Application of monadic substitution to recursive type containment

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

Abstract

In this paper, we present a computer-checked, constructive soundness and completeness result for prototypic recursive type containment with respect to containment of non-wellfounded (finite or infinite) trees. The central role is played by formalisation of substitution of recursive types as a monad, with a traverse function implementing a strategy for potentially infinite recursive unfolding. In the rigorous setting of constructive type theory, the very definition of subtyping should be scrutinised to allow a prover to accept it. As a solution, we adapt the method of weak bisimilarity by mixed induction-coinduction recently introduced to Coq by Nakata and Uustalu. However, our setting is different in that we work with a notion of weak similarity that corresponds to subtyping. We accomplish the task of representing infinitary subtyping in the Calculus of (Co-)Inductive Constructions. Our technique does not require extensions of the Calculus and therefore can be ported to other dependently typed languages.

1 Introduction

Paper-and-pen proofs of soundness and completeness of recursive type containment (subtyping) with respect to models based on tree structures have been known since the early 1990's result of Amadio and Cardelli [2]. Over the years, their technique has been gradually refined and slightly simplified in presentation [3, 5]. Recently there has been a renewed interest in subtyping [4] which was also stimulated by progress in the area of dependently typed languages such as Agda or Coq. It becomes possible to capture subtyping in such languages, although traditional proofs are not necessarily straightforward to implement. Implementations rather take into account what is practically possible in type theory and what is not. Especially, this concerns notions of rewriting required for possibly infinite unfolding of recursive types.

Here we show that a prototypic subtype relation that can neither be defined as a least fixed point nor as a greatest fixed point alone can nevertheless be defined with inductive and coinductive types and an impredicative universe of propositions. One of the most interesting points is that the definition proceeds alike a fold in functional programming, although a rather unusual one, that is not applied to any starting object. This is an illustration of a very general programming pattern allowing to alternate least and greatest fixed points. The soundness and completeness result stated in the paper, apart from guaranteeing correctness of our definition of subtyping, has a technical significance of providing a mechanism of weak head normal forms that seems indispensable in situations when we need to "evaluate" a subtyping statement.

One possible application for subtyping in a language of μ-types that motivates our work is coercive containment of regular expressions [6], a paradigm based on understanding regular expressions as types and concerned with a proof-theoretic interpretation of containment of languages denoted by regular expressions. This also motivates the choice of monadic presentations of μ-types. Monadic presentations [1, 8, 7] are usually employed to study type-preserving substitutions. In the case of coercions, regular expressions themselves are modified, but words in the

underlying language of the source of the coercion are preserved in the language of the target of the coercion. So, in fact, we might speak of type-coercing rather than type-preserving substitutions. In this paper though, coercions are not yet spelt out in the object language, and we have to study how exactly we can extract the coercive content from proofs of subtyping statements.

Throughout the paper, we use a human-oriented type theoretic notation for the meta-language in which we define recursive types (the object language): $\star$ denotes the universe of types (which is predicative), $\star'$ denotes the universe of propositions (which is impredicative), and inductive and coinductive definitions are displayed in natural-deduction style with single and, respectively, double lines.

2 Recursive types

Below is the *traditional* definition of the set of *recursive types*. It is commonly defined by the following grammar [3, 2]:

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

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 traditional omission of products and sums that are needed for practical programming is due to the fact that their treatment is very similar to that of $\twoheadrightarrow$, see, e.g., [2]. The only true problem, however, is concerned with the μ operator.

For the purpose of type-theoretic encoding, we choose another, equivalent and yet more tangible definition of recursive types, where named variables are replaced by nameless de Brujin variables. We represent recursive types in Coq [10], by induction as follows:

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

$$\frac{}{\bot : \mathsf{ty}\ n} \qquad \frac{}{\top : \mathsf{ty}\ n} \qquad \frac{i : \mathcal{I}_n}{X_i : \mathsf{ty}\ n} \qquad \frac{E : \mathsf{ty}\ n \qquad F : \mathsf{ty}\ n}{E \twoheadrightarrow F : \mathsf{ty}\ n} \qquad \frac{E : \mathsf{ty}\ (1+n) \qquad F : \mathsf{ty}\ (1+n)}{\mu\ E \twoheadrightarrow F : \mathsf{ty}\ n}$$

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 $\mu\ \_ \to \_$. 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 $m < n$. An interesting point is, if we compare $\mathcal{I}_n$ with the algebraic type of finite number employed by McBride in [8], that the relation $<$ on natural numbers can be expressed as a boolean relation, which allows to encode the whole proof of $m < n$ as a boolean value, and is very handy in case analysis.

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

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

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

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 $\leq^\infty$ (by noting that we can infer the implicit argument n):

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

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

Note that the subtree relation is contravariant in the first argument of $\rightarrowtriangle^\infty$ and covariant in the second. Two recursive types are in the subtype relation when their potentially infinite unfoldings are in the subtree relation. This is where monadic term presentations come into play. Instead of trying to define inductive limits of sequences of approximations of unfoldings of recursive types (as traditionally done in paper-and-pen proofs, e.g., in [2]), we encapsulate unfolding *ad infinitum* by relying on expressivity of dependent types of the CIC which allow to define this powerful monadic presentation structure. The point here, similar to a remark made by Amadio and Cardelli in [2], is that unfoldings of recursive types are *regular* trees, which we treat using a mix of induction and coinduction.

3 Monadic substitution

We implemented in Coq a generic notion of symbolic substitution introduced in [1] 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 [8], we call the resulting type of universe *stuff*:

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

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* [8, 7]:

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

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

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,

$$\mathsf{sub} : \mathsf{stuff} \to \mathbb{N} \to \mathbb{N} \to \star \qquad\qquad \mathsf{sub}\ T\ m\ n = m\text{-}\mathsf{tuple}\ (T\ n)$$

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

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

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

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.

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

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

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

What is the concrete structure of substitution on ty and why kit is indeed a monad will be explained in an extended version of this paper. A definition from the concrete structure of substitution, the function unfld that unfolds a μ-redex, is used in the next section.

4 Weak similarity by mixed induction-coinduction

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):

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

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

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

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, and the soundness argument would not work.

Next, we fold the inductive relation $\leq_R$ in a coinductive relation $\mathsf{tyle}\ n$ and produce a weak similarity (denoting $\mathsf{tyle}\ n\ E\ F$ by $E \leq F$):

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

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

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 (WHNF) of the relation $\leq^{\infty}$ and solve the problem via WHNFs, which is a common workaround helping with ensuring syntactic guardedness of the proof [4, 9].

6 Conclusions

We showed how monadic substitution can be used to define a rather simple case of subtyping relation: μ-types without products or sums. Our study is closely related to the work of Altenkirch and Danielsson [4] who define subtyping using a suspension computation monad inspired by semantics of programming languages. The method with the suspension monad turns out not to be immediately applicable outside the special setting of [4] because of the way the termination checker works in Coq at the moment. Namely, there is no united checker for recursive and corecursive functions. Instead, there appear to be two independent mechanisms, one for checking structurality of recursive functions, and another for checking guardedness of corecursive ones. This leads to difficulties with mixed induction-coinduction. In fact, it would have been very beneficial for the termination checker of Coq if the suspension monad was implemented in it since that would require unifying the two parts together at last.

Here, we resolve these difficulties on top of the prover, following a very general method that allows to encode infinitary subtyping by folding the inductive relation $\leq_R$ into the coinductive $\leq$. Also, we remark 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 [5]. Indeed, with our definitions, infinite transitivity chains do not arise.

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] 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.

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

[3] 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.

[4] 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. doi:10.1007/978-3-642-13321-3_8.

[5] V. Gapeyev, M. Y. Levin, and B. Pierce. Recursive subtyping revealed. *J. Fun. Prog.*, 12(6):511–548, 2002.

[6] F. Henglein and L. Nielsen. Regular Expression Containment: Coinductive Axiomatization and Computational Interpretation. In *Proc. 38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, January, 2011.

[7] C. Keller and T. Altenkirch. Hereditary substitutions for simple types, formalized. In *Proceedings of the third ACM SIGPLAN workshop on Mathematically structured functional programming*, MSFP '10, pages 3–10. ACM, 2010.

[8] C. McBride. Type-preserving renaming and substitution, 2005. Manuscript.

[9] 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.

[10] The Coq development team. The Coq proof assistant reference manual. `http://coq.inria.fr/refman/`.