Subtyping, Declaratively
An Exercise in Mixed Induction and Coinduction

Nils Anders Danielsson and Thorsten Altenkirch

University of Nottingham

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

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

The developments presented in the paper have been mechanised using Agda, a dependently typed programming language and proof assistant.

1 Introduction

Coinduction and corecursion are useful techniques for defining and reasoning about things which are potentially infinite, including streams and other (potentially) infinite data types (Coquand 1994; Giménez 1996; Turner 2004), process congruences (Milner 1990), congruences for functional programs (Gordon 1999), closures (Milner and Tofte 1991), semantics for divergence of programs (Cousot and Cousot 1992; Hughes and Moran 1995; Leroy and Grall 2009; Nakata and Uustalu 2009), and subtyping relations for recursive types (Brandt and Henglein 1998; Gapeyev et al. 2002).

However, the use of coinduction can lead to values which are "too infinite". For instance, a non-trivial binary relation defined as a coinductive inference system cannot include the rule of transitivity, because a coinductive reading of transitivity would imply that every element is related to every other (to see this, build an infinite derivation consisting solely of uses of transitivity). As pointed out by Gapeyev et al. (2002) this is unfortunate, because without transitivity, conceptually unrelated rules may have to be merged or otherwise modified in order to ensure that transitivity can be proved as a derived property. Gapeyev et al. give the example of subtyping for records, where a dedicated rule of transitivity ensures that one can give separate rules for depth subtyping (which states that a record field type can be replaced by a subtype), width subtyping (which states that new fields can be added to a record), and permutation of record fields.

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

We propose a solution to this problem. The problem stems from a *coinductive* reading of transitivity, and it can be solved by reading the rule of transitivity *inductively*, and only using coinduction where it is necessary. We illustrate this idea by using mixed induction and coinduction to define a subtyping relation for recursive types; such relations have been studied repeatedly in the past (Amadio and Cardelli 1993; Kozen et al. 1995; Brandt and Henglein 1998, and others). The rule which defines when a function type is a subtype of another is defined coinductively, following Brandt and Henglein (1998) and Gapeyev et al. (2002), while the rule of transitivity is defined inductively.

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

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

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

2 Induction and Coinduction

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

2.1 Induction

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

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