Subtyping by Folding an Inductive Relation into a Coinductive One

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 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,

R. Peña and R. Page (Eds.): TFP 2011, LNCS 7193, pp. 17–32, 2012.

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