Subtyping by folding an inductive relation into a coinductive one

Vladimir Komendantsky[1]

School of Computer Science
University of St Andrews
St Andrews, KY16 9SX, UK
vk10@st-andrews.ac.uk

Abstract. In this paper we show that a prototypical subtype relation that can neither be defined as a least fixed point nor as a greatest fixed point can nevertheless be defined in a dependently typed language with inductive and coinductive types. The definition proceeds alike a fold in functional programming, although a rather unusual one: that is not applied to any starting object. There has been a related construction of bisimilarity in Coq by Nakata and Uustalu recently, however, our case is not concerned with bisimilarity but a weaker notion of similarity that corresponds to recursive subtyping and has it's own interesting problems.

1 Introduction

It is common in practice to have datatypes formed by nested least and greatest fixed points. For example, consider a grammar and parse trees of derivations in that grammar that are allowed to be infinite only below certain non-terminal nodes. Or, a semantic model of a programming language where we distinguish between termination and diverging computation. With dependent types, it is possible to define types such as of grammars or parse trees. However, it is not straightforward to define nested fixed points using implementations of inductive and coinductive type definitions. This is mainly because these type definitions are subject to strong syntactic checks in current implementations of dependently typed languages. A strong restriction is made by type-checkers that require coinductive type definitions to satisfy syntactic soundness constraints simple enough to be machine-checkable. A common form of such syntactic constraints is known simply as *guards*. It is often a programming challenge to avoid guardedness issues and yet define a meaningful coinductive type. There are at least two different methods to encode nested fixed points in type-theoretic proof assistants that are both known as *mixed induction-coinduction*, the first is defined in [8] and the second, in [18]. The former uses a programming construct of suspension computation monad, while the latter seems to rely on a variant of fold function. Suspension monad is efficient and intuitive, however, it has to be supported by the programming language rather than simply implemented on top of it, for which many dependently typed provers would require substantial re-engineering. Not having a sufficient resource for rewriting the implementation of a prover,

we choose the second, probably not so efficient but maybe a bit more portable method and apply the fold pattern on top of the language.

The language in question is Coq [19]. It has dependent products of the form $\forall\ (x : A).\ B$ where x is a variable which is bound in B; the case when x is not free in B is denoted $A \to B$ and is a simple, non-dependent type. Also, Coq features inductive and coinductive type definitions. For the sake of presentation, we do not provide listings of Coq code, which would be plain ASCII. Instead, throughout the paper, we use a human-oriented type-theoretic notation, where Type denotes the universe of types (which is predicative), and inductive and coinductive definitions are displayed in natural-deduction style with single and, respectively, double lines.

Contribution. We develop a method for inductive-coinductive encoding for a class of similarity relations exemplified in the paper by recursive subtyping of μ-types. A mechanised version of our proofs formalised in Coq is also presented without going through too many technical details. The method allows to internalise, in type theory, similarity relations that can neither be defined as an inductive relation nor as a coinductive relation alone but as a relation formed by nesting an inductive relation inside a coinductive relation or vice versa.

The motivation for this work is a better understanding of termination issues in subtyping as an exercise in the higher-order programming style with iteration and coiteration schemes [1] with a possibility of extensions to formalisms such as *extended regular expressions* (with variables that approximate behaviour of backreferences), and paving the way for further extensions and provably correct practical applications. The generic approach to terms with variables allows to completely redefine the structure of substitution for extended cases and yet keep the same fundamental approach to subtyping (or, more generally, similarity).

Outline. In Sec. 2 we explain the subtyping relation construction method. In Sec. 3 we define the object language of recursive types formally, using Coq as the meta-language. In Sec. 4 we define subtyping in the meta-language. Sec. 5 contains the statement and a proof of soundness and completeness of our definition of recursive subtyping with respect to containment of finite and infinite trees. The powerful method of monadic substitution is described in Sec. 6. In fact, this technical section contains precise function definitions used in the earlier sections 3 and 4. Related work on subtyping for recursive types and methods for decision procedures is summarised in Sec. 7 where the *alter ego* mixed induction-coinduction method is described as well. Finally, in Sec. 8 we give concluding remarks. The interested reader can also refer to the accompanying Coq script with definitions and proofs constructed for this paper at the author's web page by the URL http://www.cs.st-andrews.ac.uk/~vk/doc/subfold.v. The script requires Coq with the Ssreflect [10] extension that are available as packages in common operating systems. The Ssreflect extension does not change the type-theoretic foundation of Coq but rather provides enhancements for the tactic language and handy type definitions and lemmas for bounded numbers and tu-

ples used in our formalisation. Therefore a proof without Ssreflect is also possible by routine redefinition of notions already available.

Notational conventions. In the paper, we use a natural-deduction style notation for inductive and coinductive definitions. For example, the inductive definition of the type Σ of dependent sum is written in two steps. First, we define the *universe*, of which our type is inhabitant (to the right of the semicolon):

$$\Sigma \quad : \quad \forall\ (A : \mathsf{Type}).\ (A \to \mathsf{Type}) \to \mathsf{Type}$$

and second, we define the *constructors* of the type by providing a natural deduction rule for each constructor. In the case of Σ, there is only one constructor, and so, only one rule:

$$\frac{\dfrac{x : A}{p : P\ x}}{\mathsf{exist}\ x\ p : \Sigma\ A\ P}$$

with exist being the name of the constructor. The structure of a rule is a finite tree whose root contains the conclusion of the rule. Let us define the *level* of the root of a rule to be 0. If the level of a given node in a tree is n then the level of the roots of its immediate subtrees is $n + 1$. The constructor exist requires three levels, from 0 to 2, because the variable p defined at level 1 depends on another variable x which should be defined first, at level 2. It is standard [19] to define first the notion a *dependent type* (any type $\forall\ (x : A).\ B$ where x is free in B), and then the notion of an *inductive type*. The latter can be quite involving. As a light-weight alternative, we can define and visualise a *dependent inductive type* starting from our presentation of inductive types in terms of rules as follows:

> A *dependent inductive type* is an inductive type with at least one of its constructors defined by a natural deduction rule containing more than 2 levels.

An inductive type corresponds to the least fixed point of the inference operator generated by the set of rules of the type definition. On the other hand, a coinductive type is supposed to approximate the greatest fixed point of the inference operator. At present, the type theory of Coq does not have mixed inductive-coinductive type definitions. Types can be defined either inductively or coinductively, and never both at the same time. This makes it easier to write rules because we do not have a choice of rule notation once it is fixed that a type is inductive or coinductive.

In the case of the definition of the non-dependent inductive relation tylei in Sec. 4, constructor names are omitted for brevity especially because they do not carry information other than that involved in theorem proving only, and we do not refer to these names the paper.

We make an exclusion from the general level pattern by writing lines above roots of 0-level rules, for example,

$$\frac{}{0 : \mathbb{N}}$$

to indicate whether these rules are inductive or coinductive, because any 0-level rule, even though it defines a *value*, is typechecked differently by Coq depending on whether it appears in an inductive or coinductive definition.

2 The fold pattern

Here is the polymorphic type of the familiar list-based left fold function:

$$\mathsf{foldl} : \forall\ (S\ T : \mathsf{Type})\ (f : T \to S \to T)\ (a : T)\ (l : \mathsf{list}\ S).\ T$$

(On a fundamental approach to fold, the reader is advised to refer to [4].) The result of iterative computation of an appropriately typed function f on values from the list l starting from a given value a is aggregated on the left.

Let us now drop the requirement that the fold starts from some object. This removes the first argument of the function f altogether. The fold function that we are going to construct has two dependent arguments: a function f and a collection l (which is not quite a list) of objects of S. This description fits the definition of the following operator $\leq_{\text{intro}}$ generating a coinductive relation $\leq$:

$$\frac{f : (\forall\ E\ F.\ R\ E\ F \to E \leq F) \qquad l : E \leq_R F}{\leq_{\text{intro}}\ R\ E\ F\ f\ l : E \leq F}$$

We can see that the arguments R, E and F can be inferred from the types of f and l. The constructor $\leq_{\text{intro}}$ has two dependent arguments, f and l, and yields an object (in fact, a proof) inhabiting a particular case of relation. The function f can indeed be seen as a mapping of a proof that from an object E we can access another object F by the relation R to a corresponding proof that from E we can also reach F by $\leq$. In other words, f is an inclusion of R into $\leq$. The interpretation of the argument l is a bit more involved. Think of $\leq_R$ as a finite relation encapsulating another, infinite one in such a way that an infinite number of steps is possible only finitely. The latter sounds rather speculative, however, the intuition is that $E \leq_R F$ is alike a type of finite list of certain abstract, possibly infinite objects. Moreover, proofs of $E \leq_R F$ can be perceived as paths from E to F. So, the function $\leq_{\text{intro}}\ R\ E\ F\ f$ collapses the finite list l of possibly infinite paths to a certain infinite coinductively defined object. Thanks to the premises of $\leq_{\text{intro}}$ we are able to compare elements of pairs in the domain of $\leq$ in a finite number of steps possibly infinitely.

It is worth noting that having a coinductive type definition such as that of $\leq$ is nothing close to requiring an infinite amount of memory for objects of that type. The shape of such an object is a regular tree which may have an infinite unfolding but in itself is a finitely presented entity.

3 Recursive types

Below in this section we give a proper inductive definition of recursive types, our object language, in Coq as the the meta-language. However, first recall a

traditional definition of the set of recursive types that uses a grammar [5, 3]:

$$E, F ::= \bot \mid \top \mid X \mid E \twoheadrightarrow F \mid \mu X.\ E \twoheadrightarrow F$$

where X is a symbolic variable taken from a set of variables. The least fixed point operator μ binds free occurrences of the variable X in $E \twoheadrightarrow F$. This definition has neither products nor sums that are needed for practical programming. However, we do not consider product or sum types in this schematic implementation because their treatment follows a similar pattern compared to $\twoheadrightarrow$, see, e.g., [3]. Moreover, we choose to replace named variables in the definition by nameless de Brujin variables, which yields an equivalent and yet more tangible construction.

A nameless variable is essentially a number m with an upper bound n where m represents the depth of the variable under binders in a term with at most n free variables. We will now define an appropriate notion of bounded number. First, take the usual inductive definition of the type of (unary) natural number:

$$\mathbb{N} : \mathsf{Type}$$

$$\frac{}{\mathsf{0} : \mathbb{N}} \qquad \frac{n : \mathbb{N}}{\mathsf{S}\ n : \mathbb{N}}$$

Further on in the paper, $1 + n$ is assumed to be convertible to $S\ n$. Natural numbers enjoy a decidable less-than relation. It can be defined via the usual truncated subtraction and decidable equality, that is, a relation with values true or false of type bool. Let us recall the equality relation on natural numbers as follows:

$$\begin{array}{lcl} \_ == \_ \ : & \mathbb{N} \to \mathbb{N} \to \mathsf{bool} \\ 0 == 0 & = & \mathsf{true} \\ S\ m == S\ n & = & m == n \\ 0 == S\ n & = & \mathsf{false} \\ S\ m == 0 & = & \mathsf{false} \end{array}$$

The less-then relation is then a function

$$m < n \quad = \quad S\ m - n == 0$$

The type of bounded number can be defined now:

$$\mathcal{I} : \mathbb{N} \to \mathsf{Type}$$

$$\frac{\dfrac{m : \mathbb{N}}{p : m < n}}{\mathsf{num}\ n\ m\ p : \mathcal{I}_n}$$

So, we have a dependent inductive type here, with the type of the variable p depending on the value m. The type $\mathcal{I}_n$ is a special dependent pair type, simpler than Σ from the Introduction, specified on the concrete boolean predicate $\lambda\ m.\ m < n$. The proof of $m < n$ is encoded as a boolean value. This has two outcomes. From one side, since it is easier to reason by cases on boolean-valued

relations than on more general relations with values in Type, this definition of bounded number greatly facilitates proof by cases. On the other hand, compared to the algebraic type of finite number employed, for example, in the construction of [8], the type $\mathcal{I}_n$ is a subtype of $\mathbb{N}$ by the coercion

$$\begin{array}{lcl} \mathsf{N\_of\_num}\ (n : \mathbb{N})\ (i : \mathcal{I}_n) & = & \mathbf{let}\ \mathsf{num}\ m\ \_ = i \\ & & \mathbf{in}\ m \end{array}$$

This permits application of lemmas for $\mathbb{N}$ to statements about $\mathcal{I}_n$ without recursive conversion of finite numbers to natural numbers. The question of automation of insertion of the coercion $\mathsf{N\_of\_nat}$ is inessential for the constructions in the paper.

Now we can give our working definition of *recursive types* by induction as follows:

$$\mathsf{ty} : \mathbb{N} \to \mathsf{Type}$$

$$\frac{}{\bot : \mathsf{ty}\ n} \quad \frac{}{\top : \mathsf{ty}\ n} \quad \frac{i : \mathcal{I}_n}{X_i : \mathsf{ty}\ n} \quad \frac{E : \mathsf{ty}\ n \qquad F : \mathsf{ty}\ n}{E \mathrel{-\!\!\triangleright} F : \mathsf{ty}\ n}$$

$$\frac{E : \mathsf{ty}\ (1+n) \qquad F : \mathsf{ty}\ (1+n)}{\mu\ E \mathrel{-\!\!\triangleright} F : \mathsf{ty}\ n}$$

where the constructors are, respectively, the empty type $\bot$, the unit type $\top$, the variable constructor X (indeed, we later use it without an index), the function type constructor $\mathrel{-\!\!\triangleright}$ and the least fixed point arrow type constructor $\mu\ \_ \mathrel{-\!\!\triangleright} \_$.

Recursive types have a correspondence with non-wellfounded (finite or infinite) trees with the following definition by coinduction:

$$\mathsf{tree} : \mathbb{N} \to \mathsf{Type}$$

$$\frac{}{\bot^\infty : \mathsf{tree}\ n} \quad \frac{}{\top^\infty : \mathsf{tree}\ n} \quad \frac{i : \mathcal{I}_n}{X_i^\infty : \mathsf{tree}\ n} \quad \frac{t : \mathsf{tree}\ n \qquad t : \mathsf{tree}\ n}{t \mathrel{-\!\!\triangleright}^\infty u : \mathsf{tree}\ n}$$

Intuitively, trees are views of μ-types unfolded *ad infinitum*. We denote the tree corresponding to a type E by $[\![E]\!]$. It is interesting to see now what is the exact connection of types with trees. We give a definition that uses a piece of notation from later on in this paper. We define $[\![\_]\!]$ corecursively using the function sbst (see the definition in Sec. 6) of capture avoiding substitution of the second argument for all occurrences of variable 0 in the first argument as follows:

$$\begin{array}{lcl} [\![\bot]\!] & = & \bot^\infty \\ [\![\top]\!] & = & \top^\infty \\ [\![X_i]\!] & = & X_i^\infty \\ [\![E \mathrel{-\!\!\triangleright} F]\!] & = & [\![E]\!] \mathrel{-\!\!\triangleright}^\infty [\![F]\!] \\ [\![\mu\ E \mathrel{-\!\!\triangleright} F]\!] & = & [\![\mathsf{sbst}\ E\ (\mu\ E \mathrel{-\!\!\triangleright} F)]\!] \mathrel{-\!\!\triangleright}^\infty [\![\mathsf{sbst}\ F\ (\mu\ E \mathrel{-\!\!\triangleright} F)]\!] \end{array}$$

The straightforward subtree relation $\mathsf{tle}\ n$ on $\mathsf{tree}\ n$ is denoted by $\leq^\infty$ (omitting the implicit argument n):

$$\mathsf{tle}\ n : \mathsf{tree}\ n \to \mathsf{tree}\ n \to \mathsf{Type}$$

$$\frac{}{\bot^\infty \leq^\infty t} \quad \frac{}{t \leq^\infty \top^\infty} \quad \frac{i : \mathcal{I}_n}{X_i^\infty \leq^\infty X_i^\infty} \quad \frac{u_1 \leq^\infty t_1 \qquad t_2 \leq^\infty u_2}{(t_1 \to^\infty t_2) \leq^\infty (u_1 \to^\infty u_2)}$$

Thus, two recursive types are in the subtype relation when their potentially infinite unfoldings are in the subtree relation. Traditionally, subtyping theorems are stated in terms of inductive limits of sequences of approximations of unfoldings of recursive types (e.g., in [3]). Instead of using explicit induction in that way, we rather rely on dependent types of the CIC which allow to define a powerful monadic structure encapsulating unfolding *ad infinitum*. The point here, similar to an observation made by Amadio and Cardelli in [3], is that unfoldings of recursive types are *regular* trees, which we treat using a mix of induction and coinduction.

4 Definition of recursive subtyping

We define the weak similarity relation $\mathsf{tyle}\ n \subseteq \mathsf{ty}\ n \times \mathsf{ty}\ n$ by folding the inductive part of the definition into the coinductive one. Our technique is an illustration of a generic method for folding one relation into another. For the purpose of having notational correspondence to the Coq proofs, we decide to keep both the Coq name for a relation and introduce a mnemonic denotation for it at the same time for readability. In what follows, $\leq_R$ denotes $\mathsf{tylei}\ n\ R$, and $\leq$ denotes $\mathsf{tyle}\ n$ for an implicit parameter n.

First, we define the inductive part $\mathsf{tylei}\ n\ R$ of the subtyping relation (denoting $\mathsf{tylei}\ n\ R\ E\ F$ by $E \leq_R F$, suppressing the implicit argument n):

$$\mathsf{tylei} : \forall\ n.\ (\mathsf{ty}\ n \to \mathsf{ty}\ n \to \mathsf{Type}) \to \mathsf{ty}\ n \to \mathsf{ty}\ n \to \mathsf{Type}$$

$$\frac{}{\bot \leq_R E} \quad \frac{}{E \leq_R \top} \quad \frac{R\ E\ F \qquad R\ G\ H}{F \to G \leq_R E \to H} \quad \frac{}{\mu\ E \to F \leq_R \mathsf{unfld}\ E\ F}$$

$$\frac{}{\mathsf{unfld}\ E\ F \leq_R \mu\ E \to F} \quad \frac{}{E \leq_R E} \quad \frac{E \leq_R F \qquad F \leq_R G}{E \leq_R G}$$

where unfld is the operation that unfolds a μ-redex by substituting the term $\mu\ E \to F$ for the variable 0 in the term $E \to F$. This operation is defined in Sec. 6. Having the rules for reflexivity and transitivity in the inductive part of the subtype relation is essential for this construction. Indeed, by having these rules explicitly, we are able to compare elements of pairs in the domain of the subtype relation in a *finite* number of steps possibly *infinitely*. Leaving transitivity out of the definition would collapse finite and infinite transitivity chains to infinite ones only.

Next step is to fold the inductive relation and produce a weak similarity. This is done by the single-constructor coinductive type:

$$\mathsf{tyle}\ n : \mathsf{ty}\ n \to \mathsf{ty}\ n \to \mathsf{Type}$$

We denote $\mathsf{tyle}\ n\ E\ F$ by $E \leq F$. In the rule below, we keep arguments n, R, E, F of the constructor despite that we can infer these from the types of other parameters because, in the proofs, we have to make partial applications of the constructor to arguments:

$$\frac{r : \mathsf{reflexive}\ R \qquad t : \mathsf{transitive}\ R \qquad f : \forall\ E\ F.\ R\ E\ F \to E \leq F \qquad l : E \leq_R F}{\leq_{\mathrm{intro}}\ n\ R\ r\ t\ E\ F\ f\ l : E \leq F}$$

The only introduction rule for $\leq$ has four hypotheses, namely, that R is reflexive, transitive and a subrelation of $\leq$, and that F is $\leq_R$-accessible from E in finitely many steps (since $\leq_R$ is an inductive relation).

5 Soundness and completeness

Below we give a proof outline for the Main Theorem. For particular details, the reader can refer to the accompanying script.

Main Theorem (Soundness and completeness).

$$\forall\ (n : \mathbb{N})\ (E\ F : \mathsf{ty}\ n).\ E \leq F \leftrightarrow [\![E]\!] \leq^{\infty} [\![F]\!]$$

Proof. "Only if" (completeness). Suppose that the following coinductive hypothesis H holds:

$$\forall\ (n : \mathbb{N})\ (E\ F : \mathsf{ty}\ n).\ [\![E]\!] \leq^{\infty} [\![F]\!] \to E \leq F$$

Let us first define the following *coinduction principle* P:

$$\leq_{\mathrm{intro}}\ n\ Q\ Q_{\mathsf{refl}}\ Q_{\mathsf{trans}}$$

where Q is the relation $\lambda\ E\ F : \mathsf{ty}\ n.\ [\![E]\!] \leq^{\infty} [\![F]\!]$ (note that we abstract over types here, not trees), and Q_{refl} and Q_{trans} are respectively a reflexivity lemma and a transitivity lemma (each proved by straightforward coinduction). The coinduction principle P allows us to prove statements of the kind

$$\forall\ E\ F : \mathsf{ty}\ n.\ \Big(\forall\ E\ F : \mathsf{ty}\ n.\ [\![E]\!] \leq^{\infty} [\![F]\!] \to E \leq F\Big) \to E \leq_Q F \to E \leq F$$

We reason by simple case analysis on $[\![E]\!]$. The case $\bot^{\infty}$ is proved by an application of P to E, F, H and the constructor for $\bot$. Most other cases are proved by case analysis on F and either a similar argument involving the coinductive principle P applied to a single constructor or proof by contradiction. The three remaining cases are

1. $[\![E_1 \to\!\!\!\triangleright E_2]\!] \leq^{\infty} [\![\mu\ F_1 \to\!\!\!\triangleright F_2]\!] \to (E_1 \to\!\!\!\triangleright E_2) \leq (\mu\ F_1 \to\!\!\!\triangleright F_2)$
2. $[\![\mu\ E_1 \to\!\!\!\triangleright E_2]\!] \leq^{\infty} [\![F_1 \to\!\!\!\triangleright F_2]\!] \to (\mu\ E_1 \to\!\!\!\triangleright E_2) \leq (F_1 \to\!\!\!\triangleright F_2)$
3. $[\![\mu\ E_1 \to\!\!\!\triangleright E_2]\!] \leq^{\infty} [\![\mu\ F_1 \to\!\!\!\triangleright F_2]\!] \to (\mu\ E_1 \to\!\!\!\triangleright E_2) \leq (\mu\ F_1 \to\!\!\!\triangleright F_2)$

These are proved by the application of the coinduction principle to E, F, H and the transitivity constructor of $\leq_Q$ with the explicitly unfolded μ-term.

"If" (soundness). We start by admitting the coinductive hypothesis H:

$$\forall\ (n : \mathbb{N})\ (E\ F : \mathsf{ty}\ n).\ E \leq F \to [\![E]\!] \leq^{\infty} [\![F]\!]$$

We eliminate the assumption $E \leq F$ by inverse application of the rule $\leq_{\mathrm{intro}}$. Thus we obtain a relation R on trees and 4 respective premises. By inductive elimination of the premiss $E \leq_R F$ we arrive at the following 4 cases:

1. $E = \bot$;
2. $F = \top$;
3. there exist E_1, E_2, F_1 and F_2 such that $E = E_1 \twoheadrightarrow E_2$, $F = F_1 \twoheadrightarrow F_2$, $R\ F_1\ E_1$ and $R\ E_2\ F_2$;
4. there exists E_1 such that $E_1 = E = F$.

This is proved by application of the rules of $\leq_R$ and the premises of $\leq_{\text{intro}}$ saying that R is reflexive and transitive. (Thus we solve the guardedness issue that would have arisen should we attempt to use reflexivity and transitivity of the coinductive relation $\leq^\infty$ instead of these two premises.) Hence we have four possibilities when $E \leq^\infty F$ can hold. The first two cases are proved by the $\bot$ and $\top$ constructors of $\leq^\infty$. The third case is proved by applying the $\twoheadrightarrow$ constructor, the hypothesis H, and the premiss saying that R is a subrelation of $\leq$. The last case is proved by reflexivity of $\leq^\infty$ (whose proof coinsides with the proof of Q_{refl} from the completeness part). □

6 Monadic substitution

Definitions in this technical section have implicit arguments being systematically omitted for conciseness. We implemented in Coq a generic notion of symbolic substitution introduced in [2] for untyped lambda terms. It is based on the notion of *universe of types*, that is, a function space $A \to \mathsf{Type}$ where A can be any given type. The type A is said to *index* the type Type. For effective indexing, the index type should be countable, and for that, it suffices to consider the type $\mathbb{N}$ of natural numbers. The terminology and basic notation are similar to those of McBride [15], although we do not use the notion of a context of types of in the definition of the universe, which makes it more generic. We call the resulting universe *stuff* (referring to its abstract character):

$$\mathsf{stuff} : \mathsf{Type}$$

$$\mathsf{stuff} = \mathbb{N} \to \mathsf{Type}$$

For a given n, the intended meaning of $\mathsf{stuff}\ n$ is *stuff with n variables*. The intention is to have a general category of objects such as formulas with n free variables. Yet objects of this category are not endowed with structure making it applicable to an as wide variety of situations as possible.

In the foundation of the method is a type of monadic structure called *kit*. Here we modify the version of [15] by removing the notion of a context of types:

$$\mathsf{kit} : \mathsf{stuff} \to \mathsf{stuff} \to \mathsf{Type}$$

$$\frac{\mathit{var} : \forall\ n.\ \mathcal{I}_n \to U\ n \qquad \mathit{lift} : \forall\ n.\ U\ n \to T\ n \qquad \mathit{wk} : \forall\ n.\ U\ n \to U\ (1+n)}{\mathsf{Kit}\ \mathit{var}\ \mathit{lift}\ \mathit{wk} : \mathsf{kit}\ U\ T}$$

A *substitution* of type $\mathsf{sub}\ T\ m\ n$ is such that it applies to stuff with at most m variables and yields stuff with at most n variables. Hence a substitution is essentially an m-tuple of $T\ n$, that is,

$$\mathsf{sub} : \mathsf{stuff} \to \mathbb{N} \to \mathbb{N} \to \mathsf{Type}$$

$$\mathsf{sub}\ T\ m\ n = m\text{-}\mathsf{tuple}\ (T\ n)$$

Here are basic functions on substitutions, with their types. We need a function to lift a substitution to the next order. This is implemented in the function $\mathsf{lift\_sub}$ of type

$$\forall\ (T\ U : \mathsf{stuff})\ (K : \mathsf{kit}\ T\ U)\ m\ n.\ \mathsf{sub}\ T\ m\ n \to \mathsf{sub}\ T\ (1+m)\ (1+n)$$

whose definition is by case analysis on K.

Next function to consider is the identity substitution $\mathsf{id\_sub}$:

$$\mathsf{id\_sub} : \forall\ (T\ U : \mathsf{stuff})\ (K : \mathsf{kit}\ T\ U)\ n.\ \mathsf{sub}\ T\ n\ n$$

It is defined by induction on n, applying $\mathsf{lift\_sub}$ on the inductive step.

We define a substitution function sub_0 that applies to stuff T with $1+n$ variables, substitutes a given term E for the variable 0, and returns stuff T with n free variables:

$$\mathsf{sub}_0 : \forall\ (T\ U : \mathsf{stuff})\ (K : \mathsf{kit}\ T\ U)\ n\ (E : T\ n).\ \mathsf{sub}\ T\ (1+n)\ n$$

and the definition is simply by consing E with $\mathsf{id\_sub}\ K\ n$.

The weakening function

$$\mathsf{wkn\_sub} : \forall\ m\ n\ T\ U\ (K : \mathsf{kit}\ T\ U).\ \mathsf{sub}\ T\ n\ (m+n)$$

is defined by induction on m, applying $\mathsf{id\_sub}$ on the inductive basis, and tuple mapping on the inductive step.

Substitutions can be easily endowed with structure of composition because they take stuff and return stuff. In order to establish compositionality on substitutions, we define applicative structure on substitutions which is called subApp:

$$\mathsf{subApp} : \mathsf{stuff} \to \mathsf{Type}$$

$$\frac{\mathit{var} : \forall\ n.\ \mathcal{I}_n \to T\ n \qquad \mathit{app} : \forall\ U\ m\ n.\ \mathsf{kit}\ U\ T \to T\ m \to \mathsf{sub}\ U\ m\ n \to T\ n}{\mathsf{SubApp}\ \mathit{var}\ \mathit{app} : \mathsf{subApp}\ T}$$

Our next goal is to define a concrete kit on stuff T given an applicative structure of substitutions on T. This is done in $\mathsf{stuffKit}$ below. We need three components of a kit: variables, lifting and weakening. For variables, we can simply reuse those of the applicative structure. Lifting is simply the identity here. Only weakening requires further definitions for substitution of variables.

Define a kit for variables as follows:

$$\mathsf{varKit} : \forall\ (T : \mathsf{stuff}).\ (\forall\ n.\ \mathcal{I}_n \to T\ n) \to \mathsf{kit}\ \mathcal{I}\ T$$

$$\mathsf{varKit}\ vr = \mathsf{Kit}\ (\lambda\ n.\ \mathsf{id}\ \mathcal{I}_n)\ vr\ (\mathsf{rshift}\ 1)$$

where rshift 1 is the operation of increment of the upper bound by 1.

Substitution of variables is defined below:

$$\mathsf{varSub} : \forall\ m\ n\ (T : \mathsf{stuff}).\ \mathsf{subApp}\ T \to T\ m \to \mathsf{sub}\ \mathcal{I}\ m\ n \to T\ n$$

$$\mathsf{varSub}\ a = \mathbf{let}\ \mathsf{SubApp}\ vr\ ap = a\ \mathbf{in}\ ap\ \mathcal{I}\ m\ n\ (\mathsf{varKit}\ vr)$$

The required kit on stuff T has the following type:

$$\mathsf{stuffKit} : \forall\ (T : \mathsf{stuff}).\ \mathsf{subApp}\ T \to \mathsf{kit}\ T\ T$$

Its variables and lifting are as defined above. Weakening is derived from substitution of variables.

$$\begin{aligned}\mathsf{stuffKit}\ a\ \ =\ \ & \mathbf{let}\ \mathsf{SubApp}\ vr\ \_ = a\ \mathbf{in}\\ & \mathsf{Kit}\ vr\ (\lambda\ \_.\ \mathsf{id})\ \big(\lambda\ n\ E.\ (\mathsf{varSub}\ a)\ E\ (\mathsf{wkn\_sub}\ 1\ n\ (\mathsf{varKit}\ vr))\big)\end{aligned}$$

Finally, for the generic applicative structure, the canonical weakening function wkstuff from stuff with n variables to stuff with $1 + n$ variables is defined as follows:

$$\mathsf{wkstuff} : \forall\ T\ (K : \mathsf{kit}\ T\ T)\ (a : \mathsf{subAppT})\ n.\ \mathsf{sub}\ T\ n\ (1+n) \to T\ n \to T\ (1+n)$$

$$\begin{aligned}\mathsf{wkstuff}\ K\ a\ s\ \ =\ \ & \mathbf{let}\ \mathsf{SubApp}\ \_\ ap = a\ \mathbf{in}\\ & \lambda\ E.\ (ap\ T\ n\ (1+n)\ K)\ E\ s\end{aligned}$$

A straightforward substitution strategy is implemented by the function trav below that traverses a term E and applies a given substitution s. Note that, since s is a tuple, s_i is a consistent notation for the i-th element of s.

$$\begin{aligned}&\mathsf{trav} && : && \forall\ T\ m\ n.\ \mathsf{kit}\ T\ \mathsf{ty} \to \mathsf{ty}\ m \to \mathsf{sub}\ T\ m\ n \to \mathsf{ty}\ n\\ &\mathsf{trav}\ K\ \bot\ s && = && \bot\\ &\mathsf{trav}\ K\ \top\ s && = && \top\\ &\mathsf{trav}\ K\ X_i\ s && = && \mathsf{let}\ \mathsf{Kit}\ \_\ li\ \_ = K\ \mathsf{in}\ li\ \_\ s_i\\ &\mathsf{trav}\ K\ (F \twoheadrightarrow G)\ s && = && (\mathsf{trav}\ K\ F\ s) \twoheadrightarrow (\mathsf{trav}\ K\ G\ s)\\ &\mathsf{trav}\ K\ (\mu\ F \twoheadrightarrow G)\ s && = && \mu\ (\mathsf{trav}\ K\ F\ (\mathsf{lift\_sub}\ K\ s)) \twoheadrightarrow (\mathsf{trav}\ K\ G\ (\mathsf{lift\_sub}\ K\ s))\end{aligned}$$

The traverse function allows to define an instance of the applicative structure on ty that we call tyApp, in Figure 1. In the definition of tyApp, we denoted the constructor of nameless variables by X. This is a consistent notation since we defined, in Sec. 3, that X_i is of type ty n for a given natural number n and a bounded number i of type $\mathcal{I}_n$. So, X is a function of type $\forall\ n.\ \mathcal{I}_n \to \mathsf{ty}\ n$. The monadic structure on ty can now be defined as a stuff kit specified on type ty. The function subty_0 substitutes a given μ-type E for the 0-th variable. It is defined using the generic substitution sub_0 we defined above. Weakening is specialised on ty by the function wkty. The function sbst is capture-avoiding substitution of

```
tyApp      :  subApp ty
tyApp      =  SubApp X trav
tyKit      :  kit ty ty
tyKit      =  stuffKit tyApp
subty₀     :  ∀ n (E : ty n). sub ty (1 + n) n
subty₀     =  sub₀ tyKit
wkty       :  ∀ n. sub ty n (1 + n) → ty n → ty (1 + n)
wkty       =  wkstuff tyKit tyApp
subKit     :  kit ty ty
subKit     =  Kit X (λ n. id (ty n)) (λ n. wkty (wk_sub n tyKit))
sbst       :  ∀ n. ty (1 + n) → ty n → ty n
sbst E F   =  trav subKit E (subty₀ F)
unfld      :  ∀ n. ty (1 + n) → ty (1 + n) → ty n
unfld E F  =  sbst (E↠F) (μ E↠F)
```

Fig. 1. The structure of substitution on ty.

a given term F for all occurrences of the 0-th variable of a term E. At last, unfld unfolds a μ-redex.

Unlike monadic presentations, named presentations of terms *with holes* can be cumbersome and have limited application. Among the closest nameless but not monadic presentations are Capretta's polynomial expressions with metavariables [6]. They require proving equality of substitutions in a context. Monadic presentations of terms allow to have substitutions as part of the construction and also allow for free to have a notion of a term with a hole. For example, it can be seen that Capretta's tree expressions with metavariables have the same expressive power as monadic substitutions on polynomial trees.

7 Related work

In their seminal paper [3], Amadio and Cardelli extended the partial order on finite types to possibly infinite recursive types and showed that it is sound and complete with respect to a certain partial order on finite and infinite trees. The partial order on trees was defined by an infinite sequence of finite approximations created by truncating trees at a finite depth. Two trees were defined to be in subtree relation if and only if the partial order holds between their finite approximations at all finite depths. The time complexity of this subtyping algorithm is exponential.

The authors of [3] stated that a relation of recursive subtyping to decidable problems on automata was not known. More exactly, the word used was "well-known", which may create some space for speculation. However, shortly after, such connection was found. An efficient, $O(n^2)$-time subtyping algorithm for

recursive types was defined in [14] using regular term semantics. The algorithm works by reduction of a subtyping problem to the emptiness problem of a special automaton called *term automaton*. It was shown there that the automaton-theoretic approach can be productively applied to subtyping.

It was spelt out by the authors of [5] that the interpretation of subtyping and equality in terms of, respectively, simulation and bisimulation leads to an inference system with coinductively motivated fixpoint rules for the term language of coercions between μ-types. This coinductive view also has a straightforward application to regular languages [11, 13] given that containment (in other words, subtyping) and equivalence there correspond to simulation and bisimulation on finite automata.

From the point of view of higher-order programming, nested type definitions can be seen as instances of iteration or coiteration schemes. This view was developed in [1]. The definition of tyle is related to type definitions by *Mendler coiteration for higher ranks*, a relationship that can be investigated further in future work. In this paper, the connection is not made explicit as it would eventually require an implementation of generalised iteration and coiteration schemes in the proof assistant. In comparison, the aim of our work is to use a minimalist and standard set of tools allowing to state the soundness and completeness result without going through a more general theory.

The soundness and completeness result allows us to tell that our definition of syntactic subtyping is correct with respect to the tree semantics. In a proof assistant this is only a change in representation. Since proof search is undecidable on the universe of types in general, it is impractical, and likely impossible, to use either of the representations for efficient proof search in a prover. Instead, we can use the approach which is known as the *two-level approach* [6] or *small-scale reflection* [10].

We can implement a decision procedure for a class of propositional goals $G \in \mathsf{Prop}$ by

1. first defining a type of *codes* $\mathsf{goal} : \mathsf{Type}$ and an interpretation function $[\![\_]\!] : \mathsf{goal} \to \mathsf{Prop}$ surjective on G,
2. and then defining a decision algorithm $\mathsf{dec} : \mathsf{goal} \to \mathsf{bool}$ which can be proved sound and complete with respect to the propositional interpretation, that is,

$$\forall g : \mathsf{goal}.\ \mathsf{dec}\ g = \mathsf{true} \leftrightarrow [\![g]\!]$$

As a result, to prove $P \in G$, it is sufficient to compute $\mathsf{dec}\ g$, where g is the code for P.

Alternatively, soundness and completeness of the decision algorithms is an object of inductive type $\mathsf{decidable} : \mathsf{Prop} \to \mathsf{bool} \to \mathsf{Type}$ defined by

$$\frac{p : P}{\mathsf{dT}\ p\ :\ \mathsf{decidable}\ P\ \mathsf{true}} \qquad \frac{p : \neg P}{\mathsf{dF}\ p\ :\ \mathsf{decidable}\ P\ \mathsf{false}}$$

Thus $\mathsf{decidable}\ P\ b$ denotes the fact that provability of P is decidable by the algorithm b. As a side note, the above inductive type can be extended to account

for many-valued decision algorithms, for example, three-valued ones, where one of the values stands for the undefined result.

Common decision procedures may be based on various notions of derivative. For example, decision procedures for regular expression containment may be based on deterministic [11] or non-deterministic derivatives [13]. Both kinds of derivative can be implemented in the type theory of Coq [17, 12].

Alternative approach to subtyping: suspension monad. A practical approach to nested induction-coinduction is presented in Agda [7]. The authors provide, at the language level, a type function $\infty : \mathsf{Type} \to \mathsf{Type}$ which marks a given type as being coinductive. This type function has an interpretation as a suspension type constructor that can be used in functional languages with eager evaluation to model laziness. This interpretation is faithful since ∞ is supplied with delay and force operators $\sharp : \forall\ A.\ A \to \infty A$ and $\flat : \forall\ A.\ \infty A \to A$ respectively.

One of the immediate advantages of having the suspension monad supported by the language is efficiency. This has also a positive effect on succinctness of function definitions by recursion-corecursion since the implementation includes an improved termination checker capable of inferring termination guarantees for such function definitions. This leaves behind the more syntactically oriented termination checker of Coq.

On the other hand, without support of suspension monad in Coq, we cannot follow this approach there. This is why it is very interesting to find ways to use type theory effectively without re-engineering the implementation. Also, note that currently the suspension monad does not allow to express directly type definitions which have an outer least fixed point and an inner greatest fixed point because of the way the termination checker of Agda works.

8 Conclusions

We showed how a rather simple fold encoding pattern can be used to define a prototypical subtyping relation: μ-types without products or sums. Our study is closely related to the work of Altenkirch and Danielsson [8] who define subtyping using a suspension computation monad inspired by semantics of programming languages. The method with the suspension monad requires support in the way of dedicated programming language primitives. However, it is not always practically possible for the user of a prover or dependently typed language to amend the implementation. Here, we follow a method that allows to encode infinitary subtyping by folding an inductive relation into a coinductive one, which can be done using standard type-theoretic means. As with the suspension monad method, proving soundness corresponds to the most technically advanced part of work. The soundness argument requires to make the introduction rule for the coinductive wrapper relation parametric not only in an abstract relation R but also in *properties of* R.

It is worth noting that the presented approach of weak similarity is a natural solution to problems arising from declaring closure properties such as transitivity

in coinductive relations that were discussed in [9]. Indeed, with our definitions, infinite transitivity chains do not arise.

The paper [16] discusses an issue with the current implementation of most dependently typed systems that does not easily allow to encode bisimilarity into substitutive equality for reasoning about corecursive functions. This can be relevant to mixing induction and coinduction since mixing is essentially a fold method which, in order to work under case analysis (that is, unfolding), has to contain a reference to an abstract unfolded relation. With current implementations of dependent elimination, restoring the concrete relation behind this abstract one corresponds to a major part of work. Meanwhile, if we had elimination being able to unfold this relation automatically, this would be a clear time-saving benefit.

We can see that the traverse function trav defined in the paper is a prototype substitution strategy in the sense that, if we define substitution monads for other term languages and subtype relations of interest, the traverse function may carry some non-trivial operational meaning such as that of various matching strategies for (possibly extended) regular expressions. One of such interesting languages is the language of regular types [15], that is, recursive types with product and sum datatype constructors, which can be viewed as generalising regular expressions with non-terminating left-recursion.

Acknowledgements. I would like to thank Keiko Nakata and Niels Anders Danielsson for their help and advice regarding theorem proving and provers, and my TFP referees for their valuable remarks. The research is supported by the research fellowship EU FP7 Marie Curie IEF 253162 'SImPL'.

References

1. A. Abel, R. Matthes, and T. Uustalu. Iteration and coiteration schemes for higher-order and nested datatypes. *Theoretical Computer Science*, 333(1–2):3–66, 2005.
2. T. Altenkirch and B. Reus. Monadic presentations of lambda-terms using generalized inductive types. In *Computer Science Logic '99*, LNCS 1683, pages 453–468. Springer, 1999.
3. R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.
4. R. Bird and O. de Moor. *Algebra of programming.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.
5. M. Brandt and F. Henglein. Coinductive axiomatization of recursive type equality and subtyping. In R. Hindley, editor, *Proc. 3rd Int. Conf. on Typed Lambda Calculi and Applications (TLCA)*, volume 1210 of *LNCS*, pages 63–81, Nancy, France, 1997. Springer-Verlag.
6. V. Capretta. Certifying the Fast Fourier Transform with Coq. In *Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '01, pages 154–168, London, UK, 2001. Springer-Verlag.
7. N. A. Danielsson and T. Altenkirch. Mixing induction and coinduction, 2009. Draft.

8. N. A. Danielsson and T. Altenkirch. Subtyping, declaratively. In C. Bolduc, J. Desharnais, and B. Ktari, editors, *Mathematics of Program Construction*, volume 6120 of *Lecture Notes in Computer Science*, pages 100–118. Springer Berlin / Heidelberg, 2010.
9. V. Gapeyev, M. Y. Levin, and B. Pierce. Recursive subtyping revealed. *J. Fun. Prog.*, 12(6):511–548, 2002.
10. G. Gonthier, A. Mahboubi, and E. Tassi. A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, INRIA, 2011.
11. F. Henglein and L. Nielsen. Declarative coinductive axiomatization of regular expression containment and its computational interpretation (preliminary version). Technical Report 612, Department of Computer Science, University of Copenhagen (DIKU), February 2010.
12. V. Komendantsky. Computable partial derivatives of regular expressions, 2011. Draft. Contributed proofs are available at http://www.cs.st-andrews.ac.uk/~vk/doc/mir.v.
13. V. Komendantsky. Regular expression containment as a proof search problem. In S. Lengrand, editor, *Pre-proceedings of the International Workshop on Proof-Search in Axiomatic Theories and Type Theories (PSATTT'11)*, Wrocław, Poland, 30 July 2011.
14. D. Kozen, J. Palsberg, and M. I. Schwartzbach. Efficient recursive subtyping. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, pages 419–428, New York, NY, USA, 1993. ACM.
15. C. McBride. Type-preserving renaming and substitution, 2005. Manuscript.
16. C. McBride. Lets see how things unfold: Reconciling the infinite with the intensional (extended abstract). In A. Kurz, M. Lenisa, and A. Tarlecki, editors, *Algebra and Coalgebra in Computer Science*, volume 5728 of *Lecture Notes in Computer Science*, pages 113–126. Springer Berlin / Heidelberg, 2009.
17. T. Miyamoto. RegExp contribution to Coq, 2011.
18. K. Nakata and T. Uustalu. Resumptions, weak bisimilarity and big-step semantics for While with interactive I/O: An exercise in mixed induction-coinduction. In *Proc. Structural Operational Semantics (SOS) 2010*, 2010.
19. The Coq development team. The Coq proof assistant reference manual. http://coq.inria.fr/refman/.