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 prototypic 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 and an impredicative universe of propositions. 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 similar construction 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.

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 grammar or parse tree. 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. There are at least two different methods to encode nested fixed points in type theory that are both known as *mixed induction-coinduction*, the first is defined in [1] and the second, in [2]. 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 [3]. 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 $A \to B$ which 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 $\star$ denotes the universe of types (which is predicative, Type), $\star'$ denotes the universe of propositions (which is impredicative, Prop), and inductive and coinductive definitions are displayed in natural-deduction style with single and, respectively, double lines.

Contribution. In this paper we show that a prototypic 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 and an impredicative universe of propositions. 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 similar construction in Coq by Nakata and Uustalu recently [2], however, our case is not concerned with bisimilarity but a weaker notion of similarity that corresponds to recursive subtyping.

The motivation for this work is better understanding of termination issues in subtyping, including formalisms such as extended regular expressions, and paving the way for further extensions and provably correct practical applications.

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 of soundness and completeness of recursive types with respect to finite and infinite trees, and a description of a useful approach to decidability of subtyping in a dependently typed language. The powerful method of monadic substitution is described in Sec. 6. The *alter ego* mixed induction-coinduction method is described in Sec. 7. Finally, in Sec. 8 we give concluding remarks.

2 The fold pattern

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

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

(On a fundamental approach to fold, the reader is advised to refer to [4].) Application of foldl to an appropriately typed function f, an object a of target type and a list l of objects of source type yields an object of target type T. The result of iterative computation of f on the list l starting from a is aggregated on the left.

Let us now drop the requirement that fold starts from some object. This removes the first argument of the function f altogether. Our hypothetic fold has two dependent arguments: a function f and a collection l (which is not quite a list) of objects of S. This is in fact a description for the definition of the following function type by coinduction:

$$\begin{array}{c} f : (\forall\ E\ F.\ R\ E\ F \to E \leq F) \qquad l : E \leq_R F \\ \hline\hline E \leq F \end{array}\ \leq\text{-intro}$$

where E and F are polymorphic arguments. From its type, we can see that $\leq$-intro has two dependent arguments, f and l, and yields an object (in fact, a proof) inhabiting a particular case of relation. The function $\leq$-intro is the constructor of the coinductive relation $\leq$. 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 a coercion from R to $\leq$. 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 $\leq_R$ is alike a type of finite list of certain abstract, possibly infinite objects. The codomain of $\leq$-intro is a type of infinite object in which we collapse all the infinite components of the argument l. The object in the codomain is coinductive, and it is only thanks to the premisses of $\leq$-intro that 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, 6]:

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

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 $E \rightarrowtriangle F$. This definition neither has products or sums that are needed for practical programming. However, we do not consider product or sum types in this schematic implementation because their treatment is alike that of $\rightarrowtriangle$, see, e.g., [6]. Moreover, we choose to replace named variables in the definition by nameless de Brujin variables, which yields an equivalent and yet more tangible construction.

Now, our working definition of *recursive types* is by induction as follows:

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

$$\frac{}{\bot : \mathsf{ty}\ n} \qquad \frac{}{\top : \mathsf{ty}\ n} \qquad \frac{i : \mathcal{I}_n}{X_i : \mathsf{ty}\ n} \qquad \frac{E : \mathsf{ty}\ n \qquad F : \mathsf{ty}\ n}{E \rightarrowtriangle F : \mathsf{ty}\ n}$$

$$\frac{E : \mathsf{ty}\ (1+n) \qquad F : \mathsf{ty}\ (1+n)}{\mu\ E \rightarrowtriangle F : \mathsf{ty}\ n}$$

where the constructors are, respectively, the empty type $\bot$, the unit type $\top$, a variable, the function type constructor $\to$ and the least fixed point arrow type constructor μ _ $\to$ _. The dependent type $\mathcal{I}_n$ represents the first n natural numbers, and therefore an object i of type $\mathcal{I}_n$ is a pair consisting of a natural number m and a proof of $m < n$.

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

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

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

Intuitively, trees are views of μ-types unfolded *ad infinitum.* We denote the tree corresponding to a type E by $[\![E]\!]$. The straightforward subtree relation $\mathsf{tle}\ n$ on $\mathsf{tree}\ n$ is denoted by $\leq^\infty$ (omitting the implicit argument n):

$$\mathsf{tle}\ n : \mathsf{tree}\ n \to \mathsf{tree}\ n \to \star'$$

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

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 [6]). 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 a remark made by Amadio and Cardelli in [6], 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 in another. We use the impredicative universe of propositions that we denote by $\star'$, which is needed for the proof of soundness and completeness. 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):

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

$$\frac{}{\bot \leq_R E} \qquad \frac{}{E \leq_R \top} \qquad \frac{R\ E\ F \qquad R\ G\ H}{F \mathrel{-\!\!\triangleright} G \leq_R E \mathrel{-\!\!\triangleright} H} \qquad \frac{}{\mu\ E \mathrel{-\!\!\triangleright} F \leq_R \mathsf{unfld}\ E\ F}$$

$$\frac{}{\mathsf{unfld}\ E\ F \leq_R \mu\ E \mathrel{-\!\!\triangleright} F} \qquad \frac{}{E \leq_R E} \qquad \frac{E \leq_R F \qquad F \leq_R G}{E \leq_R G}$$

where unfld is the operation that unfolds a μ-redex by substituting the variable 0 in the term $E {\rightarrow\!\!\triangleright} F$ with the term $\mu\ E {\rightarrow\!\!\triangleright} F$ 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:

$$\mathsf{tyle}\ n : \mathsf{ty}\ n \to \mathsf{ty}\ n \to \star'$$

We denote $\mathsf{tyle}\ n\ E\ F$ by $E \leq F$.

$$\frac{\forall\ E\ F.\ R\ E\ F \to E \leq F \qquad E \leq_R F}{E \leq F}$$

The only introduction rule for $\leq$ has two hypotheses, namely, that R is 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

Main Theorem (Soundness and completeness).

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

The "only if" direction (completeness) follows by a straightforward application of the coinduction principle. For the "if" direction (soundness), we define the weak head normal form of the relation $\leq^\infty$ and solve the problem via this notion, which is a common workaround helping to ensure syntactic guardedness of the proof [1, 2].

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. Besides, both representations are propositional. Since proof search is undecidable on the universe of propositions 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* [8] or *small-scale reflection* [9].

We can implement a decision procedure for a class of propositional goals $G \in \star'$ by

1. first defining a type of *codes* $\mathsf{goal} : \star$ and an interpretation function $[\![\_]\!] : \mathsf{goal} \to \star'$ 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 algorithm is an object of inductive type $\mathsf{decidable} : \star' \to \mathsf{bool} \to \star$ defined by

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

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 partial 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 subtyping may be based on deterministic [10] or non-deterministic derivatives [11]. Both kinds of derivative can be implemented in the type theory of Coq [12, 13].

6 Monadic substitution

We implemented in Coq a generic notion of symbolic substitution introduced in [14] for untyped lambda terms. It is based on the notion of *universe of types*, that is, a function space $A \to \star$ where A can be any given type and $\star$ is the polymorphic type of all types. A is said to *index* the type $\star$. For effective indexing, the index type should be countable, and for that, it suffices to consider the type $\mathbb{N}$ of natural numbers. Following McBride [15], we call the resulting type of universe *stuff*:

$$\mathsf{stuff} : \star$$

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

For a given n, the intended meaning of $\mathsf{stuff}\ n$ is *stuff with n variables.*

In the foundation of the method, there is a type of monadic structure called *kit* [15]:

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

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

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,

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

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

In order to establish compositionality on substitutions, we define applicative structure on substitutions which is called subApp:

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

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

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.

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

Here, lift_sub is a function that lifts a substitution to the next order, that is, shifts the indices of active variables in the substitution by $+1$. This function has type

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

The traverse function allows to define an instance of the applicative structure on ty that we call tyApp, in Figure 6. 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 using the endomorphism constructor stuffKit. The function subty_0 substitutes the 0-th variable with a given μ-type E. It is defined using a generic substitution sub_0. The type of sub_0 is

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

Weakening is specialised on ty by the function wkty. The function sbst is capture-avoiding substitution of 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 [8]. 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.

$$\begin{array}{lcl}
\mathsf{tyApp} & : & \mathsf{subApp\ ty} \\
\mathsf{tyApp} & = & \mathsf{SubApp}\ X\ \mathsf{trav} \\
\mathsf{tyKit} & : & \mathsf{kit\ ty\ ty} \\
\mathsf{tyKit} & = & \mathsf{stuffKit\ tyApp} \\
\mathsf{subty}_0 & : & \forall\ n\ (E : \mathsf{ty}\ n).\ \mathsf{sub\ ty}\ (1+n)\ n \\
\mathsf{subty}_0 & = & \mathsf{sub}_0\ \mathsf{tyKit} \\
\mathsf{wkty} & : & \forall\ n.\ \mathsf{sub\ ty}\ n\ (1+n) \to \mathsf{ty}\ n \to \mathsf{ty}\ (1+n) \\
\mathsf{wkty} & = & \mathsf{wkstuff\ tyKit\ tyApp} \\
\mathsf{subKit} & : & \mathsf{kit\ ty\ ty} \\
\mathsf{subKit} & = & \mathsf{Kit\ var}\ (\lambda\ n.\ \mathsf{id}\ (\mathsf{ty}\ n))\ (\lambda\ n.\ \mathsf{wkty}\ (\mathsf{wk\_sub}\ n\ \mathsf{tyKit})) \\
\mathsf{sbst} & : & \forall\ n.\ \mathsf{ty}\ (1+n) \to \mathsf{ty}\ n \to \mathsf{ty}\ n \\
\mathsf{sbst}\ E\ F & = & \mathsf{trav\ subKit}\ E\ (\mathsf{subty}_0\ F) \\
\mathsf{unfld} & : & \forall\ n.\ \mathsf{ty}\ (1+n) \to \mathsf{ty}\ (1+n) \to \mathsf{ty}\ n \\
\mathsf{unfld}\ E\ F & = & \mathsf{sbst}\ (E \multimap F)\ (\mu\ E \multimap F)
\end{array}$$

Fig. 1. The structure of substitution on ty.

7 Alternative approach to subtyping: suspension monad

A practical approach to nested induction-coinduction is presented in Agda [16]. The authors provide, at the language level, a type function $\infty : \star \to \star$ 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 prototypic subtyping relation: μ-types without products or sums. Our study is closely related to the work of Altenkirch and Danielsson [1] who define subtyping using a suspension computation monad inspired by semantics of programming languages. The method with the suspension monad turns out to be inapplicable outside the special setting of [1]. 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. 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 [7]. Indeed, with our definitions, infinite transitivity chains do not arise.

The paper [17] discusses an issue with the current implementation of the 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. The research is supported by the research fellowship EU FP7 Marie Curie IEF 253162 'SImPL'.

References

1. Danielsson, N.A., Altenkirch, T.: Subtyping, declaratively. In Bolduc, C., Desharnais, J., Ktari, B., eds.: Mathematics of Program Construction. Volume 6120 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2010) 100–118 doi:10.1007/978-3-642-13321-3_8.
2. Nakata, K., Uustalu, T.: 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)
3. The Coq development team: The Coq proof assistant reference manual. http://coq.inria.fr/refman/
4. Bird, R., de Moor, O.: Algebra of programming. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1997)

5. Brandt, M., Henglein, F.: Coinductive axiomatization of recursive type equality and subtyping. In Hindley, R., ed.: Proc. 3rd Int. Conf. on Typed Lambda Calculi and Applications (TLCA). Volume 1210 of LNCS., Nancy, France, Springer-Verlag (1997) 63–81
6. Amadio, R.M., Cardelli, L.: Subtyping recursive types. ACM Transactions on Programming Languages and Systems **15**(4) (1993) 575–631
7. Gapeyev, V., Levin, M.Y., Pierce, B.: Recursive subtyping revealed. J. Fun. Prog. **12**(6) (2002) 511–548
8. Capretta, V.: Certifying the Fast Fourier Transform with Coq. In: Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics. TPHOLs '01, London, UK, Springer-Verlag (2001) 154–168
9. Gonthier, G., Mahboubi, A., Tassi, E.: A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, INRIA (2011)
10. Henglein, F., Nielsen, L.: 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)
11. Komendantsky, V.: Regular expression containment proofs by construction of Mirkin prebases (2011) Submitted.
12. Miyamoto, T.: RegExp contribution to Coq (2011)
13. Komendantsky, V.: Mirkin derivatives of a regular expression (extended abstract) (2011) Submitted.
14. Altenkirch, T., Reus, B.: Monadic presentations of lambda-terms using generalized inductive types. In: Computer Science Logic '99. LNCS 1683, Springer (1999) 453–468
15. McBride, C.: Type-preserving renaming and substitution (2005) Manuscript.
16. Danielsson, N.A., Altenkirch, T.: Mixing induction and coinduction (2009) Draft.
17. McBride, C.: Lets see how things unfold: Reconciling the infinite with the intensional (extended abstract). In Kurz, A., Lenisa, M., Tarlecki, A., eds.: Algebra and Coalgebra in Computer Science. Volume 5728 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2009) 113–126