Subtyping, Declaratively
An Exercise in Mixed Induction and Coinduction

Nils Anders Danielsson and Thorsten Altenkirch

University of Nottingham

Abstract. It is natural to present subtyping for recursive types coinductively. However, Gapeyev, Levin and Pierce have noted that there is a problem with coinductive definitions of non-trivial transitive inference systems: they cannot be "declarative"—as opposed to "algorithmic" or syntax-directed—because coinductive inference systems with an explicit rule of transitivity are trivial.

We propose a solution to this problem. By using mixed induction and coinduction we define an inference system for subtyping which combines the advantages of coinduction with the convenience of an explicit rule of transitivity. The definition uses coinduction for the structural rules, and induction for the rule of transitivity. We also discuss under what conditions this technique can be used when defining other inference systems.

The developments presented in the paper have been mechanised using 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, including streams and other (potentially) infinite data types (Coquand 1994; Giménez 1996; Turner 2004), 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 subtyping relations for recursive types (Brandt and Henglein 1998; Gapeyev et al. 2002).

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.

C. Bolduc, J. Desharnais, and B. Ktari (Eds.): MPC 2010, LNCS 6120, pp. 100–118, 2010.

We propose a solution to this problem. 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. We illustrate this idea by using mixed induction and coinduction to define a subtyping relation for recursive types; such relations have been studied repeatedly in the past (Amadio and Cardelli 1993; Kozen et al. 1995; Brandt and Henglein 1998, and others). 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.

The technique of mixing induction and coinduction has been known for a long time (Park 1980; Barwise 1989; Raffalli 1994; Giménez 1996; Hensel and Jacobs 1997; Müller et al. 1999; Barthe et al. 2004; Levy 2006; Bradfield and Stirling 2007; Abel 2007; Hancock et al. 2009), but we feel that it deserves to be more well-known in the programming language community. We also believe that the approach to coinduction used in the paper, due to Coquand (1994), deserves more attention: following the Curry-Howard correspondence the coinductive definition and proof principles both take the form of guarded corecursion for (potentially indexed) lazy data types.

The main developments in the paper have been formalised using the dependently typed, total[1] functional programming language Agda (Norell 2007; Agda Team 2010), which provides good support for mixed induction and coinduction in the style mentioned above. The source code is at the time of writing available to download (Danielsson 2010a).

The rest of the paper is structured as follows: Section 2 gives an introduction to induction and coinduction in the context of Agda. Section 3 defines a small language of recursive types, and Section 4 defines a subtyping relation for this language by viewing the types as potentially infinite trees. Section 5 defines an equivalent, declarative subtyping relation using mixed induction and coinduction, and Section 6 compares this definition to another equivalent definition, given by Brandt and Henglein (1998). Finally Section 7 discusses a potential pitfall associated with the technique we propose, and Section 8 concludes.

2 Induction and Coinduction

This section gives a brief introduction to induction and coinduction, with an emphasis on how these concepts are realised in Agda. For more formal accounts of induction and coinduction see, for instance, the theses of Hagino (1987) and Mendler (1988).

2.1 Induction

Let us start with a simple inductive definition. In Agda the type of *finite* lists can be defined as follows:

[1] Agda is an experimental system. The meta-theory has not been formalised, and the type checker has not been proved bug-free, so take phrases such as "total" with a grain of salt.

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

This states that *List A* is a type (or *Set*) with two constructors, [] of type *List A* and _::_ of type $A \to List\ A \to List\ A$. The constructor _::_ is an infix operator; the underscores mark the argument positions. The type *List A* is isomorphic to the least fixpoint $\mu X.\ 1 + A \times X$ in the category of types and total functions.[2]

Agda has a termination checker which ensures that all code is terminating (or productive, see below). It is assisted by other checkers which ensure that data types are strictly positive, and not too large. The termination checker allows lists to be destructed using structural recursion:

```
map : {A B : Set} → (A → B) → List A → List B
map f []       = []
map f (x :: xs) = f x :: map f xs
```

The use of braces in $\{A\ B : Set\} \to \ldots$ means that the two type arguments *A* and *B* are *implicit*; they do not need to be given explicitly if Agda can infer them. Note that in this context *A B* is not an application, it is a sequence of variables.

2.2 Coinduction

If we want to have infinite lists, or streams, we can use the following coinductive definition instead (note that constructors, such as _::_, can be overloaded in Agda):

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

The type *Stream A* is isomorphic to the greatest fixpoint $\nu X.\ A \times X$. The type function $\infty : Set \to Set$ marks its argument as being coinductive. It is analogous to the suspension type constructors which are sometimes used to implement non-strictness in strict languages (Wadler et al. 1998), and comes with a force function and a delay constructor:

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

The constructor ♯_ is a tightly binding prefix operator. Ordinary function application binds tighter, though.

Values of coinductive types can be constructed using *guarded* corecursion (Coquand 1994):

[2] At the time of writing this is not exactly true in Agda (Danielsson and Altenkirch 2009), but the difference between *List A* and the fixpoint is irrelevant for the purposes of this paper. Similar considerations apply to greatest fixpoints.

$$map_{\mathrm{S}} : \{A\ B : Set\} \to (A \to B) \to Stream\ A \to Stream\ B$$
$$map_{\mathrm{S}}\ f\ (x :: xs) = f\ x ::^{\sharp} map_{\mathrm{S}}\ f\ (^{\flat} xs)$$

The definition of map_{S} is accepted by Agda's termination checker because the corecursive call is guarded by $\sharp\_$, without any non-constructor function between the left-hand side and the corecursive call. This syntactic notion of guardedness ensures that corecursive definitions are *productive*: even if the value being constructed is infinite, the next constructor can always be computed in a finite number of steps.

It may also be instructive to see (attempted) definitions which are not accepted:

$$bad : Stream\ \mathbb{N}$$
$$bad = \mathsf{zero} ::^{\sharp} tail\ bad$$

$$nats : Stream\ \mathbb{N}$$
$$nats = \mathsf{zero} ::^{\sharp} map_{\mathrm{S}}\ \mathsf{suc}\ nats$$

Both definitions are rejected because they are not guarded, but only the first one is non-productive; *nats* uniquely specifies the stream of natural numbers, but is rejected by the termination checker because it does not satisfy the syntactic criterion imposed by Agda.

2.3 Coinductive Relations

Let us now consider a coinductively defined *relation*: stream equality, also known as bisimilarity. Two streams are equal if they have identical heads and their tails are equal (coinductively):

$$\frac{^{\flat} xs \approx {}^{\flat} ys}{x :: xs \approx x :: ys}\ \text{(coinductive)}$$

This inference system can be represented using an indexed data type:

$$\begin{aligned}&\textbf{data}\ \_\approx\_\ \{A : Set\} : Stream\ A \to Stream\ A \to Set\ \textbf{where}\\ &\quad \_::\_ : (x : A)\ \{xs\ ys : \infty\ (Stream\ A)\} \to \infty\ (^{\flat} xs \approx {}^{\flat} ys) \to\\ &\qquad x :: xs \approx x :: ys\end{aligned}$$

Some remarks on this definition may be useful:

- The elements of the type $xs \approx ys$ are *proofs* witnessing the equality of xs and ys. Agda does not make a distinction between proofs and programs, and the termination checker ensures productivity of both kinds of definition.
- *Dependent* function spaces ($(x : A) \to B$ where x can occur in B) are used to set up dependencies of types on values.
- The first occurrence of the type constructor ∞ just reflects the fact that the second argument to the stream constructor $\_::\_$ is delayed. The second occurrence is necessary to be able to construct infinite equality proofs; if we had omitted it the relation would have been empty.

– We overload the constructor $\_::\_$ so that it stands both for the "cons" function for streams, and for the proof that cons preserves equality. The constructors can be disambiguated based on type information.

Elements of coinductively defined relations can be constructed using corecursion. As an example, let us prove the map-iterate property (Gibbons and Hutton 2005):

$$\mathit{map}_{\mathrm{S}}\ f\ (\mathit{iterate}\ f\ x)\ \approx\ \mathit{iterate}\ f\ (f\ x).$$

The function *iterate* repeatedly applies a function to a seed element and collects the results in a stream:

$$\mathit{iterate}\ f\ x\ =\ x :: ^{\sharp}\ (f\ x :: ^{\sharp}\ (f\ (f\ x) :: \ldots)).$$

The function is defined corecursively:

$$\begin{aligned}&\mathit{iterate}\ :\ \{A\ :\ \mathit{Set}\}\ \to\ (A\ \to\ A)\ \to\ A\ \to\ \mathit{Stream}\ A\\ &\mathit{iterate}\ f\ x\ =\ x :: ^{\sharp}\ \mathit{iterate}\ f\ (f\ x)\end{aligned}$$

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

$$\begin{aligned}&\mathit{map\text{-}iterate}\ :\ \{A\ :\ \mathit{Set}\}\ (f\ :\ A\ \to\ A)\ (x\ :\ A)\ \to\\ &\qquad\qquad \mathit{map}_{\mathrm{S}}\ f\ (\mathit{iterate}\ f\ x)\ \approx\ \mathit{iterate}\ f\ (f\ x)\\ &\mathit{map\text{-}iterate}\ f\ x\ =\ f\ x :: ^{\sharp}\ \mathit{map\text{-}iterate}\ f\ (f\ x)\end{aligned}$$

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 $\mathit{map}_{\mathrm{S}}\ f\ (\mathit{iterate}\ f\ x)\ \approx\ \mathit{iterate}\ f\ (f\ x)$. Agda types should always be read up to normalisation, so this is equivalent to[3]

$$f\ x :: ^{\sharp}\ \mathit{map}_{\mathrm{S}}\ f\ (^{\flat}\ (^{\sharp}\ \mathit{iterate}\ f\ (f\ x)))\ \approx\ f\ x :: ^{\sharp}\ \mathit{iterate}\ f\ (f\ (f\ x)).$$

(Note that normalisation does not involve reduction 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

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

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] This is a simplification of the current behaviour of Agda.

2.4 Mixed Induction and Coinduction

The types above are either inductive or coinductive. Let us now discuss a type which uses both induction and coinduction. Hancock et al. (2009) define a language of stream processors, representing functions of type $\mathit{Stream}\ A \to \mathit{Stream}\ B$, using a nested fixpoint: $\nu Y.\mu X.\ B \times Y + (A \to X)$. We can represent this fixpoint in Agda as follows:

```
data SP (A B : Set) : Set where
  put : B → ∞ (SP A B) → SP A B
  get : (A → SP A B)   → SP A B
```

The stream processor $\mathsf{put}\ b\ sp$ outputs b, and continues processing according to sp. The processor $\mathsf{get}\ f$ reads one element a from the input stream, and continues processing according to $f\ a$. In the case of put the recursive argument is coinductive, so it is fine to output an infinite number of elements, whereas in the case of get the recursive argument is inductive, which means that one can only read a finite number of elements before writing the next one. This ensures that the output stream can be generated productively.

We can implement a simple stream processor which copies the input to the output as follows:

```
copy : {A : Set} → SP A A
copy = get (λ a → put a (♯ copy))
```

This definition is guarded. Note that *copy* contains an infinite number of get constructors. This is fine, even though get's argument is inductive, because there is never a stretch of infinitely many get constructors without an intervening delay constructor ($\sharp\_$). On the other hand, the following definition of a sink is not guarded, and is not accepted by Agda:

```
sink : {A B : Set} → SP A B
sink = get (λ _ → sink)
```

As another example we can compute the semantics of a stream processor:

```
⟦_⟧ : {A B : Set} → SP A B → Stream A → Stream B
⟦ put b sp ⟧ as       = b :: ♯ (⟦ ♭ sp ⟧ as)
⟦ get f    ⟧ (a :: as) = ⟦ f a ⟧ (♭ as)
```

($⟦\_⟧$ is a mixfix operator.) This definition uses a lexicographic combination of guarded corecursion and higher-order structural recursion (see Section 2.5). In the first clause the corecursive call is guarded. In the second clause it "preserves guardedness" (it takes place under zero coinductive constructors rather than one), and the first argument is structurally smaller.

Note that $⟦\_⟧$ could not have been implemented if $\mathit{SP}\ A\ B$ had been defined purely coinductively (because then *sink* could be implemented with B equal to the empty type). By using both induction and coinduction in the definition we rule out certain stream processors which would otherwise have been accepted, and in return we can implement functions like $⟦\_⟧$.

2.5 A Criterion for Totality

Let us now make things more precise by giving a more detailed explanation of Agda's criterion for accepting a function as being total. The results in the paper do not depend on the exact criterion used by Agda, so we only give a conservative approximation of what is currently implemented. The description below is based on the termination checker foetus (Abel and Altenkirch 2002), extended with support for guarded coinduction based on an idea due to Andreas Abel (personal communication).

First we collect some information about the program. For every left-hand side $f\ p_1\ \ldots\ p_m$ and function call $g\ e_1\ \ldots\ e_n$ in the corresponding right-hand side the following information is recorded:

Argument structure. For every pair (p_i, e_j) it is noted if the argument e_j is structurally strictly smaller (denoted by $<$) or equal to ($=$) the pattern p_i. If neither case applies, then we use the notation ?. Note that x is not structurally smaller than $\sharp\ x$, and that $f\ x$ is strictly smaller than $\mathsf{c}\ f$, for an inductive constructor c.

Guardedness. It is also noted whether the call is guarded by constructors, at least one of which is coinductive ($<$); or whether guardedness is preserved, i.e. if the call is guarded by inductive constructors ($=$).

The next step is to combine the information about individual calls into information about all the call paths from one function to itself. We use the notation $(g \mid a_1\ \ldots\ a_n)$ to describe the information computed for a call path; here g is the guardedness information, and a_i describes how the i-th argument is changed. In the case of the function $[\![\_]\!]$ from Section 2.4 we get that there are three kinds of call paths:

1. $(< \mid = = ?\ =)$, which corresponds to the first recursive call;
2. $(= \mid = = <\ ?)$, which corresponds to the second recursive call; and
3. $(< \mid = = ?\ ?)$ for call paths which involve both recursive calls.

Finally we can give the criterion for totality: a function is accepted as total if there is some lexicographic combination of the components for which every call path is strictly decreasing. In the case of $[\![\_]\!]$ it suffices to combine the guardedness with the information about the third argument (the stream processor).

As noted by Danielsson and Altenkirch (2009, Section 7.1) the criterion above works best if all fixpoints have the form $\nu Y.\mu X.\ F\ X\ Y$ (for suitable values of F); we have not yet found a good way to incorporate fixpoints of the form $\mu X.\nu Y.\ F\ X\ Y$. However, this issue does not affect the examples in this paper.

2.6 Relations Using Mixed Induction and Coinduction

As a final example we define a relation using mixed induction and coinduction. Capretta (2005) defines the partiality monad, which can be used to represent potentially non-terminating computations, 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. Non-termination is represented as an infinitely postponed computation:

```
⊥ : {A : Set} → A ^ν
⊥ = step (♯ ⊥)
```

A natural definition of equality for partial computations is weak bisimilarity (viewing step as a silent transition):[4]

```
data _≅_ : A ^ν → A ^ν → Set where
  return :                    return v ≅ return v
  step   : ∞ (♭ x ≅ ♭ y)   → step x   ≅ step y
  step^r :      x ≅ ♭ y    →      x   ≅ step y
  step^l :    ♭ x ≅   y    → step x   ≅      y
```

This is basically the congruence generated by return and step, but allowing for finite differences in delay. Note that the requirement of *finite* differences in delay is captured by the use of induction for step$^{\mathrm{r}}$ and step$^{\mathrm{l}}$, while the use of coinduction for step is necessary to be able to prove that the relation is reflexive.

3 Recursive Types

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

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

Here $\bot$ and $\top$ are the least and greatest types, respectively, X is a variable, $\sigma \rightarrowtriangle \tau$ is a function type, and $\mu X.\, \sigma \rightarrowtriangle \tau$ 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 $\mu X.\sigma \rightarrowtriangle \tau$ should be equivalent to its unfolding $(\sigma \rightarrowtriangle \tau)[X := \mu X.\, \sigma \rightarrowtriangle \tau]$. It would be unproblematic to extend the language with other type constructors, such as products and sums.

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
```

[4] In order to reduce clutter the declarations of implicit arguments have been omitted in the remainder of the paper.

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:

$$\_[\_]\ :\ Ty\ (1 + n)\ \to\ Ty\ n\ \to\ Ty\ n$$

The following function unfolds a fixpoint one step:

$$\begin{aligned} &unfold\langle \mu\_ \to \_ \rangle\ :\ Ty\ (1 + n)\ \to\ Ty\ (1 + n)\ \to\ Ty\ n \\ &unfold\langle \mu\ \sigma \to \tau\ \rangle\ =\ (\sigma \to \tau)\ [\ \mu\ \sigma \to \tau\ ] \end{aligned}$$

(Note that $\mu\_ \to \_$, $\_[\_]$ and $unfold\langle \mu\_ \to \_ \rangle$ are all mixfix operators which take two arguments.)

4 Subtyping via Trees

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

$$\begin{aligned} &\textbf{data}\ Tree\ (n\ :\ \mathbb{N})\ :\ Set\ \textbf{where} \\ &\quad \bot\ \ :\ Tree\ n \\ &\quad \top\ \ :\ Tree\ n \\ &\quad \mathsf{var}\ \ :\ Fin\ n\ \to\ Tree\ n \\ &\quad \_ \to \_\ :\ \infty\ (Tree\ n)\ \to\ \infty\ (Tree\ n)\ \to\ Tree\ n \end{aligned}$$

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

$$\begin{aligned} &\textbf{data}\ \_ \leqslant_{\mathrm{Tree}} \_\ :\ Tree\ n\ \to\ Tree\ n\ \to\ Set\ \textbf{where} \\ &\quad \bot\ \ :\ \bot\ \ \leqslant_{\mathrm{Tree}}\ \tau \\ &\quad \top\ \ :\ \sigma\ \ \leqslant_{\mathrm{Tree}}\ \top \\ &\quad \mathsf{var}\ \ :\ \mathsf{var}\ x\ \leqslant_{\mathrm{Tree}}\ \mathsf{var}\ x \\ &\quad \_ \to \_\ :\ \infty\ (\flat\ \tau_1\ \leqslant_{\mathrm{Tree}}\ \flat\ \sigma_1)\ \to\ \infty\ (\flat\ \sigma_2\ \leqslant_{\mathrm{Tree}}\ \flat\ \tau_2)\ \to \\ &\qquad\quad \sigma_1 \to \sigma_2\ \leqslant_{\mathrm{Tree}}\ \tau_1 \to \tau_2 \end{aligned}$$

Note the contravariant treatment of the codomain of the function space. Note also that the constructors of *Tree* are overloaded—repeatedly—in order to reduce clutter.

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

$$\begin{aligned} &[\![\_]\!]\ :\ Ty\ n\ \to\ Tree\ n \\ &[\![\ \bot\ ]\!]\ \ \ \ \ \ \ \ =\ \bot \\ &[\![\ \top\ ]\!]\ \ \ \ \ \ \ \ =\ \top \\ &[\![\ \mathsf{var}\ x\ ]\!]\ \ \ \ \ =\ \mathsf{var}\ x \\ &[\![\ \ \ \sigma \to \tau\ ]\!]\ =\ \sharp\ [\![\ \sigma\ ]\!]\ \ \ \ \ \ \to \sharp\ [\![\ \tau\ ]\!] \\ &[\![\ \mu\ \sigma \to \tau\ ]\!]\ =\ \sharp\ [\![\ \sigma\ [\ \chi\ ]\ ]\!] \to \sharp\ [\![\ \tau\ [\ \chi\ ]\ ]\!] \\ &\qquad\qquad\quad \textbf{where}\ \chi\ =\ \mu\ \sigma \to \tau \end{aligned}$$

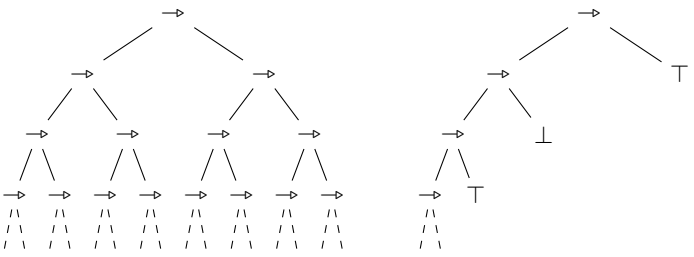

Fig. 1. The first levels of the infinite trees corresponding to the types $\mu X.\ X \to X$ and $\mu X.\ (X \to \bot) \to \top$

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

$$\begin{aligned}&\_\leqslant_{\text{Type}}\_ \ :\ Ty\ n\ \to\ Ty\ n\ \to\ Set\\ &\sigma\ \leqslant_{\text{Type}}\ \tau\ =\ [\![\ \sigma\ ]\!]\ \leqslant_{\text{Tree}}\ [\![\ \tau\ ]\!]\end{aligned}$$

As a simple example, consider the following two types, $\sigma\ =\ \mu X.\ X \to X$ and $\tau\ =\ \mu X.\ (X \to \bot) \to \top$:

$$\begin{aligned}&\sigma\ :\ Ty\ 0 && \tau\ :\ Ty\ 0\\ &\sigma\ =\ \mu\ \mathsf{var\ zero} \to \mathsf{var\ zero} && \tau\ =\ \mu\ (\mathsf{var\ zero} \to \bot) \to \top\end{aligned}$$

The first few levels of the infinite trees corresponding to these types can be seen in Fig. 1. It is straightforward to show that σ is a subtype of τ using a corecursive proof:

$$\begin{aligned}&\sigma{\leqslant}\tau\ :\ \sigma\ \leqslant_{\text{Type}}\ \tau\\ &\sigma{\leqslant}\tau\ =\ \sharp\ (\sharp\ \sigma{\leqslant}\tau \to \sharp\ \bot) \to \sharp\ \top\end{aligned}$$

(Note that $\sigma{\leqslant}\tau$ is an identifier and not a compound expression; almost any character string which does not contain whitespace can be used as an identifier.)

Amadio and Cardelli (1993) also define subtyping for recursive types by going via potentially infinite trees, but they define a subtyping relation *inductively* on finite trees, and state 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 $\sigma\ \leqslant_{\text{Type}}\ \tau$ corecursively; see the code which accompanies the paper (Danielsson 2010a).

5 Subtyping Using Mixed Induction and Coinduction

Subtyping can also be defined directly, without going via trees. The following definition is inspired by one given by Brandt and Henglein (1998), see Section 6:

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

Note that the structural rules ($\bot$, $\top$, $\_\rightarrow\!\!\!\!\bullet\_$) are defined coinductively, while the other rules, most importantly trans, are defined inductively. Note also that the inclusion of refl and trans is essential; if either constructor is removed we get a different relation.

Now, if we can prove that the relation $\_\leqslant\_$ is equivalent to $\_\leqslant_{\mathrm{Type}}\_$ (and thus also equivalent to Amadio and Cardelli's relation), then we have showed what we set out to show: that coinduction and the rule of transitivity can be combined. We can prove completeness by a simple application of guarded corecursion (omitted here):

$$\mathit{complete} : \sigma \leqslant_{\mathrm{Type}} \tau \to \sigma \leqslant \tau$$

The soundness proof is a little more tricky. The following lemmas are easy to prove:

$$
\begin{array}{ll}
\mathit{unfold}_{\mathrm{Type}} & : \mu\,\tau_1 \rightarrow\!\!\!\!\bullet\, \tau_2 \leqslant_{\mathrm{Type}} \mathit{unfold}\langle \mu\,\tau_1 \rightarrow\!\!\!\!\bullet\, \tau_2 \rangle \\
\mathit{fold}_{\mathrm{Type}} & : \mathit{unfold}\langle \mu\,\tau_1 \rightarrow\!\!\!\!\bullet\, \tau_2 \rangle \leqslant_{\mathrm{Type}} \mu\,\tau_1 \rightarrow\!\!\!\!\bullet\, \tau_2 \\
\mathit{refl}_{\mathrm{Type}} & : \tau \leqslant_{\mathrm{Type}} \tau \\
\mathit{trans}_{\mathrm{Type}} & : \tau_1 \leqslant_{\mathrm{Type}} \tau_2 \to \tau_2 \leqslant_{\mathrm{Type}} \tau_3 \to \tau_1 \leqslant_{\mathrm{Type}} \tau_3
\end{array}
$$

Using these lemmas one might think that the following should be accepted as a soundness proof:

$$
\begin{array}{lcl}
\mathit{sound} : \sigma \leqslant \tau \to \sigma \leqslant_{\mathrm{Type}} \tau & & \\
\mathit{sound}\ \bot & = & \bot \\
\mathit{sound}\ \top & = & \top \\
\mathit{sound}\ (\tau_1 \leqslant \sigma_1 \rightarrow\!\!\!\!\bullet\, \sigma_2 \leqslant \tau_2) & = & {}^\sharp\,\mathit{sound}\ ({}^\flat\,\tau_1 \leqslant \sigma_1) \rightarrow\!\!\!\!\bullet\, {}^\sharp\,\mathit{sound}\ ({}^\flat\,\sigma_2 \leqslant \tau_2) \\
\mathit{sound}\ \mathsf{unfold} & = & \mathit{unfold}_{\mathrm{Type}} \\
\mathit{sound}\ \mathsf{fold} & = & \mathit{fold}_{\mathrm{Type}} \\
\mathit{sound}\ \mathsf{refl} & = & \mathit{refl}_{\mathrm{Type}} \\
\mathit{sound}\ (\mathsf{trans}\ \tau_1 \leqslant \tau_2\ \tau_2 \leqslant \tau_3) & = & \mathit{trans}_{\mathrm{Type}}\ (\mathit{sound}\ \tau_1 \leqslant \tau_2)\ (\mathit{sound}\ \tau_2 \leqslant \tau_3)
\end{array}
$$

However, consider the case for trans. The arguments to the recursive calls are structurally smaller than the inputs, but $\mathit{trans}_{\mathrm{Type}}$ is not a constructor, so guardedness is not preserved. The proof is productive (given a suitable definition of $\mathit{trans}_{\mathrm{Type}}$), but Agda's termination checker cannot see this.

In the absence of improved termination checking for Agda we provide a workaround, using a technique described by Danielsson (2010b). If $\mathit{trans}_{\mathrm{Type}}$

had been a constructor then the definition of *sound* would have been accepted, and this observation can be used to rescue the proof. First we define a variant of $\_\leqslant_{\mathrm{Tree}}\_$ which includes an extra inductive constructor, trans:

$$
\begin{array}{l}
\textbf{data}\ \_\leqslant_{\mathrm{TreeP}}\_ : \mathit{Tree}\ n \to \mathit{Tree}\ n \to \mathit{Set}\ \textbf{where} \\
\quad \bot \quad : \bot \leqslant_{\mathrm{TreeP}} \tau \\
\quad \top \quad : \sigma \leqslant_{\mathrm{TreeP}} \top \\
\quad \mathsf{var} \quad : \mathsf{var}\ x \leqslant_{\mathrm{TreeP}} \mathsf{var}\ x \\
\quad \_\to\!\!\!\triangleright\_ : \infty\ (\flat\ \tau_1 \leqslant_{\mathrm{TreeP}} \flat\ \sigma_1) \to \infty\ (\flat\ \sigma_2 \leqslant_{\mathrm{TreeP}} \flat\ \tau_2) \to \\
\qquad\quad \sigma_1 \to \sigma_2 \leqslant_{\mathrm{TreeP}} \tau_1 \to \tau_2 \\
\quad \mathsf{trans} : \tau_1 \leqslant_{\mathrm{TreeP}} \tau_2 \to \tau_2 \leqslant_{\mathrm{TreeP}} \tau_3 \to \tau_1 \leqslant_{\mathrm{TreeP}} \tau_3
\end{array}
$$

The letter P stands for "program"; this type defines a small language of equality proof programs. It is easy to turn proofs into proof programs corecursively:

$$\ulcorner\_\urcorner : \sigma \leqslant_{\mathrm{Tree}} \tau \to \sigma \leqslant_{\mathrm{TreeP}} \tau$$

We can now write a guarded proof *program* which "proves" soundness:

$$
\begin{array}{lcl}
\mathit{sound}_{\mathrm{P}} : \sigma \leqslant \tau \to [\![\ \sigma\ ]\!] \leqslant_{\mathrm{TreeP}} [\![\ \tau\ ]\!] & & \\
\mathit{sound}_{\mathrm{P}}\ \bot & = & \bot \\
\mathit{sound}_{\mathrm{P}}\ \top & = & \top \\
\mathit{sound}_{\mathrm{P}}\ (\tau_1 \leqslant \sigma_1 \to\!\!\!\triangleright\ \sigma_2 \leqslant \tau_2) & = & \sharp\ \mathit{sound}_{\mathrm{P}}\ (\flat\ \tau_1 \leqslant \sigma_1) \to\!\!\!\triangleright\ \sharp\ \mathit{sound}_{\mathrm{P}}\ (\flat\ \sigma_2 \leqslant \tau_2) \\
\mathit{sound}_{\mathrm{P}}\ \mathsf{unfold} & = & \ulcorner\ \mathit{unfold}_{\mathrm{Type}}\ \urcorner \\
\mathit{sound}_{\mathrm{P}}\ \mathsf{fold} & = & \ulcorner\ \mathit{fold}_{\mathrm{Type}}\ \urcorner \\
\mathit{sound}_{\mathrm{P}}\ \mathsf{refl} & = & \ulcorner\ \mathit{refl}_{\mathrm{Type}}\ \urcorner \\
\mathit{sound}_{\mathrm{P}}\ (\mathsf{trans}\ \tau_1 \leqslant \tau_2\ \tau_2 \leqslant \tau_3) & = & \mathsf{trans}\ (\mathit{sound}_{\mathrm{P}}\ \tau_1 \leqslant \tau_2)\ (\mathit{sound}_{\mathrm{P}}\ \tau_2 \leqslant \tau_3)
\end{array}
$$

If we can also find a way to turn proof programs into proofs, productively, then we are done. We start by defining a type of weak head normal forms (WHNFs) for the proof programs:

$$
\begin{array}{l}
\textbf{data}\ \_\leqslant_{\mathrm{TreeW}}\_ : \mathit{Tree}\ n \to \mathit{Tree}\ n \to \mathit{Set}\ \textbf{where} \\
\quad \bot \quad : \bot \leqslant_{\mathrm{TreeW}} \tau \\
\quad \top \quad : \sigma \leqslant_{\mathrm{TreeW}} \top \\
\quad \mathsf{var} \quad : \mathsf{var}\ x \leqslant_{\mathrm{TreeW}} \mathsf{var}\ x \\
\quad \_\to\!\!\!\triangleright\_ : \flat\ \tau_1 \leqslant_{\mathrm{TreeP}} \flat\ \sigma_1 \to \flat\ \sigma_2 \leqslant_{\mathrm{TreeP}} \flat\ \tau_2 \to \\
\qquad\quad \sigma_1 \to \sigma_2 \leqslant_{\mathrm{TreeW}} \tau_1 \to \tau_2
\end{array}
$$

Note that the arguments to $\_\to\!\!\!\triangleright\_$ are *programs*, not WHNFs. One can prove by simple case analysis that $\_\leqslant_{\mathrm{TreeW}}\_$ is transitive:

$$\mathit{trans}_{\mathrm{TreeW}} : \tau_1 \leqslant_{\mathrm{TreeW}} \tau_2 \to \tau_2 \leqslant_{\mathrm{TreeW}} \tau_3 \to \tau_1 \leqslant_{\mathrm{TreeW}} \tau_3$$

From this result it follows by structural recursion that programs can be turned into WHNFs:

$$
\begin{array}{lll}
\mathit{whnf} & : \sigma \leqslant_{\mathrm{TreeP}} \tau \to \sigma \leqslant_{\mathrm{TreeW}} \tau & \\
\mathit{whnf}\ \bot & = \bot & \\
\mathit{whnf}\ \top & = \top & \\
\mathit{whnf}\ \mathsf{var} & = \mathsf{var} & \\
\mathit{whnf}\ (\tau_1 \leqslant \sigma_1 \twoheadrightarrow \sigma_2 \leqslant \tau_2) & = {}^\flat\, \tau_1 \leqslant \sigma_1 \twoheadrightarrow {}^\flat\, \sigma_2 \leqslant \tau_2 & \\
\mathit{whnf}\ (\mathsf{trans}\ \tau_1 \leqslant \tau_2\ \tau_2 \leqslant \tau_3) & = \mathit{trans}_{\mathrm{TreeW}}\ (\mathit{whnf}\ \tau_1 \leqslant \tau_2)\ (\mathit{whnf}\ \tau_2 \leqslant \tau_3) &
\end{array}
$$

The following mutually recursive functions then turn proof programs into "actual" proofs by using the *whnf* function repeatedly:

$$
\begin{array}{lll}
[\![\_]\!]_{\mathrm{W}} & : \sigma \leqslant_{\mathrm{TreeW}} \tau \to \sigma \leqslant_{\mathrm{Tree}} \tau & \\
[\![\ \bot &]\!]_{\mathrm{W}} = \bot & \\
[\![\ \top &]\!]_{\mathrm{W}} = \top & \\
[\![\ \mathsf{var} &]\!]_{\mathrm{W}} = \mathsf{var} & \\
[\![\ \tau_1 \leqslant \sigma_1 \twoheadrightarrow \sigma_2 \leqslant \tau_2\ ]\!]_{\mathrm{W}} & = {}^\sharp\, [\![\ \tau_1 \leqslant \sigma_1\ ]\!]_{\mathrm{P}} \twoheadrightarrow {}^\sharp\, [\![\ \sigma_2 \leqslant \tau_2\ ]\!]_{\mathrm{P}} &
\end{array}
$$

$$
\begin{array}{lll}
[\![\_]\!]_{\mathrm{P}} & : \sigma \leqslant_{\mathrm{TreeP}} \tau \to \sigma \leqslant_{\mathrm{Tree}} \tau & \\
[\![\ \sigma \leqslant \tau\ ]\!]_{\mathrm{P}} & = [\![\ \mathit{whnf}\ \sigma \leqslant \tau\ ]\!]_{\mathrm{W}} &
\end{array}
$$

Note that these functions are guarded and hence productive. Finally we get the soundness proof:

$$
\begin{array}{lll}
\mathit{sound} & : \sigma \leqslant \tau \to \sigma \leqslant_{\mathrm{Type}} \tau & \\
\mathit{sound}\ \sigma \leqslant \tau & = [\![\ \mathit{sound}_{\mathrm{P}}\ \sigma \leqslant \tau\ ]\!]_{\mathrm{P}} &
\end{array}
$$

6 Inductive Axiomatisation of Subtyping

Brandt and Henglein (1998) do not define subtyping using mixed induction and coinduction, as in Section 5, but using an inductive *encoding* of coinduction. 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:

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

The subtyping relation is defined as follows:

$$
\begin{array}{lll}
\mathbf{data}\ \_\vdash\_\leqslant\_\ (A : \mathit{List}\ (\mathit{Hyp}\ n)) : \mathit{Ty}\ n \to \mathit{Ty}\ n \to \mathit{Set}\ \mathbf{where} & & \\
\quad \bot & : A \vdash \bot \leqslant \tau & \\
\quad \top & : A \vdash \sigma \leqslant \top & \\
\quad \_\twoheadrightarrow\_ & : \mathbf{let}\ H = \sigma_1 \twoheadrightarrow \sigma_2 \lesssim \tau_1 \twoheadrightarrow \tau_2\ \mathbf{in} & \\
 & \quad H :: A \vdash \tau_1 \leqslant \sigma_1 \to H :: A \vdash \sigma_2 \leqslant \tau_2 \to & \\
 & \quad A \vdash \sigma_1 \twoheadrightarrow \sigma_2 \leqslant \tau_1 \twoheadrightarrow \tau_2 & \\
\quad \mathsf{unfold} & : A \vdash \mu\, \tau_1 \twoheadrightarrow \tau_2 \leqslant \mathit{unfold}\langle \mu\, \tau_1 \twoheadrightarrow \tau_2\ \rangle & \\
\quad \mathsf{fold} & : A \vdash \mathit{unfold}\langle \mu\, \tau_1 \twoheadrightarrow \tau_2\ \rangle \leqslant \mu\, \tau_1 \twoheadrightarrow \tau_2 &
\end{array}
$$

$$
\begin{aligned}
&\mathsf{refl} &&: A \vdash \tau \leqslant \tau \\
&\mathsf{trans} &&: A \vdash \tau_1 \leqslant \tau_2 \to A \vdash \tau_2 \leqslant \tau_3 \to A \vdash \tau_1 \leqslant \tau_3 \\
&\mathsf{hyp} &&: \sigma \lesssim \tau \in A \to A \vdash \sigma \leqslant \tau
\end{aligned}
$$

Here $\_\in\_$ encodes list membership. 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 5 is also decidable.

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. However, we have adapted the algorithm to this setting:

$$\_\leqslant?\_ : (\sigma\ \tau : \mathit{Ty}\ n) \to \mathit{Dec}\ ([\,] \vdash \sigma \leqslant \tau)$$

A value in $\mathit{Dec}\ A$ is either a value in A, or a proof showing that no such value exists, so this decision procedure does not merely say "yes" or "no", it backs up its verdict with solid evidence. Details of the implementation of $\_\leqslant?\_$ are available in the code accompanying the paper (Danielsson 2010a).

We know that $\_\vdash\_\leqslant\_$ is equivalent to $\_\leqslant\_$, because both relations are equivalent to Amadio and Cardelli's. However, it can still be instructive to see a direct proof of soundness of $\_\vdash\_\leqslant\_$ with respect to $\_\leqslant\_$. The proof below uses a cyclic (but productive) proof to turn the inductive encoding of coinduction used in $\_\vdash\_\leqslant\_$ into the "actual" coinduction used in $\_\leqslant\_$.

To state soundness the type *All* is used; $\mathit{All}\ P\ xs$ means that all elements in xs satisfy P:

$$
\begin{aligned}
&\mathbf{data}\ \mathit{All}\ (P : A \to \mathit{Set}) : \mathit{List}\ A \to \mathit{Set}\ \mathbf{where} \\
&\quad [\,] \quad : \mathit{All}\ P\ [\,] \\
&\quad \_::\_ : P\ x \to \mathit{All}\ P\ xs \to \mathit{All}\ P\ (x :: xs)
\end{aligned}
$$

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

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

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

$$
\begin{aligned}
&\mathit{sound\ valid}\ (\mathsf{trans}\ \tau_1{\leqslant}\tau_2\ \tau_2{\leqslant}\tau_3) &&= \mathsf{trans}\ (\mathit{sound\ valid}\ \tau_1{\leqslant}\tau_2)\\
&&&\qquad\quad (\mathit{sound\ valid}\ \tau_2{\leqslant}\tau_3)\\
&\mathit{sound\ valid}\ (\mathsf{hyp}\ h) &&= \mathit{lookup\ valid}\ h\\
&\mathit{sound\ valid}\ (\tau_1{\leqslant}\sigma_1 \rightarrowtriangle \sigma_2{\leqslant}\tau_2) &&= \mathit{proof}\\
&\quad \textbf{where}\ \mathit{proof} &&= \sharp\ \mathit{sound}\ (\mathit{proof} :: \mathit{valid})\ \tau_1{\leqslant}\sigma_1 \rightarrowtriangle\\
&&&\quad\ \ \sharp\ \mathit{sound}\ (\mathit{proof} :: \mathit{valid})\ \sigma_2{\leqslant}\tau_2
\end{aligned}
$$

Note that the last two calls to *sound* extend the list of valid assumptions with the proof currently being defined.

The definition of *proof* above is not guarded, but it would be if *sound* were a constructor. We use the technique from Section 5 to make the proof guarded. The program and WHNF types can be defined mutually as follows:

$$
\begin{aligned}
&\textbf{data}\ \_{\leqslant_{\mathrm{P}}}\_ : \mathit{Ty}\ n \rightarrow \mathit{Ty}\ n \rightarrow \mathit{Set}\ \textbf{where}\\
&\quad \mathsf{sound} : \mathit{All}\ (\mathit{Valid}\ \_{\leqslant_{\mathrm{W}}}\_)\ A \rightarrow A \vdash \sigma \leqslant \tau \rightarrow \sigma \leqslant_{\mathrm{P}} \tau\\
&\textbf{data}\ \_{\leqslant_{\mathrm{W}}}\_ : \mathit{Ty}\ n \rightarrow \mathit{Ty}\ n \rightarrow \mathit{Set}\ \textbf{where}\\
&\quad \mathsf{done} : \sigma \leqslant \tau \rightarrow \sigma \leqslant_{\mathrm{W}} \tau\\
&\quad \_{\rightarrowtriangle}\_ : \infty\ (\tau_1 \leqslant_{\mathrm{P}} \sigma_1) \rightarrow \infty\ (\sigma_2 \leqslant_{\mathrm{P}} \tau_2) \rightarrow \sigma_1 \rightarrowtriangle \sigma_2 \leqslant_{\mathrm{W}} \tau_1 \rightarrowtriangle \tau_2\\
&\quad \mathsf{trans} : \tau_1 \leqslant_{\mathrm{W}} \tau_2 \rightarrow \tau_2 \leqslant_{\mathrm{W}} \tau_3 \rightarrow \tau_1 \leqslant_{\mathrm{W}} \tau_3
\end{aligned}
$$

The cases of *sound* listed above are now part of a function $\mathit{sound}_{\mathrm{W}}$ which is used by *whnf* to interpret sound:

$$
\begin{aligned}
&\mathit{sound}_{\mathrm{W}} : \mathit{All}\ (\mathit{Valid}\ \_{\leqslant_{\mathrm{W}}}\_)\ A \rightarrow A \vdash \sigma \leqslant \tau \rightarrow \sigma \leqslant_{\mathrm{W}} \tau\\
&\ldots\\
&\mathit{sound}_{\mathrm{W}}\ \mathit{valid}\ (\mathsf{trans}\ \tau_1{\leqslant}\tau_2\ \tau_2{\leqslant}\tau_3) &&= \mathsf{trans}\ (\mathit{sound}_{\mathrm{W}}\ \mathit{valid}\ \tau_1{\leqslant}\tau_2)\\
&&&\qquad\quad (\mathit{sound}_{\mathrm{W}}\ \mathit{valid}\ \tau_2{\leqslant}\tau_3)\\
&\mathit{sound}_{\mathrm{W}}\ \mathit{valid}\ (\mathsf{hyp}\ h) &&= \mathit{lookup\ valid}\ h\\
&\mathit{sound}_{\mathrm{W}}\ \mathit{valid}\ (\tau_1{\leqslant}\sigma_1 \rightarrowtriangle \sigma_2{\leqslant}\tau_2) &&= \mathit{proof}\\
&\quad \textbf{where}\ \mathit{proof} &&= \sharp\ \mathsf{sound}\ (\mathit{proof} :: \mathit{valid})\ \tau_1{\leqslant}\sigma_1 \rightarrowtriangle\\
&&&\quad\ \ \sharp\ \mathsf{sound}\ (\mathit{proof} :: \mathit{valid})\ \sigma_2{\leqslant}\tau_2\\
&\mathit{whnf} : \sigma \leqslant_{\mathrm{P}} \tau \rightarrow \sigma \leqslant_{\mathrm{W}} \tau\\
&\mathit{whnf}\ (\mathsf{sound}\ \mathit{valid}\ \sigma{\leqslant}\tau) = \mathit{sound}_{\mathrm{W}}\ \mathit{valid}\ \sigma{\leqslant}\tau
\end{aligned}
$$

Note that *proof* is now guarded. For the definitions of $[\![\_]\!]_{\mathrm{W}}$, $[\![\_]\!]_{\mathrm{P}}$ and *sound*, see the accompanying code (Danielsson 2010a).

We have not found a proof of 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. We have implemented a decision procedure *dec* which 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. (The decision procedure *dec*, together with the proof of soundness of $\_{\vdash}\_{\leqslant}\_$, is used to implement the decision procedure $\_{\leqslant}?\_$ mentioned above.)

7 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 old to the new 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 no longer holds (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. Recall the definition of the partiality monad in Section 2.6. One can prove that the equality $\_\cong\_$ is an equivalence relation, and that it is not trivial (assuming that the result type A is inhabited). Let us now add transitivity as an inductive rule:

```
data _≅_ : A ν → A ν → Set where
  ...
  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)))
```

The proof uses the following steps: $x \cong \mathsf{step}\ (\sharp\ x) \cong \mathsf{step}\ (\sharp\ y) \cong y$. (The function *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. Note that the problem with $\_\cong\_$ is closely related to the problem of weak bisimulation up to weak bisimilarity (Sangiorgi and Milner 1992); presumably some of the techniques which have been developed to address the latter problem are also applicable to the former.

There are actually several different ways in which one can close a coinductively defined binary relation

```
data _∼_ : A → A → Set where
  ...
```

under transitivity. We list three:

1. One can include transitivity as a coinductive constructor:

   $$\begin{array}{l}\textbf{data}\ \_\sim\_ : A \to A \to \mathit{Set}\ \textbf{where}\\ \quad \ldots\\ \quad \mathsf{trans} : \infty\,(x \sim y) \to \infty\,(y \sim z) \to x \sim z\end{array}$$

   This amounts to defining the largest relation which is closed under transitivity, and is not very useful, as pointed out in the introduction.
2. One can define the least relation which includes $\_\sim\_$ and is closed under transitivity:

   $$\begin{array}{l}\textbf{data}\ \_\sim'\_ : A \to A \to \mathit{Set}\ \textbf{where}\\ \quad \mathsf{include} : x \sim y \to x \sim' y\\ \quad \mathsf{trans}\ \ : x \sim' y \to y \sim' z \to x \sim' z\end{array}$$

   This "solves" the problem outlined above, because if $\_\sim\_$ is transitive, then $\_\sim\_$ and $\_\sim'\_$ are equivalent. However, in any given proof trans can only be used a finite number of times, and this can be a rather severe restriction. For instance, the definition of $\_\leqslant\_$ in Section 5 would not have been correct if trans had been defined using this method.
3. Finally one can include transitivity as an inductive constructor, like in the definition of $\_\leqslant\_$:

   $$\begin{array}{l}\textbf{data}\ \_\sim\_ : A \to A \to \mathit{Set}\ \textbf{where}\\ \quad \ldots\\ \quad \mathsf{trans} : x \sim y \to y \sim z \to x \sim z\end{array}$$

   This definition often gives a more useful notion of transitivity than the one above, because transitivity can be used anywhere in a proof, infinitely often, as long as there is never a stretch of infinitely many transitivity constructors without any intervening coinductive constructor. However, this notion of transitivity can sometimes be too strong, as illustrated for the partiality monad equality $\_\cong\_$ above: the "infinitely transitive closure" is sometimes the trivial relation.

8 Conclusions

We have showed that coinduction can be usefully combined with the rule of transitivity, and discussed under what conditions the technique is applicable. We have also defined subtyping for recursive types in a new way, and compared this definition to a similar axiomatisation given by Brandt and Henglein (1998). Brandt and Henglein note that their inductive encoding of coinduction seems to be closely related to guarded coinduction, but leave a precise comparison to future work. This paper provides a precise comparison, albeit not for the general case, but only for a particular example (the subtyping relations given in Sections 5 and 6).

It is our hope that this paper provides a compelling example of the use of mixed induction and coinduction. We have found this technique useful in

a number of situations (Danielsson and Altenkirch 2009), and encourage more programming language researchers—as well as programmers interested in guaranteed totality—to become familiar with it.

Acknowledgements. Nils Anders Danielsson would like to thank Peter Hancock for introducing him to the technique of mixed induction and coinduction. Thorsten Altenkirch would like to thank Nicolas Oury for joint work on $\Pi\Sigma$ 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 Graham Hutton, Henrik Nilsson and some anonymous reviewers for feedback which improved the presentation. Both authors gratefully acknowledge funding from EPSRC (grant codes: EP/E04350X/1 and EP/G034109/1).

References

Abel, A.: Mixed inductive/coinductive types and strong normalization. In: Shao, Z. (ed.) APLAS 2007. LNCS, vol. 4807, pp. 286–301. Springer, Heidelberg (2007)

Abel, A., Altenkirch, T.: A predicative analysis of structural recursion. Journal of Functional Programming 12(1), 1–41 (2002)

The Agda Team. The Agda Wiki (2010), `http://wiki.portal.chalmers.se/agda/`

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

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

Barwise, J.: Mixed Fixed Points. In: The Situation in Logic. CSLI Lecture Notes, vol. 17, Center for the Study of Language and Information, Leland Stanford Junior University (1989)

Bradfield, J., Stirling, C.: Modal mu-calculi. In: Handbook of Modal Logic. Studies in Logic and Practical Reasoning, vol. 3. Elsevier, Amsterdam (2007)

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

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

Coquand, T.: Infinite objects in type theory. In: Barendregt, H., Nipkow, T. (eds.) TYPES 1993. LNCS, vol. 806, pp. 62–78. Springer, Heidelberg (1994)

Cousot, P., Cousot, R.: Inductive definitions, semantics and abstract interpretations. In: POPL '92, Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 83–94 (1992)

Danielsson, N.A.: Code accompanying the paper (2010a), `http://www.cs.nott.ac.uk/~nad/`

Danielsson, N.A.: Beating the productivity checker using embedded languages. Draft (2010b)

Danielsson, N.A., Altenkirch, T.: Mixing induction and coinduction. Draft (2009)

de Vries, E.: 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)

Gapeyev, V., Levin, M.Y., Pierce, B.C.: Recursive subtyping revealed. Journal of Functional Programming 12(6), 511–548 (2002)

Gibbons, J., Hutton, G.: Proof methods for corecursive programs. Fundamenta Informaticae 66(4), 353–366 (2005)

Giménez, E.: Un Calcul de Constructions Infinies et son Application à la Vérification de Systèmes Communicants. PhD thesis, Ecole Normale Supérieure de Lyon (1996)

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

Hagino, T.: A Categorical Programming Language. PhD thesis, University of Edinburgh (1987)

Hancock, P., Pattinson, D., Ghani, N.: Representations of stream processors using nested fixed points. Logical Methods in Computer Science 5(3:9) (2009)

Hensel, U., Jacobs, B.: Proof principles for datatypes with iterated recursion. In: Moggi, E., Rosolini, G. (eds.) CTCS 1997. LNCS, vol. 1290, pp. 220–241. Springer, Heidelberg (1997)

Hughes, J., Moran, A.: Making choices lazily. In: FPCA '95, Proceedings of the seventh international conference on Functional programming languages and computer architecture, pp. 108–119 (1995)

Kozen, D., Palsberg, J., Schwartzbach, M.I.: Efficient recursive subtyping. Mathematical Structures in Computer Science 5(1), 113–125 (1995)

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

Levy, P.B.: Infinitary Howe's method. In: Proceedings of the Eighth Workshop on Coalgebraic Methods in Computer Science (CMCS 2006). ENTCS, vol. 164, pp. 85–104 (2006)

Mendler, P.F.: Inductive Definition in Type Theory. PhD thesis, Cornell University (1988)

Milner, R.: 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)

Milner, R., Tofte, M.: Co-induction in relational semantics. Theoretical Computer Science 87(1), 209–220 (1991)

Müller, O., Nipkow, T., von Oheimb, D., Slotosch, O.: HOLCF = HOL + LCF. Journal of Functional Programming 9(2), 191–223 (1999)

Nakata, K., Uustalu, T.: Trace-based coinductive operational semantics for While; Big-step and small-step, relational and functional styles. In: Urban, C. (ed.) TPHOLs 2009. LNCS, vol. 5674, pp. 375–390. Springer, Heidelberg (2009)

Norell, U.: Towards a practical programming language based on dependent type theory. PhD thesis, Chalmers University of Technology and Göteborg University (2007)

Park, D.: On the semantics of fair parallelism. In: Bjorner, D. (ed.) Abstract Software Specifications. LNCS, vol. 86, pp. 504–526. Springer, Heidelberg (1980)

Raffalli, C.: L'Arithmétique Fonctionnelle du Second Ordre avec Points Fixes. PhD thesis, Université Paris VII (1994)

Sangiorgi, D., Milner, R.: The problem of "weak bisimulation up to". In: Cleaveland, W.R. (ed.) CONCUR 1992. LNCS, vol. 630, pp. 32–46. Springer, Heidelberg (1992)

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

Wadler, P., Taha, W., MacQueen, D.: How to add laziness to a strict language, without even being odd. In: Proceedings of the 1998 ACM SIGPLAN Workshop on ML (1998)