Coinductive Axiomatization of Recursive Type Equality and Subtyping*

Michael Brandt and Fritz Henglein

DIKU, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen East, Denmark, Email: {mick,henglein}@diku.dk

Abstract. We present new sound and complete axiomatizations of type equality and subtype inequality for a first-order type language with regular recursive types. The rules are motivated by coinductive characterizations of type containment and type equality via simulation and bisimulation, respectively. The main novelty of the axiomatization is the *fixpoint rule* (or *coinduction principle*), which has the form

$$\frac{A, P \vdash P}{A \vdash P}$$

where P is either a type equality $\tau = \tau'$ or type containment $\tau \leq \tau'$. We define what it means for a proof (formal derivation) to be formally *contractive* and show that the fixpoint rule is *sound* if the proof of the premise $A, P \vdash P$ is contractive. (A proof of $A, P \vdash P$ using the assumption axiom is, of course, *not* contractive.) The fixpoint rule thus allows us to capture a coinductive relation in the fundamentally inductive framework of inference systems.

The new axiomatizations are "leaner" than previous axiomatizations, particularly so for type containment since no separate axiomatization of type equality is required, as in Amadio and Cardelli's axiomatization. They give rise to a natural operational interpretation of proofs as *coercions*. In particular, the fixpoint rule corresponds to *definition by recursion*. Finally, the axiomatization is closely related to (known) efficient algorithms for deciding type equality and type containment. These can be modified to not only *decide* type equality and type containment, but also construct proofs in our axiomatizations efficiently. In connection with the operational interpretation of proofs as coercions this gives *efficient* ($O(n^2)$ time) algorithms for constructing *efficient* coercions from a type to any of its supertypes or isomorphic types.

1 Introduction

The simply typed λ-calculus is paradigmatic for both type inference for programming languages and the Curry-Howard isomorphism. Whereas adding recursive

* This research was partially supported by the Danish Research Council, Project *DART*.

types destroys its strong normalization property and its logical soundness under the Curry-Howard interpretation, recursive types preserve and extend the "well-typed programs don't go wrong" interpretation of λ-terms. To use recursive types it is necessary to add the rule

$$\frac{A \vdash e : \tau \qquad \vdash \tau = \tau'}{A \vdash e : \tau'} \qquad (\textsc{Equal})$$

for simple typing with recursive types [CC91] or

$$\frac{A \vdash e : \tau \qquad \vdash \tau \leq \tau'}{A \vdash e : \tau'} \qquad (\textsc{Subtype})$$

for simple subtyping with recursive types [AC91, AC93]. The question, now, is when two recursive types are equal or in the subtyping relation. This is what we study in this paper.

1.1 Recursive Types

Definition 1. The *recursive types (in canonical form)* μTp are generated by the grammar

$$\tau \equiv \bot \mid \top \mid \alpha \mid \tau_1 \to \tau_2 \mid \mu\alpha.(\tau_1 \to \tau_2)$$

where α ranges over an infinite set TVar of *type variables*, μ binds its type variable, α-congruent recursive types are identified, and in every $\mu\alpha.\tau$ the bound variable α occurs freely in τ.

Intuitively, $\mu\alpha.\tau$ denotes the recursive type defined by the type equation $\alpha = \tau$ (note that α occurs in τ), $\bot$ is contained in all other types, and $\top$ contains all other types. Our results extend to other types and type constructors such as product and sum. We let τ, σ range over recursive types and write $\mathrm{fv}(\tau)$ for the set of free type variables in τ.

1.2 Regular Trees

We define $\mathrm{Tree}(\tau)$ to be the regular (possibly infinite) tree obtained by completely unfolding all occurrences of $\mu\alpha.\tau$ to $\tau[\mu\alpha.\tau]$. (For a precise definition of $\mathrm{Tree}(\tau)$, regular trees and their properties see [Cou83, CC91, AC93].)

Henceforth we shall assume that all trees are over the ranked alphabet $\{\bot^0, \to^2, \top^0\} \cup \{\alpha^0 : \alpha \in \mathrm{TVar}\}$ of *labels*, which are ordered by the reflexive-transitive closure of $\bot < \overset{\to}{\underset{\alpha}{}} < \top$.

We can define *depth-k lower and upper approximations* $T|_k$ and $T|^k$ of a tree T as follows:

$$\begin{array}{rr}
T|_0 = \bot & T|^0 = \top \\
(T' \to T'')|_{k+1} = T'|^k \to T''|_k & (T' \to T'')|^{k+1} = T'|_k \to T''|^k \\
\bot|_{k+1} = \bot & \bot|^{k+1} = \bot \\
\top|_{k+1} = \top & \top|^{k+1} = \top \\
\alpha|_{k+1} = \alpha & \alpha|^{k+1} = \alpha
\end{array}$$

For tree T, $\mathcal{L}(T)$, the *label* of T, is the label of the root node of T. The label of a recursive type τ is the label of the tree it denotes: $\mathcal{L}(\tau) = \mathcal{L}(\mathrm{Tree}(\tau))$.