Mixing Induction and Coinduction

Nils Anders Danielsson
University of Nottingham
nad@cs.nott.ac.uk

Thorsten Altenkirch
University of Nottingham
txa@cs.nott.ac.uk

Abstract

Purely inductive definitions give rise to tree-shaped values where all branches have finite depth, and purely coinductive definitions give rise to values where all branches are potentially infinite. If this is too restrictive, then an alternative is to use mixed induction and coinduction. This technique appears to be fairly unknown. The aim of this paper is to make the technique more widely known, and to present several new applications of it, including a parser combinator library which guarantees termination of parsing, and a method for combining coinductively defined inference systems with rules like transitivity.

The developments presented in the paper have been formalised and checked in Agda, a dependently typed programming language and proof assistant.

1. Introduction

Coinduction and corecursion are useful techniques for defining and reasoning about things which are potentially infinite. These "things" include streams and other (potentially) infinite data types (Coquand 1994; Giménez 1996; Turner 2004), subtyping relations for recursive types (Brandt and Henglein 1998; Gapeyev et al. 2002), process congruences (Milner 1990), congruences for functional programs (Gordon 1999), closures (Milner and Tofte 1991), semantics for divergence of programs (Cousot and Cousot 1992; Hughes and Moran 1995; Leroy and Grall 2009; Nakata and Uustalu 2009), and other applications.

However, the use of coinduction can lead to values which are "too infinite". For instance, a non-trivial binary relation defined as a coinductive inference system cannot include the rule of transitivity, because a coinductive reading of transitivity would imply that every element is related to every other (to see this, build an infinite derivation consisting solely of uses of transitivity). As pointed out by Gapeyev et al. (2002) this is unfortunate, because without transitivity conceptually unrelated rules may have to be merged or otherwise modified, in order to ensure that transitivity can be proved as a derived property. Gapeyev et al. give the example of subtyping for records, where a dedicated rule of transitivity ensures that one can give separate rules for depth subtyping (which states that a record field type can be replaced by a subtype), width subtyping (which states that new fields can be added to a record), and permutation of record fields.

Fortunately this problem can be solved. The problem stems from a *coinductive* reading of transitivity, and it can be solved by reading the rule of transitivity *inductively*, and only using coinduction where it is necessary. In Section 4 this idea is illustrated by using mixed induction and coinduction to define a subtyping relation for recursive types. The rule which defines when a function type is a subtype of another is defined coinductively (following Brandt and Henglein (1998) and Gapeyev et al. (2002)), while the rule of transitivity is defined inductively.

It should be noted that some caution is necessary when using mixed induction and coinduction to define inference systems. Given an inductively defined inference system it is always sound to include an admissible property as a new inductive rule. However, in the presence of coinduction such an inclusion may not be sound if the admissible rule does not have a sufficiently "contractive" proof; see Section 4.5 for more details. (This problem does not affect our definition of subtyping, because we prove that the definition coincides with other definitions from the literature.)

The technique of mixing induction and coinduction is not limited to defining transitive inference systems. It is generally useful when defining sets of tree-shaped values (like the typing derivations above) where some values are guaranteed to have finite depth (typing derivations consisting solely of the rules of transitivity and reflexivity, for instance), and other values are allowed to have infinite depth.

As a second example of mixed induction and coinduction a library of parser combinators (Burge 1975; Wadler 1985; Fairbairn 1987, and many others) is defined. Parser combinators can provide an elegant and declarative method for specifying parsers. When compared with parser generators they have some advantages: it is easy to abstract over recurring grammatical patterns, and there is no need to use a separate tool just to parse something. On the other hand there are also some disadvantages: there is a risk of lack of efficiency, and parser generators can give static guarantees about termination and non-ambiguity which most parser combinator libraries fail to give. The library defined here addresses one of these points by ensuring statically that parsing will terminate. It also comes with a formal semantics.

When using parser combinators the parsers/grammars are often constructed using cyclic definitions, so it is natural to see the definitions as being partly corecursive. However, a purely coinductive reading of the choice and sequencing combinators would allow definitions like $p = p \mid p$ and $p' = p' \cdot p'$, for which it is impossible to implement parsing in a total way (in a pure setting where p and p' can only be inspected via their infinite unfoldings). As shown in Section 5 totality can be ensured by reading choice inductively, and only reading an argument of the sequencing operator coinductively when the other argument does not accept the empty string.

The examples above show how typing derivations and term languages can benefit from the use of mixed induction and coinduction. In Section 6 the technique is applied to the definition of a semantics: it is shown how a big-step semantics which handles both converging and diverging computations can be defined without duplication of rules.

Guarded corecursion (see Section 2) is used throughout the paper to define values of coinductive types. Guardedness is only an approximation of productivity, which essentially states that every finite approximation of a conceptually infinite value should be computable in a finite number of steps. The restriction to guardedness can make programs and proofs harder to define, but is easy to check mechanically, and is used in some form in several programming languages/proof assistants. Section 3 presents a method for working around the limitations of guardedness. This method is admittedly ad-hoc, but is included because it uses mixed induction and coinduction, and because the technique is used to show productivity in other parts of the paper.

The method of mixing induction and coinduction is not new; Section 7.1 lists further applications of the technique. However, the method does not seem to be well-known. For instance, several authors mention that transitivity and coinduction cannot be combined (Gapeyev et al. 2002; Levin and Pierce 2003; Colazzo and Ghelli 2005), but as mentioned above this is not generally true. The main purpose of this paper is to make this technique more well-known.

To summarise, the main contributions of this paper are as follows:

- The technique of mixing induction and coinduction is explained, and its utility is demonstrated using several examples.
- It is shown that it is possible to combine coinduction and the rule of transitivity when defining an inference system.
- It is shown how parser combinators can be implemented in such a way that termination is guaranteed.
- A new method for working around the limitations of guardedness is explained.

Furthermore all the main developments have been formalised using the experimental, dependently typed, total[1] functional programming language Agda (Norell 2007; Agda Team 2009), and the source code can (at the time of writing) be downloaded from the first author's web page.

The rest of the paper is structured as follows: Section 2 gives an introduction to coinduction, and Section 3 describes a technique for working around the limitation of guardedness. Section 4 discusses subtyping for recursive types, Section 5 describes a parser combinator library which guarantees termination, and Section 6 discusses a big-step semantics which handles both converging and diverging computations without duplication of rules. Finally Section 7 discusses related work and Section 8 concludes.

2. Coinduction

Coinduction and corecursion can be defined in several ways. For concreteness this paper uses one particular method, but the main ideas should be generally valid (in theories which are sufficiently strong). In order to show that the techniques which are explained can be used in practical formalisations and programs, to reduce the risk of errors, and to make things concrete, the definitions in this paper will be carried out in Agda, slightly modified[2] to make things simpler and aid readability.

[1] Note that the meta-theory of Agda has not been properly formalised, so take statements such as "total" with a grain of salt. However, even if Agda should turn out to be unsound the definitions and proofs in the paper are still valid constructions in intuitionistic type theory.

[2] We have omitted some type declarations for implicit arguments, simplified the definitional equality of corecursive programs, and ignored universe levels (Set_0, $Set_1 \ldots$).

This brief introduction does not attempt to explain coinduction in detail, but rather aims to give the intuition necessary to understand the rest of the paper. Coinductive definitions can be viewed as the categorical dual of inductive definitions (Hagino 1987). In the context of intuitionistic type theory coinductive types have been investigated by Mendler (1988), Coquand (1994), and Giménez (1996), and Leroy and Grall (2009) relate one type theoretic approach to coinduction with the approach based on the Knaster-Tarski fixpoint theorem.

The rest of this section illustrates the approach to coinduction taken in Agda. To start with, let us define the type of infinite streams:

```
data Stream (A : Set) : Set where
  _::_ : A → ∞ (Stream A) → Stream A
```

This states that *Stream* is a data type parametrised on the element type (or set) A, with a single constructor $\_{::}\_$ which takes two arguments ($\_{::}\_$ is an infix operator; the underscores mark the argument positions). The type function $\infty \ : \ Set \rightarrow Set$ marks an argument as being *coinductive*. Data type definitions can be seen as defining fixpoints of functors, and a data type definition $T \ = \ F \ (\infty \ T) \ T$, where F is a strictly positive functor in two arguments which does not mention T, should be read as the nested fixpoint $\nu C. \ \mu I. \ F \ C \ I$ (almost; see Section 7.1). Here $\mu X.G \ X$ and $\nu X.G \ X$ stand for the initial algebra and terminal coalgebra, respectively, of a given functor G. In the case of *Stream* we get the expected interpretation $Stream \ A \ = \ \nu C.A \times C$, modulo the presence of the named constructor $\_{::}\_$.

Programmers may prefer to think about ∞ as the suspension type constructor which is used to implement non-strictness in strict languages (Wadler et al. 1998). Just as the suspension type constructor the function ∞ comes with delay and force operators, here called $\sharp\_$ and $\flat$:

```
♯_ : ∀ {A} →     A → ∞ A
♭  : ∀ {A} → ∞ A →     A
```

($\sharp\_$ is a tightly binding prefix operator; ordinary function application binds tighter, though. Arguments in braces, {...}, are implicit, and do not need to be given explicitly as long as Agda can infer them from the context.)

Functions constructing coinductive values, like the *map* function for streams, can be defined by corecursion:

```
map : ∀ {A B} → (A → B) → Stream A → Stream B
map f (x :: xs) = f x :: ♯ map f (♭ xs)
```

Agda is a total language, so functions which construct infinite values have to be productive; even if the value being constructed is infinite it should always be possible to compute the next constructor in a finite number of steps. This is enforced by limiting corecursion to *guarded* corecursion (Coquand 1994). The definition of *map* is accepted because the corecursive call is guarded by the coinductive constructor $\sharp\_$, without any non-constructor function between the left-hand side and the corecursive call.

The *map* function is easy to define using guarded corecursion. However, sometimes guardedness is an inconvenient restriction. Fortunately guardedness is only an approximation of productivity, used because it is easy to check mechanically. When working informally, or in a formal setting where productivity is established semantically, guardedness can be replaced by more liberal principles.

The uses of $\sharp\_$ and $\flat$ in the definition of *map* can be perceived as cluttering the code. One could imagine designing a language in which these operators can be inferred automatically, based on type information. However, for the purposes of this paper it seems better to keep this information explicit.

Let us now see how coinduction can be mixed with induction. Hancock et al. (2009) define a language of stream processors, taking streams of As to streams of Bs, as the nested fixpoint $\nu C.\ \mu I.\ (A \rightarrow I) + B \times C$. This language can be represented using the following data type:

$$\begin{array}{l}
\textbf{data}\ \mathit{SP}\ (A\ B\ :\ \mathit{Set})\ :\ \mathit{Set}\ \textbf{where} \\
\quad \mathsf{get}\ :\ (A \rightarrow \mathit{SP}\ A\ B) \qquad \rightarrow \mathit{SP}\ A\ B \\
\quad \mathsf{put}\ :\ B \rightarrow \infty\ (\mathit{SP}\ A\ B) \rightarrow \mathit{SP}\ A\ B
\end{array}$$

(Note that $(A\ B\ :\ \mathit{Set})$ means that both the parameters A and B have type Set; it is not an application of A to B.) The recursive argument of get is inductive, while the recursive argument of put is coinductive. This means that a stream processor can only read a finite number of elements from the input before having to produce some output. The semantics of stream processors can be defined as follows:

$$\begin{array}{l}
\llbracket\_\rrbracket\ :\ \forall\ \{A\ B\} \rightarrow \mathit{SP}\ A\ B \rightarrow \mathit{Stream}\ A \rightarrow \mathit{Stream}\ B \\
\llbracket\ \mathsf{get}\,f \quad\ \rrbracket\ (a :: as)\ =\ \llbracket\ f\ a\ \rrbracket\ (\flat\ as) \\
\llbracket\ \mathsf{put}\ b\ sp\ \rrbracket\ as \qquad\ =\ b :: \sharp\ \llbracket\ \flat\ sp\ \rrbracket\ as
\end{array}$$

In the case of get one element from the input stream is consumed, and potentially used to guide the rest of the computation, while in the case of put one output element is produced. The definition of $\llbracket\_\rrbracket$ uses a lexicographic combination of guarded corecursion and structural recursion: in the second clause the corecursive call is guarded, while in the first clause the recursive call "preserves guardedness" (it takes place under zero coinductive constructors rather than at least one) and the stream processor argument is structurally smaller.

Let us now turn to coinductively defined *relations*. Equality of streams can be defined as follows:

$$\begin{array}{l}
\textbf{data}\ \_{\approx}\_\ \{A\ :\ \mathit{Set}\}\ :\ \mathit{Stream}\ A \rightarrow \mathit{Stream}\ A \rightarrow \mathit{Set}\ \textbf{where} \\
\quad \_::\_\ :\ (x\ :\ A)\ \{xs\ ys\ :\ \infty\ (\mathit{Stream}\ A)\} \rightarrow \\
\qquad\qquad \infty\ (\flat\ xs\ \approx\ \flat\ ys) \rightarrow x :: xs\ \approx\ x :: ys
\end{array}$$

This definition states that two streams are equal whenever their heads are (definitionally) equal and their tails are, recursively, equal. Note that this definition should be read coinductively, and that the tails are forced in the recursive argument's type. Note also the use of the dependent function space $(x\ :\ A) \rightarrow B$, where B can depend on x. Note finally that constructors can be overloaded in Agda.

Elements of coinductively defined relations can be constructed by using corecursion. As an example, let us prove the *map-iterate* property. The function *iterate* repeatedly applies a function to a seed element and collects the results in a stream, $\mathit{iterate}\ f\ x\ =\ x :: \sharp\ (f\ x :: \sharp\ (f\ (f\ x) :: \ldots))$:

$$\begin{array}{l}
\mathit{iterate}\ :\ \forall\ \{A\} \rightarrow (A \rightarrow A) \rightarrow A \rightarrow \mathit{Stream}\ A \\
\mathit{iterate}\ f\ x\ =\ x :: \sharp\ \mathit{iterate}\ f\ (f\ x)
\end{array}$$

The *map-iterate* property can then be proved by using guarded corecursion (the term *guarded coinduction* could also be used):

$$\begin{array}{l}
\mathit{map\text{-}iterate}\ :\ \forall\ \{A\}\ (f\ :\ A \rightarrow A)\ (x\ :\ A) \rightarrow \\
\qquad\qquad\qquad \mathit{map}\ f\ (\mathit{iterate}\ f\ x)\ \approx\ \mathit{iterate}\ f\ (f\ x) \\
\mathit{map\text{-}iterate}\ f\ x\ =\ f\ x :: \sharp\ \mathit{map\text{-}iterate}\ f\ (f\ x)
\end{array}$$

To see how this proof works, consider how it can be built up step by step (as in an interactive Agda session):

$$\mathit{map\text{-}iterate}\ f\ x\ =\ ?$$

The type of the *goal* ? is $\mathit{map}\ f\ (\mathit{iterate}\ f\ x)\ \approx\ \mathit{iterate}\ f\ (f\ x)$. Agda types should always be read up to normalisation, so this is equivalent to

$$\begin{array}{l}
f\ x :: \sharp\ \mathit{map}\ f\ (\flat\ (\sharp\ \mathit{iterate}\ f\ (f\ x)))\ \approx \\
f\ x :: \sharp\ \mathit{iterate}\ f\ (f\ (f\ x))\ .
\end{array}$$

(Note that normalisation does not involve evaluation under $\sharp\_$, and that $\flat\ (\sharp\ x)$ reduces to x.) This type matches the result type of the equality constructor $\_::\_$, so we can refine the goal:

$$\mathit{map\text{-}iterate}\ f\ x\ =\ f\ x :: ?$$

The new goal type is

$$\infty\ \big(\mathit{map}\ f\ (\mathit{iterate}\ f\ (f\ x))\ \approx\ \mathit{iterate}\ f\ (f\ (f\ x))\big),$$

so the proof can be finished by an application of the coinductive hypothesis under the guarding constructor $\sharp\_$:

$$\mathit{map\text{-}iterate}\ f\ x\ =\ f\ x :: \sharp\ \mathit{map\text{-}iterate}\ f\ (f\ x)$$

3. An ad-hoc method for making corecursive definitions guarded

As mentioned in Section 2 guardedness is sometimes an inconvenient restriction: there are productive programs which are not syntactically guarded. This section illustrates a technique which can be used to work around this restriction.

As an example, consider the following definition of the stream of Fibonacci numbers:

$$\begin{array}{l}
\mathit{fib}\ :\ \mathit{Stream}\ \mathbb{N} \\
\mathit{fib}\ =\ 0 :: \sharp\ \mathit{zipWith}\ \_{+}\_\ \mathit{fib}\ (1 :: \sharp\ \mathit{fib})
\end{array}$$

The definition uses the function *zipWith*, which combines two streams:

$$\begin{array}{l}
\mathit{zipWith}\ :\ \forall\ \{A\ B\ C\} \rightarrow (A \rightarrow B \rightarrow C) \rightarrow \\
\qquad\qquad\quad \mathit{Stream}\ A \rightarrow \mathit{Stream}\ B \rightarrow \mathit{Stream}\ C \\
\mathit{zipWith}\ f\ (x :: xs)\ (y :: ys)\ =\ f\ x\ y :: \sharp\ \mathit{zipWith}\ f\ (\flat\ xs)\ (\flat\ ys)
\end{array}$$

While the definition of *fib* is productive, it is not guarded, because the function *zipWith* is not a constructor. If *zipWith* were a constructor the definition would be guarded, though, and this presents a way out: we can define a problem-specific language which includes *zipWith* as a constructor, and then define an interpreter for the language by using guarded corecursion.

A simple language of stream programs can be defined as follows:

$$\begin{array}{l}
\textbf{data}\ \mathit{Stream}_{\mathrm{P}}\ :\ \mathit{Set} \rightarrow \mathit{Set}\ \textbf{where} \\
\quad \_::\_ \qquad\ :\ \forall\ \{A\} \rightarrow A \rightarrow \infty\ (\mathit{Stream}_{\mathrm{P}}\ A) \rightarrow \mathit{Stream}_{\mathrm{P}}\ A \\
\quad \mathsf{zipWith}\ :\ \forall\ \{A\ B\ C\} \rightarrow (A \rightarrow B \rightarrow C) \rightarrow \\
\qquad\qquad\qquad \mathit{Stream}_{\mathrm{P}}\ A \rightarrow \mathit{Stream}_{\mathrm{P}}\ B \rightarrow \mathit{Stream}_{\mathrm{P}}\ C
\end{array}$$

Note that the stream program argument of $\_::\_$ is coinductive, while the arguments of zipWith are inductive; a stream program consisting of an infinitely deep application of zipWiths would not be productive.

Stream programs will be turned into streams in two steps. First a kind of weak head normal form (WHNF) for stream programs is computed recursively, and then the resulting stream is computed corecursively. The WHNFs are defined in the following way:

$$\begin{array}{l}
\textbf{data}\ \mathit{Stream}_{\mathrm{W}}\ :\ \mathit{Set} \rightarrow \mathit{Set}\ \textbf{where} \\
\quad \_::\_\ :\ \forall\ \{A\} \rightarrow A \rightarrow \mathit{Stream}_{\mathrm{P}}\ A \rightarrow \mathit{Stream}_{\mathrm{W}}\ A
\end{array}$$

Note that the stream argument to $\_::\_$ is a ("suspended") program, not a WHNF. The function *whnf* which computes WHNFs can be defined by structural recursion:

```
whnf : ∀ {A} → Stream_P A → Stream_W A
whnf (x :: xs)              = x :: ♭ xs
whnf (zipWith f xs ys) with whnf xs | whnf ys
whnf (zipWith f xs ys) |     x :: xs′  | y :: ys′ =
  f x y :: zipWith f xs′ ys′
```

(The **with** construct is used to pattern match on the results of intermediate computations.) WHNFs can then be turned into streams corecursively:

```
mutual
  ⟦_⟧_W : ∀ {A} → Stream_W A → Stream A
  ⟦ x :: xs ⟧_W = x :: ♯ ⟦ xs ⟧_P
  ⟦_⟧_P : ∀ {A} → Stream_P A → Stream A
  ⟦ xs ⟧_P = ⟦ whnf xs ⟧_W
```

Note that this definition is guarded. (Agda accepts definitions like this one even though it is split up over two mutually defined functions.) Note also that this method can turn problems with productivity into problems with termination: if the definition of *zipWith* had not been productive the definition of *whnf* would not have been terminating.

Given the language above we can now define the stream of Fibonacci numbers using guarded corecursion:

```
fib_P : Stream_P ℕ
fib_P = 0 :: ♯ zipWith _+_ fib_P (1 :: ♯ fib_P)
fib : Stream ℕ
fib = ⟦ fib_P ⟧_P
```

One can prove that this definition satisfies the original equation for *fib* by first proving coinductively that $⟦\_⟧_P$ is homomorphic with respect to *zipWith*/zipWith:

```
zipWith-hom : ∀ {A B C} (f : A → B → C) xs ys →
   ⟦ zipWith f xs ys ⟧_P ≈ zipWith f ⟦ xs ⟧_P ⟦ ys ⟧_P
fib-correct : fib ≈ 0 :: ♯ zipWith _+_ fib (1 :: ♯ fib)
```

(The proofs are omitted.)

As stated in the introduction the method presented in this section is ad-hoc: when a new problem is encountered a new language such as *Stream*$_P$ may have to be defined. However, when working in a language which requires corecursion to be guarded we have found the method to be useful, and it readily generalises to other types than streams, and to developments using several types. In fact, the method is probably more useful for defining proofs (for instance subtyping proofs; see Sections 4.3 and 4.4) than for defining ordinary programs, because when defining a proof the computational overhead of the coding is usually irrelevant.

4. Subtyping for recursive types

This section shows how a coinductively defined inference system can be combined with the rule of transitivity. The technique is general, but illustrated in the context in which the problem was raised: subtyping for recursive types (Gapeyev et al. 2002).

4.1 Syntax

Brandt and Henglein (1998) define the following language of recursive types:

$$\sigma, \tau ::= \bot \mid \top \mid X \mid \sigma \multimap \tau \mid \mu X.\, \sigma \multimap \tau$$

Here $\bot$ and $\top$ are the least and greatest types, respectively, X is a variable, $\sigma \multimap \tau$ is a function type, and $\mu X.\, \sigma \multimap \tau$ is a fixpoint, with bound variable X. (The body of the fixpoint is required to be a function type, so types like $\mu X.X$ are ruled out.) The intention is that a fixpoint $\mu X.\sigma \multimap \tau$ should be equivalent to its unfolding $(\sigma \multimap \tau)[X := \mu X.\, \sigma \multimap \tau]$.

The language above can be represented in Agda as follows:

```
data Ty (n : ℕ) : Set where
  ⊥     : Ty n
  ⊤     : Ty n
  var   : Fin n                    → Ty n
  _⊸_   : Ty n       → Ty n        → Ty n
  μ_⊸_  : Ty (1 + n) → Ty (1 + n)  → Ty n
```

Here variables are represented using de Bruijn indices: *Ty n* represents types with at most *n* free variables, and *Fin n* is a type representing the first *n* natural numbers. Substitution can also be defined; $\sigma\ [\ \tau\ ]$ is the capture-avoiding substitution of τ for variable 0 in σ:

```
_[_] : ∀ {n} → Ty (1 + n) → Ty n → Ty n
```

The following function unfolds a fixpoint one step:

```
unfold⟨μ_⊸_⟩ : ∀ {n} → Ty (1 + n) → Ty (1 + n) → Ty n
unfold⟨μ σ ⊸ τ ⟩ = (σ ⊸ τ) [ μ σ ⊸ τ ]
```

(Note that _[_] and *unfold*⟨μ_⊸_⟩ are both mixfix operators which take two arguments.)

4.2 Subtyping via trees

A natural definition of subtyping goes via subtyping for potentially infinite trees (Gapeyev et al. 2002):

```
data Tree (n : ℕ) : Set where
  ⊥    : Tree n
  ⊤    : Tree n
  var  : Fin n                    → Tree n
  _⊸_  : ∞ (Tree n) → ∞ (Tree n)  → Tree n
```

The subtyping relation for trees can be given coinductively as follows:

```
data _≤_Tree_ : Tree n → Tree n → Set where
  ⊥    : ⊥ ≤_Tree τ
  ⊤    : σ ≤_Tree ⊤
  var  : var x ≤_Tree var x
  _⊸_  : ∞ (♭ τ₁ ≤_Tree ♭ σ₁) → ∞ (♭ σ₂ ≤_Tree ♭ τ₂) →
         σ₁ ⊸ σ₂ ≤_Tree τ₁ ⊸ τ₂
```

Note the contravariant treatment of the codomain of the function space. (Note also that the declarations of bound variables like n and τ have been omitted to reduce clutter.)

The semantics of a recursive type can be given in terms of its unfolding as a potentially infinite tree:

```
⟦_⟧ : ∀ {n} → Ty n → Tree n
⟦ ⊥ ⟧         = ⊥
⟦ ⊤ ⟧         = ⊤
⟦ var x ⟧     = var x
⟦   σ ⊸ τ ⟧   = ♯ ⟦ σ ⟧         ⊸ ♯ ⟦ τ ⟧
⟦ μ σ ⊸ τ ⟧   = ♯ ⟦ σ [ χ ] ⟧ ⊸ ♯ ⟦ τ [ χ ] ⟧
                where χ = μ σ ⊸ τ
```

The subtyping relation for types can then be defined by combining $\_{\leqslant_{\text{Tree}}}\_$ and $⟦\_⟧$:

```
_≤_Type_ : ∀ {n} → Ty n → Ty n → Set
σ ≤_Type τ = ⟦ σ ⟧ ≤_Tree ⟦ τ ⟧
```

Amadio and Cardelli (1993) also define subtyping for recursive types by going via potentially infinite trees, but they define the subtyping relation *inductively* on finite trees, and define that an infinite tree σ is a subtype of another tree τ when every finite approximation (of a certain kind) of σ is a subtype of the corresponding approximation of τ. It is easy to show that this definition, as adapted by Brandt and Henglein (1998), is equivalent to the one given above. One direction of the proof uses induction on the depth of approximation, and the other constructs elements of $\sigma \leqslant_{\text{Tree}} \tau$ corecursively (see the code which accompanies the paper).

4.3 Subtyping using mixed induction and coinduction

Subtyping can also be defined directly, without going via trees. The following definition is inspired by Brandt and Henglein (1998), see Section 4.4:

$$\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \_\leqslant\_ \ :\ \mathit{Ty}\ n \to \mathit{Ty}\ n \to \mathit{Set}\ \textbf{where}} \\
\quad \bot & : & \bot \leqslant \tau \\
\quad \top & : & \sigma \leqslant \top \\
\quad \_\rightarrowtriangle\_ & : & \infty\ (\tau_1 \leqslant \sigma_1) \to \infty\ (\sigma_2 \leqslant \tau_2) \to \\
 & & \sigma_1 \rightarrowtriangle \sigma_2 \leqslant \tau_1 \rightarrowtriangle \tau_2 \\
\quad \mathsf{unfold} & : & \mu\ \tau_1 \rightarrowtriangle \tau_2 \leqslant \mathit{unfold}\langle \mu\ \tau_1 \rightarrowtriangle \tau_2\ \rangle \\
\quad \mathsf{fold} & : & \mathit{unfold}\langle \mu\ \tau_1 \rightarrowtriangle \tau_2\ \rangle \leqslant \mu\ \tau_1 \rightarrowtriangle \tau_2 \\
\quad \mathsf{refl} & : & \tau \leqslant \tau \\
\quad \mathsf{trans} & : & \tau_1 \leqslant \tau_2 \to \tau_2 \leqslant \tau_3 \to \tau_1 \leqslant \tau_3
\end{array}$$

Note that the structural rules ($\bot$, $\top$, $\_\rightarrowtriangle\_$) are defined coinductively, while the other rules, most importantly trans, are defined inductively. Hence, assuming that the relation expresses the right thing, this definition shows that coinduction and the rule of transitivity can be combined.

Showing that the relation expresses the right thing, i.e. that it is equivalent to the one given in Section 4.2 (and thus equivalent to Amadio and Cardelli's), is straightforward:

$$\begin{array}{lll}
\mathit{complete} & : & \sigma \leqslant_{\text{Type}} \tau \to \sigma \leqslant \tau \\
\mathit{sound} & : & \sigma \leqslant \tau \to \sigma \leqslant_{\text{Type}} \tau
\end{array}$$

Completeness can be proved by a simple application of guarded corecursion. The soundness proof is a little more tricky. Assume that we have proved that $\_\leqslant_{\text{Type}}\_$ is transitive:

$$\mathit{trans}_{\text{Type}}\ :\ \tau_1 \leqslant_{\text{Type}} \tau_2 \to \tau_2 \leqslant_{\text{Type}} \tau_3 \to \tau_1 \leqslant_{\text{Type}} \tau_3$$

One might be tempted to prove soundness by a simple application of lexicographic guarded corecursion and structural recursion, the technique used to define $[\![\_]\!]$ in Section 2, but this fails. Consider the case for trans:

$$\begin{array}{l}
\mathit{sound}\ (\mathsf{trans}\ \tau_1{\leqslant}\tau_2\ \tau_2{\leqslant}\tau_3) \ = \\
\quad \mathit{trans}_{\text{Type}}\ (\mathit{sound}\ \tau_1{\leqslant}\tau_2)\ (\mathit{sound}\ \tau_2{\leqslant}\tau_3)
\end{array}$$

(Note that $\tau_1{\leqslant}\tau_2$ and $\tau_2{\leqslant}\tau_3$ are variable names.) Here the recursive calls are on smaller arguments, but $\mathit{trans}_{\text{Type}}$ is not a constructor, so guardedness is not preserved. If $\mathit{trans}_{\text{Type}}$ were a constructor the proof would be accepted, though; the proof can be rescued by an application of the technique presented in Section 3.

4.4 Inductive axiomatisation of subtyping

Brandt and Henglein (1998) define subtyping essentially as in Section 4.3, but instead of using mixed induction and coinduction they encode coinduction inductively. Their subtyping relation is ternary: $A \vdash \sigma \leqslant \tau$ means that σ is a subtype of τ given the assumptions in A. An assumption (a hypothesis) is simply a pair of types:

$$\begin{array}{l}
\textbf{data}\ \mathit{Hyp}\ (n\ :\ \mathbb{N})\ :\ \mathit{Set}\ \textbf{where} \\
\quad \_,\_\ :\ \mathit{Ty}\ n \to \mathit{Ty}\ n \to \mathit{Hyp}\ n
\end{array}$$

The subtyping relation is defined as follows:

$$\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \_\vdash\_\leqslant\_\ (A\ :\ \mathit{List}\ (\mathit{Hyp}\ n))\ :} \\
\multicolumn{3}{l}{\qquad\qquad\qquad \mathit{Ty}\ n \to \mathit{Ty}\ n \to \mathit{Set}\ \textbf{where}} \\
\quad \bot & : & A \vdash \bot \leqslant \tau \\
\quad \top & : & A \vdash \sigma \leqslant \top \\
\quad \_\rightarrowtriangle\_ & : & \textbf{let}\ H\ =\ (\sigma_1 \rightarrowtriangle \sigma_2,\ \tau_1 \rightarrowtriangle \tau_2)\ \textbf{in} \\
 & & H :: A \vdash \tau_1 \leqslant \sigma_1 \to H :: A \vdash \sigma_2 \leqslant \tau_2 \to \\
 & & A \vdash \sigma_1 \rightarrowtriangle \sigma_2 \leqslant \tau_1 \rightarrowtriangle \tau_2 \\
\quad \mathsf{unfold} & : & A \vdash \mu\ \tau_1 \rightarrowtriangle \tau_2 \leqslant \mathit{unfold}\langle \mu\ \tau_1 \rightarrowtriangle \tau_2\ \rangle \\
\quad \mathsf{fold} & : & A \vdash \mathit{unfold}\langle \mu\ \tau_1 \rightarrowtriangle \tau_2\ \rangle \leqslant \mu\ \tau_1 \rightarrowtriangle \tau_2 \\
\quad \mathsf{refl} & : & A \vdash \tau \leqslant \tau \\
\quad \mathsf{trans} & : & A \vdash \tau_1 \leqslant \tau_2 \to A \vdash \tau_2 \leqslant \tau_3 \to A \vdash \tau_1 \leqslant \tau_3 \\
\quad \mathsf{hyp} & : & (\sigma,\ \tau) \in A \to A \vdash \sigma \leqslant \tau
\end{array}$$

Note that coinduction is encoded in the $\_\rightarrowtriangle\_$ rule by inclusion of the consequent in the lists of assumptions of the antecedents.

Brandt and Henglein prove that their relation (with an empty list of assumptions) is equivalent to Amadio and Cardelli's. Their proof is considerably more complicated than the proof outlined above which shows that $\_\leqslant\_$ is equivalent to Amadio and Cardelli's definition, but as part of the proof they show that subtyping is decidable. By composing the two equivalence proofs we get that subtyping as defined in Section 4.3 is also decidable. However, Brandt and Henglein use a classical argument to show that their algorithm terminates, so it is not entirely obvious that it can be implemented in a total, constructive type theory like Agda. Fortunately some small modifications suffice to show that subtyping is decidable in this setting; details are available in the code accompanying the paper.

It may be interesting to see how the inductive encoding of coinduction used in $\_\vdash\_\leqslant\_$ can be turned directly into the "actual" coinduction used in $\_\leqslant\_$, i.e. to see how soundness of $\_\vdash\_\leqslant\_$ can be proved: it can be done using a cyclic proof. To state soundness the type *All* is used; *All P xs* means that all elements in *xs* satisfy *P*:

$$\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \mathit{All}\ (P\ :\ A \to \mathit{Set})\ :\ \mathit{List}\ A \to \mathit{Set}\ \textbf{where}} \\
\quad [\,] & : & \mathit{All}\ P\ [\,] \\
\quad \_::\_ & : & P\ x \to \mathit{All}\ P\ \mathit{xs} \to \mathit{All}\ P\ (x :: \mathit{xs})
\end{array}$$

The soundness proof shows that if $A \vdash \sigma \leqslant \tau$, where all pairs $(\sigma',\ \tau')$ in A satisfy $\sigma' \leqslant \tau'$, then $\sigma \leqslant \tau$:

$$\begin{array}{l}
\mathit{Valid}\ :\ (\mathit{Ty}\ n \to \mathit{Ty}\ n \to \mathit{Set}) \to \mathit{Hyp}\ n \to \mathit{Set} \\
\mathit{Valid}\ \_R\_\ (\sigma_1,\ \sigma_2)\ =\ \sigma_1\ R\ \sigma_2 \\
\mathit{sound}\ :\ \mathit{All}\ (\mathit{Valid}\ \_\leqslant\_)\ A \to A \vdash \sigma \leqslant \tau \to \sigma \leqslant \tau
\end{array}$$

The interesting cases of *sound* are the ones for hyp and $\_\rightarrowtriangle\_$. Hypotheses can simply be looked up in the list of valid assumptions (using $\mathit{lookup}\ :\ \mathit{All}\ P\ \mathit{xs} \to x \in \mathit{xs} \to P\ x$), and function spaces can be handled by defining a cyclic proof:

$$\begin{array}{lll}
\mathit{sound}\ \mathit{valid}\ (\mathsf{hyp}\ h) & = & \mathit{lookup}\ \mathit{valid}\ h \\
\mathit{sound}\ \mathit{valid}\ (\tau_1{\leqslant}\sigma_1 \rightarrowtriangle \sigma_2{\leqslant}\tau_2) & = & \mathit{proof} \\
\multicolumn{3}{l}{\quad \textbf{where}\ \mathit{proof}\ =\ \sharp\ \mathit{sound}\ (\mathit{proof} :: \mathit{valid})\ \tau_1{\leqslant}\sigma_1 \rightarrowtriangle} \\
\multicolumn{3}{l}{\qquad\qquad\qquad\quad \sharp\ \mathit{sound}\ (\mathit{proof} :: \mathit{valid})\ \sigma_2{\leqslant}\tau_2}
\end{array}$$

Note that the corecursive calls extend the list of valid assumptions with the proof currently being defined. The definition of *proof* is not guarded, but it would be if *sound* were a constructor. Yet again this problem can be solved by using the technique from Section 3. The program and WHNF types can be defined mutually as follows:

$$\begin{array}{lll}
\multicolumn{3}{l}{\textbf{mutual}} \\
\multicolumn{3}{l}{\quad \textbf{data}\ \_\leqslant_{\text{P}}\_\ :\ \mathit{Ty}\ n \to \mathit{Ty}\ n \to \mathit{Set}\ \textbf{where}} \\
\qquad \mathsf{sound} & : & \mathit{All}\ (\mathit{Valid}\ \_\leqslant_{\text{W}}\_)\ A \to A \vdash \sigma \leqslant \tau \to \sigma \leqslant_{\text{P}} \tau \\
\multicolumn{3}{l}{\quad \textbf{data}\ \_\leqslant_{\text{W}}\_\ :\ \mathit{Ty}\ n \to \mathit{Ty}\ n \to \mathit{Set}\ \textbf{where}} \\
\qquad \mathsf{done} & : & \sigma \leqslant \tau \to \sigma \leqslant_{\text{W}} \tau \\
\qquad \_\rightarrowtriangle\_ & : & \infty\ (\tau_1 \leqslant_{\text{P}} \sigma_1) \to \infty\ (\sigma_2 \leqslant_{\text{P}} \tau_2) \to \\
 & & \sigma_1 \rightarrowtriangle \sigma_2 \leqslant_{\text{W}} \tau_1 \rightarrowtriangle \tau_2 \\
\qquad \mathsf{trans} & : & \tau_1 \leqslant_{\text{W}} \tau_2 \to \tau_2 \leqslant_{\text{W}} \tau_3 \to \tau_1 \leqslant_{\text{W}} \tau_3
\end{array}$$

The cases of *sound* listed above are now part of the *whnf* function:

```
whnf : σ ≤P τ → σ ≤W τ
...
whnf (sound valid (hyp h))                = lookup valid h
whnf (sound valid (τ1≤σ1 ⟶ σ2≤τ2)) = proof
  where proof = ♯ sound (proof ∷ valid) τ1≤σ1 ⟶
                ♯ sound (proof ∷ valid) σ2≤τ2
```

Note that *proof* is now guarded.

We have not found a proof for completeness of $\_\vdash\_\leqslant\_$ with respect to $\_\leqslant\_$ which does not use a decision procedure for subtyping. This is not entirely surprising: such a completeness proof must turn a potentially infinite proof of $\sigma \leqslant \tau$ into a finite proof of $[\,] \vdash \sigma \leqslant \tau$, so some "trick" is necessary. With a suitably formulated decision procedure at hand the trick is simple: the decision procedure gives either a proof of $[\,] \vdash \sigma \leqslant \tau$, or a proof which shows that $\sigma \leqslant \tau$ is impossible. In the first case we are done, and in the second case a contradiction can be derived.

4.5 Postulating an admissible rule may not be sound

Given an inductively defined inference system one can add a new rule corresponding to an admissible property without changing the set of derivable properties. It is easy to prove this statement by defining functions which translate between the two inference systems. Translating derivations from the smaller to the larger inference system is trivial. When translating in the other direction one can replace all occurrences of the new rule with instances of the proof of admissibility; this process can be implemented using recursion over the structure of the input derivation.

However, when coinduction comes into the picture this property is no longer true (de Vries 2009). The proof given above breaks down because there is no guarantee that the second translation can be implemented in a productive way. The problem is that, although the admissible rule has a proof, this proof may not be sufficiently "contractive" (for instance, the proof may replace coinductive rules in the input derivation with inductive rules in the output derivation).

The following example illustrates the problem. Capretta (2005) defines the so-called partiality monad as follows:

```
data _ν (A : Set) : Set where
  return : A         → A ν
  step   : ∞ (A ν) → A ν
```

The constructor return returns a result, and step postpones a computation. Two computations are deemed equivalent if they yield the same result, perhaps with finite differences in delay:

```
data _≈_ : A ν → A ν → Set where
  return :                          return v ≈ return v
  step   : ∞ (♭ x ≈ ♭ y) → step x   ≈ step y
  stepʳ  :     x ≈ ♭ y  → x        ≈ step y
  stepˡ  :   ♭ x ≈  y   → step x   ≈ y
```

It can be proved that this equality is an equivalence relation, and that it is not trivial (assuming that A is inhabited). Let us now add transitivity as an inductive rule:

```
...
trans : x ≈ y → y ≈ z → x ≈ z
```

Given this new constructor we can prove, using guarded coinduction, that the relation is trivial:

```
trivial : (x y : A ν) → x ≈ y
trivial x y =
  trans (stepʳ (refl x))
        (trans (step (♯ trivial x y)) (stepˡ (refl y)))
```

(Here *refl* is a proof of reflexivity.)

This problem does not affect the definition of subtyping given above, which has been proved to be equivalent to other definitions from the literature. However, it means that one should exercise caution when defining relations using mixed induction and coinduction, and avoid relying on results or intuitions which are only valid in the inductive case.

5. Total parser combinators

This section describes how parser combinators which guarantee termination can be implemented. For simplicity only recognition is discussed. The accompanying code uses the ideas presented below to implement monadic parser combinators which return proper results, not just yes or no.

5.1 Parsers

The aim is to define a data type with the following basic combinators as constructors: ∅, which always fails; ε, which accepts the empty string; tok, which accepts a given token; $\_|\_$, symmetric choice; and $\_\cdot\_$, sequencing. Another combinator is added below.

Let us first consider whether the combinators should be read inductively or coinductively. An infinite choice cannot be decided, so we choose to read choices inductively. The situation is a bit trickier for sequencing. Consider definitions like $p = p \cdot p'$ or $p = p' \cdot p$. If p' accepts the empty string, then it seems hard to make any progress with these definitions. However, if p' is guaranteed not to accept the empty string then parsing can be implemented by a structurally recursive top-down approach, because the (finite) input string will be shorter when we descend on the recursive occurrence of p. To make use of this idea we will indicate whether or not a parser is nullable (accepts the empty string) in its type, and the left (right) argument of $\_\cdot\_$ will be coinductive iff the right (left) argument is not nullable. This "conditional coinduction" is encoded using the following data type:

```
data ∞? (A : Set) : Bool → Set where
  ⟨_⟩  :     A → ∞? A true
  ⟪_⟫  : ∞ A → ∞? A false
```

For convenience the index true is used for the inductive case, and false for the coinductive case. The type comes with corresponding conditional delay and force operators:

```
♯? : ∀ {b A} → A → ∞? A b
♯? {true}  x = ⟨   x ⟩
♯? {false} x = ⟪ ♯ x ⟫

♭? : ∀ {b A} → ∞? A b → A
♭? ⟨ x ⟩ =   x
♭? ⟪ x ⟫ = ♭ x
```

Based on the observations and definitions above the type P of parsers (recognisers) can now be defined:

```
data P : Bool → Set where
  ∅   : P false
  ε   : P true
  tok : Tok → P false
  _|_ : ∀ {n1 n2} → P n1 → P n2 → P (n1 ∨ n2)
  _·_ : ∀ {n1 n2} →
          ∞? (P n1) n2 → ∞? (P n2) n1 → P (n1 ∧ n2)
```

(*Tok* is the token type.) Note how the conditional coinduction operator is used to express whether the two arguments to the sequencing operator should be read inductively or coinductively. Note also that ∅ and tok do not accept the empty string, while ε does. A choice $p_1 \mid p_2$ is nullable if either p_1 or p_2 is, and a sequence $p_1 \cdot p_2$ is nullable if both p_1 and p_2 are.

5.2 Examples

Using the definition above it is easy to define parsers which are both left and right recursive:

$$\begin{aligned}&\mathit{leftRight} \;:\; P\ \mathsf{false}\\ &\mathit{leftRight} \;=\; \langle\!\langle\, \sharp\ \mathit{leftRight}\,\rangle\!\rangle \cdot \langle\!\langle\, \sharp\ \mathit{leftRight}\,\rangle\!\rangle\end{aligned}$$

Using the semantics in Section 5.3 it is easy to show that *leftRight* does not accept any string (so $\emptyset$ could be a derived combinator).

As a more useful example of how the combinators above can be used to define derived parsers, consider the following definition of the Kleene star:

$$\begin{aligned}&\mathbf{mutual}\\ &\quad \_\star \;:\; P\ \mathsf{false} \to P\ \mathsf{true}\\ &\quad p\,\star \;=\; \varepsilon \mid p\,+\\ &\quad \_+ \;:\; P\ \mathsf{false} \to P\ \mathsf{false}\\ &\quad p\,+ \;=\; \langle\, p\,\rangle \cdot \langle\!\langle\, \sharp\ (p\,\star)\,\rangle\!\rangle\end{aligned}$$

The recogniser $p\,\star$ accepts zero or more occurrences of whatever p accepts, and $p\,+$ accepts one or more occurrences; this is easy to prove using the semantics in Section 5.3. Note that this definition is guarded, and hence productive. Note also that p must not accept the empty string, because if it did, then the right hand side of $p\,+$ would have to be written $\langle\, p\,\rangle \cdot \langle\, p\,\star\,\rangle$, which would make the definition unproductive.

One might argue that the types are a bit restrictive here: why should the argument parser not be allowed to be nullable? In the case of recognition this would make sense, but when parsers return results the situation is less clear. In that case a natural generalisation of the parser $p\,\star$ returns a list of all the results of the parser p, and if p is nullable this list could be infinitely long (and highly ambiguous). One solution to this problem is to use a definition of $\_\star$ which returns a list of all results corresponding to *nonempty* matches. However, this behaviour might surprise a user of the parser combinator library who thought that $\_\star$ returned *all* results. We think it is better to provide the user with a combinator $\mathsf{nonempty}$ such that $\mathsf{nonempty}\ p$ is a variant of p which does not accept the empty string:

$$\mathsf{nonempty} \;:\; \forall\,\{n\} \to P\ n \to P\ \mathsf{false}$$

The user can then make an explicit choice to throw away certain results: $(\mathsf{nonempty}\ p)\,\star$.

The example above is very small; larger examples can also be constructed. For instance, Danielsson and Norell (2009) construct mixfix operator grammars using a parser combinator library which is based on the ideas described here.

5.3 Semantics

The semantics of the parsers is defined inductively. The type $s \in p$ is inhabited iff the token string s is a member of the language defined by p:

$$\mathbf{data}\ \_\!\in\!\_ \;:\; \mathit{List\ Tok} \to P\ n \to \mathit{Set}\ \mathbf{where}$$

No string is a member of the language defined by $\emptyset$, so there is no constructor for it in $\_\!\in\!\_$. The empty string is recognised by ε:

$$\varepsilon \;:\; [\,] \in \varepsilon$$

The singleton t is recognised by $\mathsf{tok}\ t$:

$$\mathsf{tok} \;:\; [\,t\,] \in \mathsf{tok}\ t$$

If s is recognised by p_1, then it is also recognised by $p_1 \mid p_2$, and similarly for p_2:

$$\begin{aligned}&|^{\mathrm{l}} \;:\; s \in p_1 \to s \in p_1 \mid p_2\\ &|^{\mathrm{r}} \;:\; s \in p_2 \to s \in p_1 \mid p_2\end{aligned}$$

If s_1 is recognised by p_1 (suitably forced), and s_2 is recognised by p_2 (suitably forced), then the concatenation of s_1 and s_2 is recognised by $p_1 \cdot p_2$:

$$\_\cdot\_ \;:\; s_1 \in \flat^{?}\ p_1 \to s_2 \in \flat^{?}\ p_2 \to s_1 \mathbin{+\!\!+} s_2 \in p_1 \cdot p_2$$

Finally, if a nonempty string is recognised by p, then it is also recognised by $\mathsf{nonempty}\ p$:

$$\mathsf{nonempty} \;:\; t :: s \in p \to t :: s \in \mathsf{nonempty}\ p$$

It is easy to show that the semantics and the nullability index agree: if $p \;:\; P\ n$, then $[\,] \in p$ iff n is equal to true (both directions can be proved by structural induction). Given this result it is easy to decide whether or not $[\,] \in p$; it suffices to inspect the index:

$$\mathit{nullable?} \;:\; \forall\,\{n\}\ (p \;:\; P\ n) \to \mathit{Dec}\ ([\,] \in p)$$

The type $\mathit{Dec}\ P$ states that P is decidable; an element of $\mathit{Dec}\ P$ is either a proof of P or a proof showing that P is impossible:

$$\begin{aligned}&\mathbf{data}\ \mathit{Dec}\ (P \;:\; \mathit{Set}) \;:\; \mathit{Set}\ \mathbf{where}\\ &\quad \mathsf{yes} \;:\; \,P \to \mathit{Dec}\ P\\ &\quad \mathsf{no} \;\;:\; \neg\,P \to \mathit{Dec}\ P\end{aligned}$$

Here logical negation is represented as a function into the empty type $\bot$:

$$\begin{aligned}&\neg\_ \;:\; \mathit{Set} \to \mathit{Set}\\ &\neg\,P \;=\; P \to \bot\end{aligned}$$

5.4 Backend

Let us finally consider how the relation $\_\!\in\!\_$ can be decided, i.e. how a parser backend can be implemented for P. No attempt is made to make this backend efficient, the focus is on correctness.

The parser backend will be implemented using so-called derivatives (Brzozowski 1964). The derivative $\partial\ p\ t$ of p with respect to t is the "remainder" of p after p has matched the token t; it should satisfy the equivalence

$$s \in \partial\ p\ t \;\;\leftrightarrow\;\; t :: s \in p.$$

By applying the derivative operator ∂ to p and t_1, then to $\partial\ p\ t_1$ and t_2, and so on for every element in the input string s, one can decide if $s \in p$ is inhabited.

The new parser constructed by ∂ may not have the same nullability index as the original parser, so ∂ has the following type signature:

$$\partial \;:\; \forall\,\{n\}\ (p \;:\; P\ n)\ (t \;:\; \mathit{Tok}) \to P\ (\partial^{\mathrm{n}}\ p\ t)$$

The extensional behaviour of $\partial^{\mathrm{n}} \;:\; \forall\,\{n\} \to P\ n \to \mathit{Tok} \to \mathit{Bool}$ is uniquely constrained by the definition of ∂, so the definition of ∂^{n} is omitted here.

The main derivative operator is implemented as follows. The combinators $\emptyset$ and ε never accept any token, so they both have the derivative $\emptyset$:

$$\begin{aligned}&\partial\ \emptyset\ t \;=\; \emptyset\\ &\partial\ \varepsilon\ t \;=\; \emptyset\end{aligned}$$

The combinator $\mathsf{tok}\ t'$ has a non-zero derivative with respect to t iff t' and t are equal (*Tok* is assumed to come with a decision procedure $\_\!\stackrel{?}{\equiv}\!\_$ for definitional equality):

$$\begin{aligned}&\partial\ (\mathsf{tok}\ t')\ t\ \mathbf{with}\ t' \stackrel{?}{\equiv} t\\ &\partial\ (\mathsf{tok}\ t')\ t \mid \mathsf{yes}\ t'{\equiv}t \;=\; \varepsilon\\ &\partial\ (\mathsf{tok}\ t')\ t \mid \mathsf{no}\ \ t'{\not\equiv}t \;=\; \emptyset\end{aligned}$$

The derivative of $\mathsf{nonempty}\ p$ is the derivative of p:

$$\partial\ (\mathsf{nonempty}\ p)\ t \;=\; \partial\ p\ t$$

The derivative of a choice is the choice of the derivatives of its arguments:

$$\partial\ (p_1 \mid p_2)\ t \ = \ \partial\ p_1\ t \mid \partial\ p_2\ t$$

The final and most interesting case is sequencing. If p_1 is nullable, then the result is implemented as a choice, because the remainder of $p_1 \cdot \langle\ p_2\ \rangle$ could be either the remainder of p_1 followed by p_2, or the remainder of p_2:

$$\begin{array}{lcl}
\partial\ (\langle\ p_1\ \rangle \cdot \langle\ p_2\ \rangle)\ t & = & \langle\quad \partial \quad p_1\ \ t\ \ \rangle \cdot \sharp^{?}\ p_2 \ \mid\ \partial\ p_2\ t \\
\partial\ (\langle\!\langle\ p_1\ \rangle\!\rangle \cdot \langle\ p_2\ \rangle)\ t & = & \langle\!\langle\ {}^{\sharp}\ (\partial\ ({}^{\flat}\ p_1)\ t)\ \rangle\!\rangle \cdot \sharp^{?}\ p_2 \ \mid\ \partial\ p_2\ t
\end{array}$$

(Note that we may need to delay p_2, depending on the nullability index of the derivative of p_1.) If p_1 is not nullable, then the second choice above should not be included, because the first token which is accepted (if any) has to be accepted by p_1:

$$\begin{array}{lcl}
\partial\ (\langle\ p_1\ \rangle \cdot \langle\!\langle\ p_2\ \rangle\!\rangle)\ t & = & \langle\quad \partial \quad p_1\ \ t\ \ \rangle \cdot \sharp^{?}\ ({}^{\flat}\ p_2) \\
\partial\ (\langle\!\langle\ p_1\ \rangle\!\rangle \cdot \langle\!\langle\ p_2\ \rangle\!\rangle)\ t & = & \langle\!\langle\ {}^{\sharp}\ (\partial\ ({}^{\flat}\ p_1)\ t)\ \rangle\!\rangle \cdot \sharp^{?}\ ({}^{\flat}\ p_2)
\end{array}$$

The derivative operator ∂ is implemented using a lexicographic combination of guarded corecursion and structural recursion, and the index function ∂^{n} uses recursion over the *inductive* structure of the parser. Note that in the last two sequencing cases p_2 is delayed, but ∂ is not applied recursively to p_2 because p_1 is known not to accept the empty string.

It is straightforward to show that the derivative operator ∂ satisfies both directions of its specification:

$$\begin{array}{lcl}
\partial\textit{-sound} & : & s \in \partial\ p\ t \rightarrow t :: s \in p \\
\partial\textit{-complete} & : & t :: s \in p \rightarrow s \in \partial\ p\ t
\end{array}$$

These statements can be proved by induction on the structure of the semantics.

Once the derivative operator is defined and proved correct it is easy to decide $\_\!\in\!\_$:

$$\begin{array}{l}
\_\!\in?\_\ :\ \forall\ \{n\}\ (s\ :\ \mathit{List}\ \mathit{Tok})\ (p\ :\ P\ n) \rightarrow \mathit{Dec}\ (s \in p) \\
[\,]\quad\ \in?\ p = \ \mathit{nullable?}\ p \\
t :: s\ \in?\ p\ \textbf{with}\ s\ \in?\ \partial\ p\ t \\
t :: s\ \in?\ p\ \mid\ \mathsf{yes}\ s{\in}\partial pt = \ \mathsf{yes}\ (\partial\textit{-sound}\ s{\in}\partial pt) \\
t :: s\ \in?\ p\ \mid\ \mathsf{no}\ \ s{\notin}\partial pt = \ \mathsf{no}\ \ (s{\notin}\partial pt \circ \partial\textit{-complete})
\end{array}$$

In the case of the empty string the nullability index tells whether the string should be accepted or not, and otherwise $\_\!\in?\_$ is recursively applied to the derivative and the tail of the string; the specification of ∂ ensures that this is correct.

6. Big-step semantics for both converging and diverging computations

This section shows how a big-step semantics which handles both converging and diverging computations can be defined without duplication of rules.

The untyped λ-calculus with an infinite set of constants can be represented as follows:

$$\begin{array}{llll}
\multicolumn{4}{l}{\textbf{data}\ \mathit{Tm}\ (n\ :\ \mathbb{N})\ :\ \mathit{Set}\ \textbf{where}} \\
\quad \mathsf{con} & :\ \mathbb{N} & \rightarrow \mathit{Tm}\ n & \text{-- Constant.} \\
\quad \mathsf{var} & :\ \mathit{Fin}\ n & \rightarrow \mathit{Tm}\ n & \text{-- Variable.} \\
\quad \lambda & :\ \mathit{Tm}\ (1+n) & \rightarrow \mathit{Tm}\ n & \text{-- Abstraction.} \\
\quad \_\cdot\_ & :\ \mathit{Tm}\ n \rightarrow \mathit{Tm}\ n & \rightarrow \mathit{Tm}\ n & \text{-- Application.}
\end{array}$$

Here de Bruijn indices are used to represent variables, just as in Section 4.1, and capture-avoiding substitution can be defined here as well:

$$\_[\_]\ :\ \forall\ \{n\} \rightarrow \mathit{Tm}\ (1+n) \rightarrow \mathit{Tm}\ n \rightarrow \mathit{Tm}\ n$$

Leroy and Grall (2009) define a call-by-value semantics for (a slight variant of) this language by using two big-step relations: $t \Downarrow v$ means that t converges to the value v, and $t \Uparrow$ means that t diverges. The only values are constants and abstractions, and values can be turned into terms by using $\ulcorner\_\urcorner$:

$$\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \mathit{Value}\ (n\ :\ \mathbb{N})\ :\ \mathit{Set}\ \textbf{where}} \\
\quad \mathsf{con} & :\ \mathbb{N} & \rightarrow \mathit{Value}\ n \\
\quad \lambda & :\ \mathit{Tm}\ (1+n) & \rightarrow \mathit{Value}\ n
\end{array}$$

$$\begin{array}{l}
\ulcorner\_\urcorner\ :\ \forall\ \{n\} \rightarrow \mathit{Value}\ n \rightarrow \mathit{Tm}\ n \\
\ulcorner\ \mathsf{con}\ i\ \urcorner\ =\ \mathsf{con}\ i \\
\ulcorner\ \lambda\ t\quad \urcorner\ =\ \lambda\ t
\end{array}$$

Leroy and Grall define the big-step relations roughly as follows:

$$\begin{array}{l}
\textbf{data}\ \_\Downarrow\_\ :\ \mathit{Tm}\ n \rightarrow \mathit{Value}\ n \rightarrow \mathit{Set}\ \textbf{where} \\
\quad \mathsf{val}\ \ :\ \ulcorner\ v\ \urcorner \Downarrow v \\
\quad \mathsf{app}\ :\ t_1 \Downarrow \lambda\ t\ \rightarrow\ t_2 \Downarrow v\ \rightarrow\ t\ [\ \ulcorner\ v\ \urcorner\ ] \Downarrow v'\ \rightarrow \\
\qquad\qquad t_1 \cdot t_2 \Downarrow v' \\
\textbf{data}\ \_\Uparrow\ :\ \mathit{Tm}\ n \rightarrow \mathit{Set}\ \textbf{where} \\
\quad \infty\cdot\ \ :\ \infty\ (t_1 \Uparrow) \qquad\qquad\qquad\quad \rightarrow\ t_1 \cdot t_2 \Uparrow \\
\quad \cdot\infty\ \ :\ t_1 \Downarrow v\ \rightarrow\ \infty\ (t_2 \Uparrow)\ \rightarrow\ t_1 \cdot t_2 \Uparrow \\
\quad \mathsf{app}\ :\ t_1 \Downarrow \lambda\ t\ \rightarrow\ t_2 \Downarrow v\ \rightarrow\ \infty\ (t\ [\ \ulcorner\ v\ \urcorner\ ] \Uparrow)\ \rightarrow \\
\qquad\qquad t_1 \cdot t_2 \Uparrow
\end{array}$$

Note that convergence is defined inductively, while divergence is defined coinductively.

Leroy and Grall observe that the big-step definition of the semantics above contains some duplication, and note that this can be problematic when realistic languages with many rules are formalised. As an attempt to avoid this duplication they investigate a so-called coevaluation relation $\_\overset{\mathrm{co}}{\Rightarrow}\_$, defined coinductively roughly as follows:

$$\begin{array}{l}
\textbf{data}\ \_\overset{\mathrm{co}}{\Rightarrow}\_\ :\ \mathit{Tm}\ n \rightarrow \mathit{Value}\ n \rightarrow \mathit{Set}\ \textbf{where} \\
\quad \mathsf{val}\ \ :\ \ulcorner\ v\ \urcorner \overset{\mathrm{co}}{\Rightarrow}\ v \\
\quad \mathsf{app}\ :\ \infty\ (t_1 \overset{\mathrm{co}}{\Rightarrow} \lambda\ t)\ \rightarrow\ \infty\ (t_2 \overset{\mathrm{co}}{\Rightarrow} v)\ \rightarrow \\
\qquad\qquad \infty\ (t\ [\ \ulcorner\ v\ \urcorner\ ] \overset{\mathrm{co}}{\Rightarrow} v')\ \rightarrow\ t_1 \cdot t_2 \overset{\mathrm{co}}{\Rightarrow} v'
\end{array}$$

However, this relation turns out to be different from the ones defined above: there are terms which diverge according to $\_\Uparrow$, but which do not coevaluate to any value (consider the application of a diverging term to a stuck term like $\mathsf{con}\ 0 \cdot \mathsf{con}\ 1$).

In order to avoid the most glaring duplication—the two variants of the rule app—one can instead use mixed induction and coinduction. Define a semantic domain *Sem* as follows:

$$\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \mathit{Sem}\ (n\ :\ \mathbb{N})\ :\ \mathit{Set}\ \textbf{where}} \\
\quad \bot & : & \mathit{Sem}\ n \\
\quad \mathsf{val} & : & \mathit{Value}\ n \rightarrow \mathit{Sem}\ n
\end{array}$$

Here $\bot$ stands for divergence, and $\mathsf{val}\ v$ for convergence with the value v. A semantics with only one occurrence of the rule app can then be defined:

$$\begin{array}{l}
\textbf{mutual} \\
\quad \textbf{data}\ \_\Rightarrow\_\ :\ \mathit{Tm}\ n \rightarrow \mathit{Sem}\ n \rightarrow \mathit{Set}\ \textbf{where} \\
\qquad \mathsf{val}\ \ :\ \ulcorner\ v\ \urcorner \Rightarrow \mathsf{val}\ v \\
\qquad \mathsf{app}\ :\ t_1 \overset{\infty?}{\Rightarrow} \mathsf{val}\ (\lambda\ t)\ \rightarrow\ t_2 \overset{\infty?}{\Rightarrow} \mathsf{val}\ v\ \rightarrow \\
\qquad\qquad\quad t\ [\ulcorner\ v\ \urcorner] \overset{\infty?}{\Rightarrow} s\ \rightarrow\ t_1 \cdot t_2 \Rightarrow s \\
\qquad \infty\cdot\ \ :\ t_1 \overset{\infty?}{\Rightarrow} \bot \qquad\qquad\qquad\qquad\ \rightarrow\ t_1 \cdot t_2 \Rightarrow \bot \\
\qquad \cdot\infty\ \ :\ t_1 \overset{\infty?}{\Rightarrow} \mathsf{val}\ v\ \rightarrow\ t_2 \overset{\infty?}{\Rightarrow} \bot\ \rightarrow\ t_1 \cdot t_2 \Rightarrow \bot \\
\quad \_\overset{\infty?}{\Rightarrow}\_\ :\ \mathit{Tm}\ n \rightarrow \mathit{Sem}\ n \rightarrow \mathit{Set} \\
\quad t \overset{\infty?}{\Rightarrow} \bot \qquad\ = \ \infty\ (t \Rightarrow \bot) \\
\quad t \overset{\infty?}{\Rightarrow} \mathsf{val}\ v\ = \qquad t \Rightarrow \mathsf{val}\ v
\end{array}$$

The function $\_\overset{\infty?}{\Rightarrow}\_$ is used to ensure that coinduction is only used for diverging computations.

The semantics $\_\Rightarrow\_$ has more rules than $\_\overset{\mathrm{co}}{\Rightarrow}\_$, but at least it avoids the duplication of app, and it is correct; it is easy to show that $\_\Rightarrow\_$ is equivalent to the combination of $\_\Downarrow\_$ and $\_\Uparrow$:

$$\begin{array}{lll}
\textit{sound-}\Downarrow & : t \Rightarrow \mathsf{val}\ v \to t \Downarrow v & \\
\textit{complete-}\Downarrow & : t \Downarrow v & \to t \Rightarrow \mathsf{val}\ v \\
\textit{sound-}\Uparrow & : t \Rightarrow \bot & \to t \Uparrow \\
\textit{complete-}\Uparrow & : t \Uparrow & \to t \Rightarrow \bot
\end{array}$$

The first two statements can be proved using structural induction, while the latter two statements can be proved using the previous two proofs and guarded coinduction.

The soundness and completeness proofs point to a problem with the definition of $\_\Rightarrow\_$: the app rule comes with different proof principles depending on whether s is $\bot$ or not, and this can bring the duplication of app back in proofs. However, in the context of mechanised formalisations the definition of a semantics can be much more important than proofs about the semantics: an incorrect proof will (hopefully) not be accepted by the proof checker, whereas an incorrect semantics can make the entire development pointless. Avoiding code duplication and corresponding issues with overview and maintenance can hence be more important in a semantics than in proofs related to the semantics.

The problem mentioned in the previous paragraph becomes more important if the actual implementation of functions defined over the type matters. (In the case of proofs the mere existence of a function with the right type is often enough.) As an example, consider the type $\mathit{List}^{\infty?}\ r\ A$ of lists which are finite when r is μ, and potentially infinite when r is ν:

$$\begin{array}{l}
\textbf{data}\ \mathit{Rec}\ :\ \mathit{Set}\ \textbf{where} \\
\quad \mu\ :\ \mathit{Rec} \\
\quad \nu\ :\ \mathit{Rec} \\
\infty^?\ :\ \mathit{Rec} \to \mathit{Set} \to \mathit{Set} \\
\infty^?\ \mu\ A\ =\ \ A \\
\infty^?\ \nu\ A\ =\ \infty\, A \\
\textbf{data}\ \mathit{List}^{\infty?}\ (r\ :\ \mathit{Rec})\ (A\ :\ \mathit{Set})\ :\ \mathit{Set}\ \textbf{where} \\
\quad [\,]\ \ \ :\ \mathit{List}^{\infty?}\ r\ A \\
\quad \_::\_\ :\ A \to \infty^?\ r\ (\mathit{List}^{\infty?}\ r\ A) \to \mathit{List}^{\infty?}\ r\ A
\end{array}$$

There is only one cons constructor, but the following definition of *map* has two cases for cons, because in one case structural recursion is used, and in the other case guarded corecursion:

$$\begin{array}{lll}
\mathit{map} & :\ \forall\ \{r\ A\ B\} \to (A \to B) \to \mathit{List}^{\infty?}\ r\ A \to \mathit{List}^{\infty?}\ r\ B & \\
\mathit{map} & f\ [\,] & =\ [\,] \\
\mathit{map}\ \{\mu\} & f\ (x :: xs) & =\ f\ x :: \mathit{map}\ f\ xs \\
\mathit{map}\ \{\nu\} & f\ (x :: xs) & =\ f\ x :: \sharp\ \mathit{map}\ f\ (\flat\ xs)
\end{array}$$

(Note that it is possible to pattern match on implicit arguments by enclosing the patterns in braces.) It would be unfortunate if two equivalent finite lists with different *Rec* parameters were handled differently by *map*, so the definition above should be "*Rec*-parametric":

$$\begin{array}{l}
\textit{map-parametric}\ : \\
\quad \forall\ \{A\ B\}\ (f\ :\ A \to B)\ (xs\ :\ \mathit{List}^{\infty?}\ \mu\ A) \to \\
\quad \mathit{map}\ f\ (\mathit{lift}\ xs)\ \approx\ \mathit{lift}\ (\mathit{map}\ f\ xs)
\end{array}$$

Here $\mathit{lift}\ :\ \forall\ \{A\} \to \mathit{List}^{\infty?}\ \mu\ A \to \mathit{List}^{\infty?}\ \nu\ A$ is the obvious coercion and $\_\approx\_$ is a suitably defined equality.

Note that the code duplication in *map* above could have been avoided if Agda had allowed other forms of recursion. It could be useful to investigate some sort of language support for avoiding this duplication.

7. Related Work

7.1 Combining induction and coinduction

In the dependently typed core language ΠΣ Altenkirch and Oury (2008) use constructions very similar to ∞, $\sharp\_$ and $\flat$, and this work influenced the current implementation of coinductive types in Agda. Another influence came from Setzer (2009), who proposes an approach to coinductive types more directly based on the category-theoretic notion of weakly final coalgebras.

Park (1980) uses nested induction and coinduction to define what a fair merge of two potentially infinite lists is, and notes that the fair merge is a fixpoint of a certain operator, but neither the least nor the greatest fixpoint.

Barwise (1989) discusses how one can define sets and proper classes by combining greatest and least fixpoints, and mentions that such definitions are used in situation theory.

Cousot and Cousot (1992, 2009) describe so-called *bi-inductive* definitions, which generalise inductive and coinductive definitions, and give a number of examples of their use. One of their examples is a semantics for a λ-calculus which captures both terminating and non-terminating behaviours in a single definition, without duplication of rules; the definition in Section 6 was inspired by this example. Adapting the example slightly to the development in Section 6, an operator F on $\wp(\mathit{Tm}\ n \times \mathit{Sem}\ n)$ is first defined by the following inference rules (note the similarity to the rules in Section 6):

$$\ulcorner v \urcorner \Rightarrow \mathsf{val}\ v \qquad \frac{t_1 \Rightarrow \bot}{t_1 \cdot t_2 \Rightarrow \bot} \qquad \frac{t_1 \Rightarrow \mathsf{val}\ v \qquad t_2 \Rightarrow \bot}{t_1 \cdot t_2 \Rightarrow \bot}$$

$$\frac{t_1 \Rightarrow \mathsf{val}\ (\lambda\ t) \qquad t_2 \Rightarrow \mathsf{val}\ v \qquad t\ [\ulcorner v \urcorner] \Rightarrow s}{t_1 \cdot t_2 \Rightarrow s}$$

These rules should neither be read inductively nor coinductively. The semantics is instead obtained as the least fixpoint of F with respect to the order $\_\sqsubseteq\_$ defined by

$$X \sqsubseteq Y \quad = \quad X^+ \subseteq Y^+ \quad \wedge \quad X^- \supseteq Y^-,$$

where

$$Z^+ = \{\, (t, s) \in Z \mid s \neq \bot \,\} \text{ and}$$
$$Z^- = \{\, (t, s) \in Z \mid s = \bot \,\}.$$

F is not monotone with respect to $\_\sqsubseteq\_$ (which forms a complete lattice), so Cousot and Cousot give an explicit proof of the existence of a least fixpoint.

Process calculi supporting recursive definitions often include some notion of guardedness (Hoare 1985; Milner 1990; Prasad 1995). Giménez (1996) represents processes by their potentially infinite unfoldings, and uses mixed induction and coinduction to represent guardedness. Abstracting from the concrete example given by Giménez the idea is to read the arguments of the guarding constructs coinductively, and the arguments of the other constructs inductively.

HOLCF (Müller et al. 1999) supports marking individual constructor arguments as lazy. This corresponds closely to our use of ∞, but in a domain-theoretic setting with partial functions.

Levy (2006) uses nested induction and coinduction to generalise Howe's method to languages with non-well-founded syntax.

As mentioned in Section 2 Hancock et al. (2009) represent stream processors, i.e. functions from streams to streams, using nested induction and coinduction. Barthe et al. (2004) and Abel (2009) discuss total λ-calculi with nested least and greatest fixpoints and sized types, and Abel also shows how stream processors can be implemented in his calculus.

All uses of mixed induction and coinduction in previous sections can be phrased using an outer greatest fixpoint and an inner least fixpoint. Raffalli (1994) also discusses the other situation, with an outer least fixpoint and an inner greatest fixpoint. He defines two types of infinite bit streams, essentially

$$\mu O.\ \nu Z.\ 0\ :\ Z+1\ :\ O \text{ and}$$
$$\nu Z.\ \mu O.\ 0\ :\ Z+1\ :\ O$$

(read $0\ :\ Z+1\ :\ O$ as a labelled sum with constructors 0 and 1). The first one contains streams with an infinite number of zeros and a finite number of ones, while the second one contains streams with an infinite number of zeros and either a finite or an infinite number of ones.

More complicated nesting of fixpoints is also possible. Hensel and Jacobs (1997) discuss proof principles for nested induction and coinduction with arbitrary alternation of fixpoints, and include four examples: trees with finite or potentially infinite depth and finite or potentially infinite branching. Bradfield and Stirling (2007) discuss alternating fixpoints in the context of modal μ-calculus.

As an aside it is interesting to note that Agda cannot, in general, directly represent fixpoints which do not have the form $\nu X.\ \mu Y.\ F\ X\ Y$ (or an equivalent form like $\nu X.\ \mu Y.\ \mu Z.\ F\ X\ Y\ Z$). Based on the category theoretical explanation of ∞ given in Section 2 one might believe that the following definition of O should be isomorphic to $\mu O.\ \nu Z.\ 0\ :\ Z+1\ :\ O$:

```
data Z (O : Set) : Set where
  0 : ∞ (Z O) → Z O
  1 :       O  → Z O
data O : Set where
  ↓ : Z O → O
```

However, the following definition of a bit stream with an infinite number of ones is accepted by Agda's productivity checker:

```
s : O
s = ↓ (0 (♯ 1 s))
```

This is in line with the view of ∞ as a suspension type constructor. As a consequence we get that the suspension view and the category theoretical view of ∞ do not quite match. In general we have that $\mu X.\ \nu Y.\ F\ X\ Y$ is not isomorphic to $\nu Y.\ \mu X.\ F\ X\ Y$ (in the category of sets and total functions), whereas the corresponding domain theoretic expressions $\mu X.\ \mu Y.\ F\ X\ (Y_\perp)$ and $\mu Y.\ \mu X.\ F\ X\ (Y_\perp)$, obtained from the suspension view, are isomorphic (here $\perp$ stands for lifting). Note, however, that this mismatch does not affect the definitions given in previous sections, because they are not of the form exhibited by O above.

7.2 Making corecursive definitions guarded

Conor McBride (personal communication) has developed a technique for ensuring guardedness, based on the work of Hancock and Setzer (2000). The idea is to represent the right-hand sides of function definitions using a type *RHS g*, where *g* indicates whether the context is guarding or not, and to only allow corecursive calls in a guarding context.

Di Gianantonio and Miculan (2003) describe a general framework for defining values using a mixture of recursion and corecursion, based on functions which satisfy a notion of contractivity. The method seems to be quite general, and has been implemented in Coq (which is based on structural recursion and guarded corecursion).

Bertot (2005) implements a filter function for streams in Coq. An unrestricted filter function is not productive, so Bertot restricts the function's inputs using predicates of the form "always (eventually P)". The always part is defined coinductively, and the eventually part inductively (mixed induction and coinduction is not necessary).

The partiality monad was defined in Section 4.5. It is easy to define bind for this monad:

```
_>>=_ : ∀ {A} → A ν → (A → B ν) → B ν
```

Unfortunately it can be inconvenient to use this definition of bind in systems based on guarded corecursion, because _>>=_ is not a constructor. Megacz (2007) suggests (essentially) the following alternative definition:

```
data _ν (A : Set) : Set where
  return : A → A ν
  _>>=_  : ∀ {B} → ∞ (B ν) → (B → ∞ (A ν)) → A ν
```

One can note that this corresponds directly to the first step of the technique presented in Section 3. Megacz does not translate from the second to the first type, though.

Bertot and Komendantskaya (2009) describe a method for replacing corecursion with recursion. They map values of type *Stream A* to and from the isomorphic type $\mathbb{N} \to A$, and values of this type can be defined recursively. The authors state that the method is still very limited and that, as presented, it cannot handle van de Snepscheut's corecursive definition of the Hamming numbers (Dijkstra 1981), which can easily be handled using the method described in this paper.

Morris et al. (2006) use the technique of replacing functions with constructors to show *termination* rather than productivity (see Morris et al. (2007) for an explanation of the technique). They replace a partially applied recursive call (which is not necessarily structural, because it could later be applied to anything), nested inside another recursive call, with a constructor application. If this constructor application is later encountered it is handled using structural recursion.

The technique presented here also shares some traits with Reynolds' *defunctionalisation* (1972). Defunctionalisation is used to translate programs written in higher-order languages to first-order languages, and it basically amounts to representing function spaces using application-specific data types, and implementing interpreters for these data types.

7.3 Total parser combinators

There does not seem to be much prior work on *formally* verified termination for parser combinators (or other general parsing frameworks). McBride and McKinna (2002) define grammars inductively, and use types to ensure that a token is consumed before a non-terminal can be encountered, thereby ruling out left recursion and non-termination. Danielsson and Norell (2008) use similar ideas. Muad`Dib (2009) uses a monad annotated with Hoare-style pre- and post-conditions (Swierstra 2009) to define total parser combinators, including a fixpoint combinator whose type rules out left recursion by requiring the input to be shorter in recursive calls.

The parser combinators defined here cannot handle unrestricted left recursion (consider definitions like $p\ =\ p$). Lickman (1995) defines a parser combinator library which has a tailor-made fixpoint combinator (based on an idea due to Philip Wadler) which can be used to handle left recursion. Frost et al. (2008) also define a parser combinator library which can handle left recursion, this time by requiring the user to annotate non-terminals manually with a memoisation combinator. Both libraries give guarantees of termination (assuming that the informal proofs/arguments are correct, and that the libraries are used in the right way), and in both cases termination relies on a representation of parsers/grammars in which recursion is explicit.

Unlike all the approaches mentioned above the parser combinator library presented in Section 5 follows the example of most other parser combinator libraries in that it does not rely on any explicit representation of recursion (like grammars or fixpoints). Termination is instead guaranteed by using types to restrict how parsers can be defined. This appears to be the first time guaranteed termination has been achieved without any explicit representation of recursion.[3]

7.4 Big-step semantics for converging as well as diverging computations

Cousot and Cousot (2009) define several big-step semantics for an untyped λ-calculus. Every definition describes both terminating and non-terminating computations without duplication of rules.

Nakata and Uustalu (2009) define a big-step semantics for a while language. Their definition is coinductive and trace-based, and has the property that the trace can be computed (productively) for any source term, converging or diverging.

8. Conclusions

By giving a number of examples we hope to have shown that the technique of mixing induction and coinduction is generally useful, and deserves a place in the toolbox both of people formalising languages and people who are interested in ensuring totality of the programs they write.

Acknowledgments

Nils Anders Danielsson would like to thank Peter Hancock for introducing him to the technique of mixed induction and coinduction, Ulf Norell for previous joint work on total parsing, and Graham Hutton and Henrik Nilsson for feedback which improved the presentation. Thorsten Altenkirch would like to thank Nicolas Oury for joint work on ΠΣ which has had an impact on the work in this paper, and Graham Hutton for earlier joint but unpublished work on coinductive reasoning. Both authors would like to thank Conor McBride, Nicolas Oury and Anton Setzer for many discussions about coinduction which have influenced this work, and some anonymous reviewers for useful feedback. Both authors gratefully acknowledge funding from EPSRC; Nils Anders Danielsson would also like to thank the Royal Swedish Academy of Sciences' funds for financial support.

References

Andreas Abel. Mixed inductive/coinductive types and strong normalization. In *Programming Languages and Systems, 5th Asian Symposium, APLAS 2007*, volume 4807 of *LNCS*, pages 286–301, 2009.

The Agda Team. The Agda Wiki. Available at `http://wiki.portal.chalmers.se/agda/`, 2009.

Thorsten Altenkirch and Nicolas Oury. ΠΣ: A core language for dependently typed programming. Draft, 2008.

Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.

G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. *Mathematical Structures in Computer Science*, 14(1):97–141, 2004.

Jon Barwise. *The Situation in Logic*, volume 17 of *CSLI Lecture Notes*, chapter Mixed Fixed Points. Center for the Study of Language and Information, Leland Stanford Junior University, 1989.

[3] Danielsson and Norell (2009) describe a less general, more specialised version of the library which is described here. Only the interface is discussed, not the backend.

Yves Bertot. Filters on coinductive streams, an application to Eratosthenes' sieve. In *Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005*, volume 3461 of *LNCS*, pages 102–115, 2005.

Yves Bertot and Ekaterina Komendantskaya. Using structural recursion for corecursion. In *Types for Proofs and Programs, International Conference, TYPES 2008*, volume 5497 of *LNCS*, pages 220–236, 2009.

Julian Bradfield and Colin Stirling. *Handbook of Modal Logic*, chapter Modal Mu-Calculi. Elsevier, 2007.

Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticae*, 33(4):309–338, 1998.

Janusz A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, 1964.

William H. Burge. *Recursive Programming Techniques*. Addison-Wesley, 1975.

Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2):1–28, 2005.

Dario Colazzo and Giorgio Ghelli. Subtyping, recursion, and parametric polymorphism in kernel Fun. *Information and Computation*, 198:71–147, 2005.

Thierry Coquand. Infinite objects in type theory. In *Types for Proofs and Programs, International Workshop TYPES'93*, volume 806 of *LNCS*, pages 62–78, 1994.

Patrick Cousot and Radhia Cousot. Bi-inductive structural semantics. *Information and Computation*, 207(2):258–283, 2009.

Patrick Cousot and Radhia Cousot. Inductive definitions, semantics and abstract interpretations. In *POPL '92, Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 83–94, 1992.

Nils Anders Danielsson and Ulf Norell. Structurally recursive descent parsing. Presentation (given by Danielsson) at the Dependently Typed Programming workshop, Nottingham, UK, 2008.

Nils Anders Danielsson and Ulf Norell. Parsing mixfix operators. To appear in the proceedings of the 20th International Symposium on the Implementation and Application of Functional Languages (IFL 2008), 2009.

Edsko de Vries. Re: [Coq-Club] Adding (inductive) transitivity to weak bisimilarity not sound? (was: Need help with coinductive proof). Message to the Coq-Club mailing list, August 2009.

Pietro Di Gianantonio and Marino Miculan. A unifying approach to recursive and co-recursive definitions. In *Types for Proofs and Programs, International Workshop, TYPES 2002*, volume 2646 of *LNCS*, pages 148–161, 2003.

Edsger W. Dijkstra. Hamming's exercise in SASL. EWD792 (privately circulated note), 1981.

Jon Fairbairn. Making form follow function: An exercise in functional programming style. *Software: Practice and Experience*, 17(6):379–386, 1987.

Richard A. Frost, Rahmatullah Hafiz, and Paul Callaghan. Parser combinators for ambiguous left-recursive grammars. In *Practical Aspects of Declarative Languages, 10th International Symposium, PADL 2008*, volume 4902 of *LNCS*, pages 167–181, 2008.

Vladimir Gapeyev, Michael Y. Levin, and Benjamin C. Pierce. Recursive subtyping revealed. *Journal of Functional Programming*, 12(6):511–548, 2002.

Eduardo Giménez. *Un Calcul de Constructions Infinies et son Application à la Vérification de Systèmes Communicants*. PhD thesis, Ecole Normale Supérieure de Lyon, 1996.

Andrew D. Gordon. Bisimilarity as a theory of functional programming. *Theoretical Computer Science*, 228(1–2):5–47, 1999.

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

Peter Hancock and Anton Setzer. Interactive programs in dependent type theory. In *Computer Science Logic, 14th International Workshop, CSL 2000*, volume 1862 of *LNCS*, pages 317–331, 2000.

Peter Hancock, Dirk Pattinson, and Neil Ghani. Representations of stream processors using nested fixed points. *Logical Methods in Computer Science*, 5(3:9), 2009.

Ulrich Hensel and Bart Jacobs. Proof principles for datatypes with iterated recursion. In *Category Theory and Computer Science, 7th International Conference, CTCS '97*, volume 1290 of *LNCS*, pages 220–241, 1997.

C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.

John Hughes and Andrew Moran. Making choices lazily. In *FPCA '95, Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 108–119, 1995.

Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009.

Michael Y. Levin and Benjamin C. Pierce. TinkerType: a language for playing with formal systems. *Journal of Functional Programming*, 13 (2):295–316, 2003.

Paul Blain Levy. Infinitary Howe's method. In *Proceedings of the Eighth Workshop on Coalgebraic Methods in Computer Science (CMCS 2006)*, volume 164 of *ENTCS*, pages 85–104, 2006.

Paul Lickman. Parsing with fixed points. Master's thesis, University of Cambridge, 1995.

Conor McBride and James McKinna. Seeing and doing. Presentation (given by McBride) at the Workshop on Termination and Type Theory, Hindås, Sweden, 2002.

Adam Megacz. A coinductive monad for Prop-bounded recursion. In *PLPV'07, Proceedings of the 2007 workshop on Programming languages meets program verification*, pages 11–20, 2007.

Paul Francis Mendler. *Inductive Definition in Type Theory*. PhD thesis, Cornell University, 1988.

Robin Milner. Operational and algebraic semantics of concurrent processes. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. The MIT Press and Elsevier, 1990.

Robin Milner and Mads Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87(1):209–220, 1991.

Peter Morris, Thorsten Altenkirch, and Conor McBride. Exploring the regular tree types. In *Types for Proofs and Programs, International Workshop, TYPES 2004*, volume 3839 of *LNCS*, pages 252–267, 2006.

Peter Morris, Thorsten Altenkirch, and Neil Ghani. Constructing strictly positive families. In *Theory of Computing 2007, Proceedings of the Thirteenth Computing: The Australasian Theory Symposium (CATS2007)*, pages 111–121, 2007.

Muad`Dib. Strongly specified parser combinators. Post to the Muad`Dib blog, 2009.

Olaf Müller, Tobias Nipkow, David von Oheimb, and Oscar Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9(2):191–223, 1999.

Keiko Nakata and Tarmo Uustalu. Trace-based coinductive operational semantics for While; big-step and small-step, relational and functional styles. To appear in the proceedings of TPHOLs 2009, 2009.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology and Göteborg University, 2007.

David Park. On the semantics of fair parallelism. In *Abstract Software Specifications*, volume 86 of *LNCS*, pages 504–526, 1980.

K. V. S. Prasad. A calculus of broadcasting systems. *Science of Computer Programming*, 25(2–3):285–327, 1995.

Christophe Raffalli. *L'Arithmétique Fonctionnelle du Second Ordre avec Points Fixes*. PhD thesis, Université Paris VII, 1994.

John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM '72, Proceedings of the ACM annual conference*, volume 2, pages 717–740, 1972.

Anton Setzer. Coalgebras and codata in Agda. Presentation at the 3rd Wessex Theory Seminar, Bath, UK, 2009.

Wouter Swierstra. A Hoare logic for the state monad. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009*, volume 5674 of *LNCS*, pages 440–451, 2009.

D. A. Turner. Total functional programming. *Journal of Universal Computer Science*, 10(7):751–768, 2004.

Philip Wadler. How to replace failure by a list of successes; a method for exception handling, backtracking, and pattern matching in lazy functional languages. In *Functional Programming Languages and Computer Architecture*, volume 201 of *LNCS*, pages 113–128, 1985.

Philip Wadler, Walid Taha, and David MacQueen. How to add laziness to a strict language, without even being odd. In *Proceedings of the 1998 ACM SIGPLAN Workshop on ML*, 1998.