually for relevant functionals F.

Given $F{-}ext$, we may prove that the Acc argument to cofixWf′ is irrelevant:

```
cofixWf′−Acc−irrelevant : ∀ {x} (acc acc′ : Acc _<_ x)
  → inf (cofixWf′ <−wf F x acc )
  ≡ inf (cofixWf′ <−wf F x acc′)
cofixWf′−Acc−irrelevant (acc rs) (acc rs′)
  = F−ext _
    (λ _ → refl)
    (λ y y<x → cofixWf′−Acc−irrelevant (rs y y<x) (rs′ y y<x))
```

This allows us to prove the desired cofixWf−unfold:

```
cofixWf−unfold : ∀ x
  → inf (cofixWf <−wf F x)
```

```
    ≡ F x (cofixWf <−wf F) (λ y _ → inf (cofixWf <−wf F y))
cofixWf−unfold x with (<−wf x)
... | (acc rs)
    = F−ext _
        (λ _ → refl)
        (λ y y<x → cofixWf′−Acc−irrelevant (rs y y<x) (<−wf y))
```

Applying cofixWf−unfold, we may verify that runs′ is extensionally equal to the version we would write in a partial language. To do so, we first define the expected runtime behaviour:

```
runs′−body : List ℕ × Stream ℕ ∞ → Stream (List ℕ) ∞
runs′−body (run , xs) with head (tail xs) <? head xs
... | yes _ = runs′ (run ::′ head xs , tail xs)
... | no  _ = cons (run ::′ head xs) (runs′ ([] , tail xs))
```

cofixWf−unfold then allows us to prove that observing the result of runs′ is the same as observing the result of runs′−body for any input.

```
runs′−unfold₁ : ∀ run xs
  → inf (runs′ (run , xs)) ≡ inf (runs′−body (run , xs))
runs′−unfold₁ run xs
  rewrite cofixWf−unfold <<<−wf runs′F runs′F−ext (run , xs)
  with head (tail xs) <? head xs
... | yes _ = refl
... | no  _ = refl
```

The third argument to cofixWf−unfold, runs′F−ext, is a routine proof of the extensionality property $F{-}ext$ specialised to runs′F.

runs′−unfold$_1$ is useful because it allows us to treat runs′ as a regular recursive function which we can unfold without ever having to deal with cofixWf – but only under an observation via inf, which is cumbersome in practice. Ideally, we would like to prove a stronger statement:

```
runs′−unfold₂ : ∀ run xs → runs′ (run , xs) ≡ runs′−body (run , xs)
```

This would allow us to unfold runs′ regardless of the surrounding context.

Alas, this proposition is, to our knowlegde, not provable in Agda. In general, for two inhabitants r_1, r_2 of a coinductive record with fields $f_1 \dots f_n$, we cannot conclude $r_1 \equiv r_2$ from $\bigwedge_{i=1}^{n}(f_i\ r_1 \equiv f_i\ r_2)$ because pattern matching on coinductive records is not allowed.

However, we conjecture that this relationship between equality of coinductive records and equality of their fields may be added as an axiom without introducing inconsistency. The rationale is similar to that for functional extensionality: Two inhabitants of a record can only be distinguished by observing them, so if all possible observations are equal, the inhabitants are interchangeable in any context. Hence, we postulate the following for M:

```
postulate
  M−Extensionality : ∀ {c s} {t : Size< s} {x y : M c s}
    → inf x ≡ inf y → x ≡ y
```

This allows us to simplify $\mathsf{runs'{-}unfold_1}$:

```
runs′−unfold₂ : ∀ run xs → runs′ (run , xs) ≡ runs′−body (run , xs)
runs′−unfold₂ run xs = M−Extensionality (runs′−unfold₁ run xs)
```

We stress that our development does not require the use of M−Extensionality. As an alternative, it is always possible to prove, for any function f which takes an M-type as its argument, that inf $x \equiv$ inf y implies f $x \equiv$ f y. However, for convenience, we assume M−Extensionality in the remainder of this text.

4 Generalisation to Indexed Types

As mentioned, our general coinductive type M can only represent non-indexed data types. While this is sufficient for many purposes, we would also like to use cofixWf when constructing indexed types such as coinductively defined relations. This section therefore generalises our previous results to the indexed case, reusing the names for brevity.

4.1 Indexed Containers and M-types

Going from regular to indexed M means going from regular to indexed containers [9, 25]: While the former represent strictly positive functors, the latter represent strictly positive indexed functors. Indexed containers are defined as follows:

```
record Container (I : Set) : Set₁ where
  constructor _◁_/_
  field
    Shape : I → Set
    Pos   : ∀ {i} → Shape i → Set
    next  : ∀ {i} → (c : Shape i) → Pos c → I
```

The extension of an indexed container is an indexed functor, i.e. a function between sets indexed by I:

```
⟦_⟧ : ∀ {I} → Container I → (I → Set) → (I → Set)
⟦ Shape ◁ Pos / next ⟧ X i
  = Σ (Shape i) (λ s → (p : Pos s) → X (next s p))
```

Like with non-indexed containers, the extension is a pair whose first element is a Shape s, which may be thought of as a constructor (if the functor was defined as a regular indexed data type). The second element is a function which returns, for each position p in Pos s, a member of the indexed input set X. Its index is determined by next, and can depend on both the shape and the position.

Indexed M-types are sized greatest fixed points over container functors:

```
record M {I} (c : Container I) (s : Size) (i : I) : Set where
  coinductive
  field inf : ∀ {t : Size< s} → ⟦ c ⟧ (M c t) i

Whnf : ∀ {I} → (Container I) → Size → I → Set
Whnf c s i = ⟦ c ⟧ (M c s) i
```

To represent families with multiple indexes, choose the product of the index sets for I. By choosing the nullary product, $\top$, we obtain a non-indexed M-type.

As an example, we define the predicate All in terms of the indexed M. Given a $P : A \to \mathsf{Set}$, All P is a predicate on A-streams which holds iff every element of a stream satisfies P.

```
AllC : ∀ {A} → (A → Set) → Container (Stream A ∞)
AllC {A} P = Shape ◁ (λ {xs} → Pos {xs}) / (λ {xs} → next {xs})
  module AllC where
    Shape : Stream A ∞ → Set
    Shape xs = P (head xs)

    Pos : ∀ {xs} → Shape xs → Set
    Pos _ = ⊤

    next : ∀ {xs} → (s : Shape xs) → Pos {xs} s → Stream A ∞
    next {xs} _ _ = tail xs

All : ∀ {A} → (A → Set) → Size → Stream A ∞ → Set
All P = M (AllC P)

AllWhnf : ∀ {A} → (A → Set) → Size → Stream A ∞ → Set
AllWhnf P = Whnf (AllC P)
```

Again, Shape encodes constructors and their non-recursive fields – here just one, which contains a proof that head xs satisfies P. Pos enumerates recursive positions, of which there is precisely one. next gives the recursive position's index, which is tail xs because we desire a proof of the statement All P (tail xs). The similarity to our stream definition from Sec. 3 is no coincidence, since All is effectively a stream of proofs.

To get a feeling for the definition, we implement a few utility functions which will be needed later.

```
module _ {A} {P : A → Set} {xs : Stream A ∞} where

  All-head : ∀ {s} {t : Size< s} → All P s xs → P (head xs)
  All-head all = proj₁ (inf all)

  All-tail : ∀ {s} {t : Size< s} → All P s xs → All P t (tail xs)
```

```
All-tail all = proj₂ (inf all) tt

All-cons : ∀ {s} → P (head xs) → All P s (tail xs) → All P (↑ s) xs
All-cons p-head p-tail .inf = p-head , λ _ → p-tail
```

4.2 Cofixpoint Combinator

With indexed M-types defined, we proceed to construct an indexed version of cofixWf.

```
module _
  {I : Set} {c : Container I} {In : Set}
  {_<_ : Rel In} (<-wf : Well-founded _<_)
  {Ix : In → I}
  (F : ∀ {t}
    → (x : In)
    → ((y : In) → M c t (Ix y))
    → ((y : In) → y < x → Whnf c t (Ix y))
    → Whnf c t (Ix x))
  where

  cofixWf′ : ∀ {s} → (x : In) → Acc _<_ x → M c s (Ix x)
  cofixWf′ x (acc rs) .inf
    = F x
      (λ y → cofixWf′ y (<-wf y))
      (λ y y<x → inf (cofixWf′ y (rs y y<x)))

  cofixWf : (x : In) → M c ∞ (Ix x)
  cofixWf x = cofixWf′ x (<-wf x)
```

Operationally, cofixWf′ and cofixWf work exactly like their non-indexed variants. The only difference is in the types, where we must supply indexes. We make these dependent upon the input (via Ix) to support the use case where our output is a proof of a statement about the input. The example below illustrates this mechanism.

The indexed cofixWf also admits a fixpoint equation which we omit here because it is completely analogous to the non-indexed version.

To demonstrate the indexed cofixWf's utility, we shall prove a simple lemma about runs, namely that its output contains only elements from the input. Before being able to state this proposition, we must define some preliminary notions.

Something is an element of a stream iff we can find it in a finite prefix of said stream:

```
data _∈_ {A} : A → Stream A ∞ → Set where
  here  : ∀ {xs}                → head xs ∈ xs
  later : ∀ {x xs} → x ∈ tail xs → x ∈ xs
```

A list is a sublist of a stream iff every element of the list is also an element of the stream. For simplicity, we disregard the order of elements.

```
_⊂L_ : ∀ {A} → List A → Stream A ∞ → Set
[]       ⊂L ys = ⊤
(x :: xs) ⊂L ys = x ∈ ys × xs ⊂L ys
```

A stream is a substream of another stream iff every element of the first stream is also an element of the second stream (again disregarding order).

```
_⊆_ : ∀ {A} → Rel (Stream A ∞)
xs ⊆ ys = All (_∈ ys) ∞ xs
```

Now we can formulate the lemma we wish to prove: all lists in runs xs should be sublists of xs.

```
runs−⊂L : ∀ xs → All (_⊂L xs) ∞ (runs xs)
```

An attempt to prove this lemma directly will, however, get stuck. We must generalise the lemma to runs′, which yields the following goal:

```
runs′−⊂L : ∀ ys xs run
  → xs ⊆ ys
  → run ⊂L ys
  → All (_⊂L ys) ∞ (runs′ (run , xs))
```

This statement admits a proof by coinduction whose structure precisely mirrors that of runs′. But this means that we must use cofixWf since the body of runs′−⊂L will contain recursive calls justified by well-founded induction.

Hence, we must define a functional corresponding to runs′−⊂L. This is a little more complicated than for the cases we have seen before because there are more inputs. Of these, ys can remain constant in runs′−⊂L's body, but all other inputs take on different values in the recursive calls. Therefore, we define a compound input data type containing all of them:

```
record In (ys : Stream ℕ ∞) : Set where
  constructor build−In
  field
    xs : Stream ℕ ∞
    run : List ℕ
    sub−xs : xs ⊆ ys
    sub−run : run ⊂L ys
```

A suitable well-founded relation for In is defined in the familiar manner:

```
_<<<_ : ∀ {ys} → Rel (In ys)
_<<<_ = _<_ on (head ∘ xs)

<<<−wf : ∀ {ys} → Well−founded (_<<<_ {ys})
<<<−wf = wf−inverse−image <−wf
```

Now we are ready to define the functional for runs′−⊂L. Since the proof requires some work, we break up the definition into parts.

```
Ix : ∀ {ys} → In ys → Stream (List ℕ) ∞
Ix (build−In xs run _ _) = runs′ (run , xs)

runs′−⊂LF : ∀ ys {t}
  → (x : In ys)
  → ((y : In ys) → All (_⊂L ys) t (Ix y))
  → ((y : In ys) → y <<< x → AllWhnf (_⊂L ys) t (Ix y))
  → AllWhnf (_⊂L ys) t (Ix x)
```

Ix defines the object of interest – the stream generated by runs′ – given a particular input. In the functional, it is used as the index of the All M-type. Recall that we introduced the dependency between index and input in order to enable predicates over the input to be proven; runs′−⊂LF is an example of this.

```
runs′−⊂LF ys {t} (build−In xs run sub−xs sub−run) corec rec
  = cast (≡.cong (AllWhnf (_⊂L ys) t) (≡.sym (runs′−unfold₂ run xs))) go
where
```

The first order of business is replacing runs′ with runs′−body. To do so, we use a lemma go (to be defined momentarily) and, using runs′−unfold$_2$, replace the reference to runs′−body in its type by runs′. cast is a helper function which lets us use a value of type A where something of type B is required, provided that $A \equiv B$.[8]

```
go : AllWhnf (_⊂L ys) t (runs′−body (run , xs))
go with head (tail xs) <? head xs
... | yes p = rec (build−In (tail xs)
                            (run ::′ head xs)
                            sub−tail−xs
                            sub−run′)
                  p
... | no np = sub−run′
            , λ _ → corec (build−In (tail xs) [] sub−tail−xs tt)

sub−tail−xs : tail xs ⊆ ys

sub−run′ : (run ::′ head xs) ⊂L ys
```

sub−tail−xs and sub−run′ are simple lemmas whose implementations we omit. The meat of the proof is in go, which mirrors the structure of runs′:

- In the yes case, runs′−body (run , xs) is equal to runs′ (run ::′ head , tail xs) by conversion. Hence, we need to prove that all lists in the latter stream are sublists of ys. This is achieved by recursing with the corresponding inputs; the recursive call is justified by well-founded recursion in the familiar manner.

[8] It should be possible to perform this replacement using Agda's built-in rewriting, but this does not work in the current version of Agda (2.5.2). We suspect an error in the rewriting implementation.

- In the no case, runs′−body (*run* , *xs*) is equal to cons (*run* ::′ head *xs*) (runs′ ([] , tail *xs*)). Hence, we can build the first element of the All stream – a proof that *run* ::′ head *xs* is a sublist of *ys* – before recursing into the tail of *xs*. The corecursion is justified by "guardedness".

Having thus constructed runs′−⊂LF, runs−⊂L is simply a matter of taking the fixed point and providing the correct input.

```
runs−⊂L : ∀ xs → All (_⊂L xs) ∞ (runs xs)
runs−⊂L xs = cofixWf <<<−wf (runs′−⊂LF xs) (build−In xs [] ⊆−refl tt)
```

⊆−refl is a lemma stating that the substream relation is reflexive.

5 Application: Trace Languages of Session Types

Session types are a typing discipline for the description of communication protocols, originally developed by Honda et al. [26, 38]. We roughly follow the presentation of Gay and Vasconcelos [23], assuming a type of messages B and a type of variables Var, both equipped with a decidable equality relation. Additionally, we introduce a tag to identify the sending or receiving end of a two-way communication.

```
postulate
  B Var : Set
  _≟M_ : (x y : B)   → Dec (x ≡ y)
  _≟V_ : (x y : Var) → Dec (x ≡ y)

data Tag : Set where
  Sender Receiver : Tag
```

Session types are then defined by the following data type S:[9]

```
data S : Set where
  end    :                  S
  msg    : Tag → B → S   → S
  branch : Tag → S → S   → S
  ν      : Var → S       → S
  var    : Var           → S
```

end denotes the end of communication. msg *t m s* sends or receives a message, depending on the value of the *t* tag, and then continues with the protocol described by *s*. branch *t l r* provides a choice between protocols *l* and *r*: For *t* = Sender, a choice is offered to the communication partner, whereas *t* = Receiver demands that the communication partner offer a choice. ν is a variable binder and var denotes references to these variables. ν forms the greatest fixed point of a session type, which is a protocol

[9] ν is usually called μ. This is a misnomer since the latter is also commonly used for least fixed point combinators, while the former generates greatest fixed points.

that repeats *ad infinitum*. For example, $\nu\ X$ (msg Sender $b\ X$) describes a process that continuously sends the message b. Since the semantics of session types do not concern us much in the following, we refer to [23] for further details.

Our task is now to generate the trace language of a given session type s. This is a tree containing sent and received messages in all possible branches of the protocol described by s. And this tree can be infinite (because ν forms a greatest fixed point), so it must be coinductively defined. Effectively, the trace language of s is simply s with all variables replaced by the subtree they point to, *ad infinitum*.

In fact, this last remark gives us an implementation strategy for the generation of trace languages. We first define a naïve substitution operation:

```
_[_↦_] : S → Var → S → S
end          [ y ↦ s′ ] = end
msg b t s    [ y ↦ s′ ] = msg b t (s [ y ↦ s′ ])
branch t l r [ y ↦ s′ ] = branch t (l [ y ↦ s′ ]) (r [ y ↦ s′ ])
ν x s        [ y ↦ s′ ] with x ≟V y
... | yes _ = ν x s
... | no  _ = ν x (s [ y ↦ s′ ])
var x        [ y ↦ s′ ] with x ≟V y
... | yes _ = s′
... | no  _ = var x
```

Then the trace language of a session type is defined informally as follows:

```
{-# NON_TERMINATING #-}
tr : S → Tr
tr end           = end
tr (msg t b s)   = msg t b (tr s)
tr (branch t l r) = branch t (tr l) (tr r)
tr (ν x s)       = tr (s [ x ↦ ν x s ])
tr (var x)       = undefined
  where postulate undefined : _
```

The first three branches simply recurse over the session type. The ν branch is what makes this exercise both difficult and interesting: It first replaces x by $\nu\ x\ s$ in s, then calls itself to build the trace language of the resulting session type. This makes tr a non-terminating process: While recursing into $s\ [x \mapsto \nu\ x\ s]$, it will eventually reach $\nu\ x\ s$ again, performing the same substitution, and repeat *ad infinitum*.

tr is therefore defined by corecursion. But it is not obviously productive, since the self-referential call in the ν branch is not guarded. Indeed, there are inputs for which tr loops, such as $\nu\ x$ (var x):

$$
\begin{aligned}
& \text{tr } (\nu\ x\ (\text{var } x)) \\
= \ & \text{tr } ((\text{var } x)[\ x \mapsto \nu\ x\ (\text{var } x)\ ]) \\
= \ & \text{tr } (\nu\ x\ (\text{var } x)) \\
= \ & \ldots
\end{aligned}
$$

To have any hope of defining tr in Agda, we must therefore exclude such inputs. We do so by stipulating that ν may never be followed directly by a variable. Let Contractive : S → Set be the obvious predicate that implements this restriction.

Meanwhile, the var branch of tr is undefined. This is because only closed session types have meaningful trace languages. Hence, we define another predicate, Closed : S → Set, for session types with no free variables. The definition is standard, so we omit it (along with the vast majority of proof code in the remainder of this section, which would only distract from the application of cofixWf.) For our input, we then define the type of well-formed session types, which are session types that are both contractive and closed:

```
record SWf : Set where
  constructor swf
  field
    s : S
    contractive : Contractive s
    closed : Closed s
```

Now we want to use cofixWf (in its non-indexed form) to define tr, and so we must find a well-founded relation which decreases as we go from $\nu\ x\ s$ to $s\ [x \mapsto \nu\ x\ s]$. The central idea is that as we "substitute away" one ν, the number of ν constructors at the front of the input decreases. Formally:

```
ν−count : S → ℕ
ν−count (ν x s) = 1 + ν−count s
ν−count _       = 0


ν−expand−ν−count : ∀ {x} s
  → Contractive (ν x s)
  → ν−count (s [ x ↦ ν x s ]) < ν−count (ν x s)
```

This observation directly gives us a well-founded relation which decreases in tr's ν case:

```
_<<<_ : Rel SWf
_<<<_ = _<_ on (ν−count ∘ SWf.s)

<<<−wf : Well−founded _<<<_
<<<−wf = wf−inverse−image <−wf
```

Up to this point, we have defined trace languages only informally. We now rectify this omission, using an M-type because trace languages will be generated by cofixWf. We also give some helper functions to make the following code more readable.[10]

```
TrC : Container
TrC = Shape ◁ Pos
```

[10] The form λ where … defines an anonymous function. In the block that follows the where, each line gives an equation by pattern matching on the lambda's input(s).

```
  module TrC where
    data Shape : Set where
      end : Shape
      msg : Tag → B → Shape
      branch : Tag → Shape

    Pos : Shape → Set
    Pos end = ⊥
    Pos (msg _ _) = ⊤
    Pos (branch x) = Bool

Tr : Size → Set
Tr = M TrC

TrWhnf : Size → Set
TrWhnf s = ⟦ TrC ⟧ (Tr s)

Tr-end : TrWhnf ∞
Tr-end = TrC.end , λ()

Tr-msg : ∀ {s} → Tag → B → Tr s → TrWhnf s
Tr-msg t b tr = TrC.msg t b , λ _ → tr

Tr-branch : ∀ {s} → Tag → Tr s → Tr s → TrWhnf s
Tr-branch t l r = TrC.branch t , λ where
  true  → l
  false → r
```

In Pos, the ⊤ stands for one recursive occurence, while Bool stands for two.

With this infrastructure developed, our fixpoint combinator allows us to define trace language generation exactly as specified. However, the definition is a little obscured due to the need to provide well-formed session types to recursive calls, which we must construct out of the well-formed session types we get as input. Therefore, we extract this task to its own function, components, which also uses some auxiliary lemmas. The reader should not concern themselves too much with it, except perhaps to verify that in the ν x s case we recurse into s [x ↦ ν x s].

```
components : (s : SWf) →
  case SWf.s s of λ where
    end          → ⊤
    (msg _ _ _)  → SWf
    (branch _ _ _) → SWf × SWf
    (ν _ _)      → SWf
    (var _)      → ⊤
components (swf end _ _) = tt
components (swf (msg t b s) (msg s-contr) (msg s-closed))
    = swf s s-contr s-closed
```

```
components (swf  (branch t l r) (branch l-contr r-contr)
                 (branch l-closed r-closed))
    = swf l l-contr l-closed , swf r r-contr r-closed
components (swf (ν x s) contr@(ν _ s-contr) closed@(ν s-closed))
    = swf  (s [ x ↦ ν x s ])
           (subst-preserves-Contractive s-contr contr)
           (subst-preserves-Well-Scoped s-closed closed)
components (swf (var x) _ _) = tt

trF : ∀ {t}
  → (x : SWf)
  → (SWf → Tr t)
  → (∀ y → y <<< x → TrWhnf t)
  → TrWhnf t
trF (swf end _ _) corec rec = Tr-end
trF s@(swf (msg t b s′) _ _) corec rec
    = Tr-msg t b (corec (components s))
trF s@(swf (branch t l r) _ _) corec rec
    = Tr-branch t
        (corec (proj₁ (components s)))
        (corec (proj₂ (components s)))
trF s@(swf (ν x s′) contr@(ν _ _) (ν _)) corec rec
    = rec (components s) (ν-expand-ν-count s′ contr)
trF (swf (var x) _ (var ()))

tr = cofixWf <<<-wf trF
```

trF now follows the familiar pattern: The first branch is non-recursive, while the next two add new observations to the returned trace language and can therefore corecurse. The ν case recurses; the recursive call's second argument is a proof that $s\ [\ x \mapsto \nu\ x\ s\ ]$ has less ν constructors at the front than $\nu\ x\ s$. The last case is impossible since the input session type is closed.

As a final note, there is nothing specific to session types about this technique for turning a tree with "backreferences" into an infinite tree. Generalising from binary trees to rose trees would let us generate a coinductive representation of arbitrary graphs [35] from an inductive, finite one.

6 Limitations

The main drawback of an M-type-based approach is the use of M-types: They introduce significant conceptual overhead and require users of our library to structure their data types in a particular fashion. Using containers also adds moderate ergonomic issues because Agda's goal simplification heuristic often yields unsatisfying results, expanding too much or too little of an M-type's definition. If Agda supported MiniAgda's polarity annotations, these issues would be less severe since we could replace containers with

concrete functors.

As already mentioned in Sec. 3.3, the fixpoint equation for cofixWf is also suboptimal since it requires either the use of an additional axiom or somewhat cumbersome workarounds.

All of these issues could be avoided by specialising both cofixWf and its fixpoint equation to particular coinductive types, avoiding the need for M-types and container functors. This may be the more economical choice for users who only have to deal with few different coinductive types. It would also be possible, though not entirely straightforward, to generate such specialised cofixpoint combinators by metaprogramming.

Sized types play a central role in our development, which prevents us from porting our library to systems lacking comparable features, especially Coq [40].

7 Related Work

Bertot and Komendantskaya describe a general technique for encoding mixed recursive-corecursive functions like runs [12], which Bertot had used earlier to define a filtering function on streams [11]. Their central idea is to define two special-purpose predicates per function: one inductive predicate characterising the recursive component of a mixed recursive-corecursive function, and one coinductive predicate characterising the corecursive component. The mixed function is then defined by recursing structurally over the inductive predicate and nesting this recursion in a guarded corecursive definition. The authors give a (metatheoretical) algorithm for separating the recursive and corecursive parts of a function and constructing the predicates.

Compared to our cofixpoint combinator, this means greater effort for users, since they have to apply the algorithm to their particular problem by hand. In contrast, our combinator separates recursive and corecursive components of a function mostly automatically; users only have to provide a justification for the well-founded recursive calls. However, the predicate technique is advantageous for some partial corecursive functions, i.e. functions which are defined only for some of their possible inputs. In these cases, the admissible inputs may have to be defined via a nested inductive-coinductive predicate anyway, and recursing over the predicate is then the most straightforward solution.

Matthews [29], Di Gianantonio and Miculan [21], and Charguéraud [15] develop cofixpoint combinators based on complete ordered families of equivalences (cofes) or the closely related notion of converging equivalence relations. These require the user to supply a cofe for their coinductive data types and prove that their recursive functions respect the cofe, which we claim is significantly harder than providing a well-founded relation for the recursive component of a mixed recursive-corecursive function. It is unclear whether the cofe approach is able to accomodate more functions than our combinator, but at least all non-partial examples from the referenced papers can be defined using cofixWf.

Matthews and Charguéraud additionally use non-constructive axioms in their developments. This leads to a more ergonomical treatment of partial corecursive functions, but prevents computation with the defined functions.

Danielsson describes a trick to get around the limits of guarded corecursion using embedded languages [18]. Operations on coinductive data structures are added to these structures as additional constructors[11], so streams for example gain a ZipWith constructor representing a zipping operation. These extended data structures are then interpreted by a nested recursive/corecursive function. Danielsson's technique is orthogonal to ours since it is chiefly concerned with allowing corecursive calls guarded by arbitrary functions (rather than constructors). We use sized types for this purpose, which were not available when Danielsson developed his solution.

Capretta [14], Danielsson [19], and Abel and Chapman [4] have used the Delay, or Partiality, monad to separate the definition of a function from its termination proof. The type Delay A is coinductively defined, and observing it results in either a value of A or another value of Delay A – which means that no A need ever be returned. As such, Delay can be used to represent general recursive functions. It is then possible to give a separate proof that observing some $x : \mathsf{Delay}\ A$ yields a $y : A$, which corresponds to termination of the computation that produced x and simultaneously provides its result.

We can also use Delay to define corecursive functions which are not obviously productive. But extracting an infinite value constructed by such a function seems impossible since we are unable to obtain finite prefixes of the value – and in contrast to an inductively defined function, the computation is never intended to terminate, which would give us access to the entire value. Therefore, this technique is not applicable to the problem we address.

Blanchette, Popescu and Traytel [13] implement a framework for corecursion and coinduction in Isabelle/HOL [30] that includes special support for mixed recursive-corecursive functions. Since they operate within a logic which has no intrinsic concept of coinductive datatypes, there is not much overlap with our work.

8 Conclusion

We have developed a technique for encoding mixed recursive-corecursive functions as well as proofs by mixed recursion-corecursion, and have demonstrated its utility in two exemplary cases. Our cofixpoint combinator represents an advancement of the state of the art because where it is applicable, it is easier to use than existing solutions. At the same time, our development remains entirely constructive, though one may optionally use some conservative axioms to avoid boilerplate.

References

[1] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing strictly positive types. *Theoretical Computer Science*, 342(1):3–27, 2005.

[11] Danielsson's work predates Agda's new coinduction system which does away with constructors of coinductive data types.

[2] Andreas Abel. MiniAgda: Integrating sized and dependent types. In Ana Bove, Ekaterina Komendantskaya, and Milad Niqui, editors, *Proceedings of the Workshop on Partiality And Recursion in Interative Theorem Provers (PAR 2010)*, 2010.

[3] Andreas Abel and Thorsten Altenkirch. A predicative analysis of structural recursion. *Journal of Functional Programming*, 12(1):1–41, 2002.

[4] Andreas Abel and James Chapman. Normalization by evaluation in the delay monad: A case study for coinduction via copatterns and sized types. In Paul Levy and Neel Krishnaswami, editors, *Proceedings of the 5th Workshop on Mathematically Structured Functional Programming*, 2014.

[5] Andreas Abel and Brigitte Pientka. Wellfounded recursion with copatterns: A unified approach to termination and productivity. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 185–196, New York, 2013. ACM.

[6] Andreas Abel and Brigitte Pientka. Well-founded recursion with copatterns and sized types. *Journal of Functional Programming*, 26, 2016.

[7] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: Programming infinite structures by observations. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 27–38, New York, 2013. ACM.

[8] Peter Aczel. An introduction to inductive definitions. In Jon Barwise, editor, *Handbook of mathematical logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*, pages 739–782. North-Holland, Amsterdam, 1977.

[9] Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. Indexed containers. *Journal of Functional Programming*, 25, 2015.

[10] Antonia Balaa and Yves Bertot. Fix-point equations for well-founded recursion in type theory. In Mark Aagaard and John Harrison, editors, *Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics*, pages 1–16, Berlin/Heidelberg, 2000. Springer.

[11] Yves Bertot. Filters on coinductive streams, an application to Eratosthenes' sieve. In Paweł Urzyczyn, editor, *Proceedings of the 7th International Conference on Typed Lambda Calculi and Applications*, volume 3461 of *LNCS*, pages 102–115, Berlin/Heidelberg, 2005. Springer.

[12] Yves Bertot and Ekaterina Komendantskaya. Inductive and coinductive components of corecursive functions in Coq. *Electronic Notes in Theoretical Computer Science*, 203(5):25–47, 2008.

[13] Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel. Foundational extensible corecursion: A proof assistant perspective. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 192–204, New York, 2015. ACM.

[14] Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2), 2005.

[15] Arthur Charguéraud. The optimal fixed point combinator. In Matt Kaufmann and Lawrence C. Paulson, editors, *Proceeding of the first international conference on Interactive Theorem Proving (ITP)*, volume 6172 of *LNCS*, pages 195–210, Berlin/Heidelberg, 2010. Springer.

[16] Thierry Coquand. Infinite objects in type theory. In Henk Barendregt and Tobias Nipkow, editors, *Selected Papers of the International Workshop on Types for Proofs and Programs 1993*, pages 62–78, Berlin/Heidelberg, 1994. Springer.

[17] Haskell Brooks Curry and Robert Feys. *Combinatory Logic*. North-Holland, Amsterdam, 1958.

[18] Nils Anders Danielsson. Beating the productivity checker using embedded languages. In Ana Bove, Ekaterina Komendantskaya, and Milad Niqui, editors, *Proceedings of the Workshop on Partiality And Recursion in Interative Theorem Provers (PAR 2010)*, 2010.

[19] Nils Anders Danielsson. Operational semantics using the partiality monad. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 127–138, New York, 2012. ACM.

[20] Nicolaas Govert de Bruijn. The mathematical language AUTOMATH, its usage, and some of its extensions. In M. Laudet, D. Lacombe, L. Nolin, and M. Schützenberger, editors, *Symposium on Automatic Demonstration 1968*, pages 29–61, Berlin/Heidelberg, 1970. Springer.

[21] Pietro Di Gianantonio and Marino Miculan. A unifying approach to recursive and co-recursive definitions. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs. TYPES 2002*, pages 148–161, Berlin/Heidelberg, 2003. Springer.

[22] Simon Marlow (editor). Haskell 2010 language report. https://www.haskell.org/definition/haskell2010.pdf.

[23] Simon Gay and Vasco Thudichum Vasconcelos. Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(1):19–50, 2010.

[24] Eduardo Giménez. Codifying guarded definitions with recursive schemes. In Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Selected Papers of the International Workshop on Types for Proofs and Programs 1994*, pages 39–59, Berlin/Heidelberg, 1995. Springer.

[25] Peter Hancock and Pierre Hyvernat. Programming interfaces and basic topology. *Annals of Pure and Applied Logic*, 137(1):189–239, 2006.

[26] Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *Proceedings of the 4th International Conference on Concurrency Theory*, volume 715 of *LNCS*, pages 509–523, Berlin/Heidelberg, 1993. Springer.

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

[28] Jannis Limperg. wellfounded-coind. http://doi.org/10.5281/zenodo.573022. Version 0.0.1.

[29] John Matthews. Recursive function definition over coinductive types. In Yves Bertot, Gilles Dowek, Laurent Théry, André Hirschowitz, and Christine Paulin, editors, *Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics*, pages 73–90, Berlin/Heidelberg, 1999. Springer.

[30] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson, editors. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, Berlin/Heidelberg, 2002.

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

[32] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, September 2007.

[33] Lawrence C. Paulson. Constructing recursion operators in intuitionistic type theory. *Journal of Symbolic Computation*, 2(4):325 – 355, 1986.

[34] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, 2003.

[35] Celia Picard. *Représentation coinductive des graphes*. PhD thesis, Université Toulouse III, école Mathématiques Informatique Télécommunications, June 2012.

[36] Ben A. Sijtsma. On the productivity of recursive list definitions. *ACM Trans. Program. Lang. Syst.*, 11(4):633–649, 1989.

[37] Wouter Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, 2008.

[38] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In Costas Halatsis, Dimitrios Maritsas, George Philokyprou, and Sergios Theodoridis, editors, *Proceedings of the 6th International Conference on Parallel Architectures and Languages Europe*, volume 817 of *LNCS*, pages 398–413, Berlin/Heidelberg, 1994. Springer.

[39] The Agda team. The Agda standard library. https://github.com/agda/agda-stdlib. Version 0.13.

[40] The Coq Development Team. Reference manual. https://coq.inria.fr/refman/. Version 8.6.

[41] Benno van den Berg and Federico De Marchi. Non-well-founded trees in categories. *Annals of Pure and Applied Logic*, 146(1):40–59, 2007.

Zusammenfassung

In totalen funktionalen Programmiersprachen muss statisch sichergestellt werden, dass alle rekursiven Funktionen terminieren. Für corekursive Funktionen, die unendliche Datenstrukturen wie unendliche Listen oder Bäume erzeugen, gilt stattdessen die Anforderung der Produktivität: Jedes endliche Präfix einer solchen Datenstruktur – beispielsweise die ersten *n* Elemente einer Liste – muss sich in endlicher Zeit, das heißt in endlich vielen Evaluationsschritten, bestimmen lassen.

Um Produktivität statisch zu garantieren, verwenden die meisten totalen Sprachen guarded corecursion. Dies bedeutet, dass in einer corekursiven Funktion selbstreferentielle Aufrufe nur direkt unter dem Konstruktor eines corekursiv definierten Datentypes zulässig sind. Als Beispiel für eine Funktion, die dieser Anforderung genügt, betrachte man map über garantiert-unendliche Sequenzen (Streams).[12]

```
map :: (a -> b) -> Stream a -> Stream b
map f (x : xs) = f x : map f xs
```

Der rekursive Aufruf von map ist ein Argument des Stream-Konstruktors :, deswegen ist diese Definition zulässig.

Guarded corecursion ist allerdings problematisch für eine Klasse von Funktionen, die wir gemischt rekursiv-corekursiv nennen. Sie zeichnen sich dadurch aus, dass sie eine terminierende Rekursion in eine unendliche Corekursion integrieren. Als Beispiel betrachte man die Funktion runs, die aus einem Stream natürlicher Zahlen alle zusammenhängenden Sequenzen strikt absteigender Zahlen extrahiert. Sie lässt sich in einer partiellen funktionalen Sprache mittels folgender Hilfsfunktion definieren:

```
runs' : [Nat] -> Stream Nat -> Stream [Nat]
runs' run (x : y : xs)
    = if y < x
        then runs' (run ++ [x]) (y : xs)
        else (run ++ [x]) : runs' [] (y : xs)
```

runs' erhält als Eingabe die aktuelle strikt absteigende Sequenz und den Stream. Es testet dann, ob die ersten beiden Elemente des Streams zur gleichen Sequenz gehören. Wenn ja, wird *x* zur Sequenz hinzugefügt, dann rekurriert. Wenn nein, endet die aktuelle Sequenz mit *x* und wird dem Ausgabestream hinzugefügt, bevor runs' rekurriert. runs ist dann schlicht runs' [ ].

Diese Definition erfüllt nicht die Anforderungen der guarded corecursion. Zwar liegt der selbstreferentielle Aufruf im else-Fall unter dem Stream-Konstruktor und ist damit akzeptabel, aber im then-Fall wird die Rekursion durch nichts "bewacht".

Dennoch ist runs' produktiv, wie die folgenden Überlegung zeigt: Für einen Stream, der mit der natürlichen Zahl *n* beginnt, kann höchstens *n*-mal nacheinander der then-Fall auftreten, denn dann ist die Null erreicht und damit die strikt absteigende Sequenz zu Ende. Wenn dann gezwungenermaßen der else-Fall auftritt, wird ein Konstruktor zur Ausgabe hinzugefügt. Das erste Element der Ausgabe kann somit stets nach endlich vielen Evaluationsschritten – nämlich höchstens $n + 1$ – beobachtet werden, was runs' produktiv macht.

[12] Alle Codebeispiele in diesem Abschnitt sind in Haskell verfasst.

In dieser Arbeit entwickeln wir eine Technik, die es erlaubt, gemischt rekursiv-corekursive Funktionen wie runs' mit vergleichsweise geringem Aufwand in der totalen funktionalen Sprache Agda zu formalisieren. Dazu definieren wir den Cofixpunkt-Kombinator cofixWf, der den größten Fixpunkt eines offen rekursiven Funktionals nimmt und diesem dabei erlaubt, auf zwei verschiedene Arten zu rekurrieren: Entweder via guarded corecursion, wie im else-Fall von runs', oder via well-founded recursion, wie im then-Fall. Letzteres erfordert einen Beweis, dass der rekursive Teil von runs' – das Herunterzählen bis Null – terminiert; dieser Beweis ist mit etablierten Techniken leicht zu erbringen.

Unsere Methode ist wesentlich einfacher anzuwenden als bestehende Techniken zur Kodierung gemischt rekursiv-corekursiver Funktionen. Ihre Implementation beruht allerdings auf relativ neuen Komponenten des Agda-Typsystems, insbesondere sized types.

Im Rahmen dieser Arbeit haben wir eine Agda-Bibliothek entwickelt, die unsere Technik vollständig konstruktiv formalisiert.

Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Ort, Datum	Unterschrift