

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

^{*} To appear in Proc. 3rd Int’l Conference on Typed Lambda-Calculi and Applications (TLCA), Nancy, France, April 2-4, 1997, Springer-Verlag, Lecture Notes in Computer Science. 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

$$\frac{A \vdash e : \tau \quad \vdash \tau = \tau'}{A \vdash e : \tau'} \quad (\text{EQUAL})$$

for simple typing with recursive types [CC91] or

$$\frac{A \vdash e : \tau \quad \vdash \tau \leq \tau'}{A \vdash e : \tau'} \quad (\text{SUBTYPE})$$

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

$$\tau \equiv \perp \mid \top \mid \alpha \mid \tau_1 \rightarrow \tau_2 \mid \mu\alpha.(\tau_1 \rightarrow \tau_2)$$

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), \perp 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 $\text{fv}(\tau)$ for the set of free type variables in τ .

1.2 Regular Trees

We define $\text{Tree}(\tau)$ 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 $\text{Tree}(\tau)$, regular trees and their properties see [Cou83, CC91, AC93].)

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

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

$$\begin{array}{ll} T|_0 = \perp & T|_0^k = \top \\ (T' \rightarrow T'')|_{k+1} = T'|_k \rightarrow T''|_k & (T' \rightarrow T'')|_{k+1}^k = T'|_k \rightarrow T''|_k \\ \perp|_{k+1} = \perp & \perp|_{k+1}^k = \perp \\ \top|_{k+1} = \top & \top|_{k+1}^k = \top \\ \alpha|_{k+1} = \alpha & \alpha|_{k+1}^k = \alpha \end{array}$$

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: $\mathcal{L}(\tau) = \mathcal{L}(\text{Tree}(\tau))$.

1.3 Recursive Type Equality

Cardone and Coppo [CC91] show the following results about recursive type equality (for types without \top):

- Interpreting recursive types as ideals in a universal domain, τ, τ' are *semantically equivalent* (denote the same ideal) if and only if $\text{Tree}(\tau) = \text{Tree}(\tau')$.
- *Weak type equality*, the congruence generated by axiom FOLD/UNFOLD in Figure 1, is properly weaker than semantic type equivalence.
- The principal typing property in the sense of Hindley [Hin69] and Ben-Yelles [BY79] extends to simple typing with recursive types if type equality in Rule (EQUAL) is taken to be semantic type equivalence, yet it breaks if it is defined as weak equality.

Let us write $\tau \approx \tau'$ if $\text{Tree}(\tau) = \text{Tree}(\tau')$. Axiomatizations of \approx are given by Amadio/Cardelli [AC91] and Ariola/Klop [AK95]. It is clear, however, that this kind of axiomatization has been known for a long time; see for example Salomaa [Sal66] and Milner [Mil84].

All these axiomatizations are a variant of the inference system presented in Figure 1.² (In Rule CONTRACT, recursive type τ is contractive in type variable α if α occurs in τ only under \rightarrow , if at all.)

$$\begin{array}{c}
 \vdash \tau = \tau \qquad \frac{\vdash \tau = \tau' \quad \vdash \tau' = \tau''}{\vdash \tau = \tau''} \qquad \frac{\vdash \tau' = \tau}{\vdash \tau = \tau'} \\
 \\
 \frac{\vdash \tau = \tau' \quad \vdash \sigma = \sigma'}{\vdash \tau \rightarrow \sigma = \tau' \rightarrow \sigma'} \qquad \frac{\vdash \tau = \tau'}{\vdash \mu\alpha.\tau = \mu\alpha.\tau'} \quad (\mu\text{-COMPAT}) \\
 \\
 \vdash \mu\alpha.\tau = \tau[\mu\alpha.\tau/\alpha] \qquad (\text{FOLD/UNFOLD}) \\
 \\
 \frac{\vdash \tau_1 = \tau[\tau_1/\alpha] \quad \vdash \tau_2 = \tau[\tau_2/\alpha]}{\vdash \tau_1 = \tau_2} \quad (\tau \text{ contractive in } \alpha) \quad (\text{CONTRACT})
 \end{array}$$

Fig. 1. Classical axiomatization of recursive type equality

² Instead of Rule CONTRACT Ariola and Klop [AK95] use the equivalent rule

$$\frac{\vdash \tau_1 = \tau[\tau_1/\alpha]}{\mu\alpha.\tau = \tau_1} \quad \tau \text{ contractive in } \alpha.$$

$$\begin{array}{c}
\Gamma \vdash \perp \leq \tau \qquad \qquad \qquad \Gamma \vdash \tau \leq \top \\
\\
\Gamma \vdash \tau \leq \tau \qquad \qquad \qquad \frac{\Gamma \vdash \tau \leq \tau' \quad \Gamma \vdash \tau' \leq \tau''}{\Gamma \vdash \tau \leq \tau''} \\
\\
\Gamma, \tau \leq \tau', \Gamma' \vdash \tau \leq \tau' \qquad \qquad \frac{\Gamma \vdash \tau = \tau'}{\Gamma \vdash \tau \leq \tau'} \\
\\
\frac{\Gamma \vdash \tau' \leq \tau \quad \Gamma \vdash \sigma \leq \sigma'}{\Gamma \vdash \tau \rightarrow \sigma \leq \tau' \rightarrow \sigma'} \qquad \frac{\Gamma, \alpha \leq \beta \vdash \tau \leq \sigma}{\Gamma \vdash \mu\alpha.\tau \leq \mu\beta.\sigma} \quad (\alpha, \beta \text{ not free in } \sigma, \tau)
\end{array}$$

Fig. 2. Amadio/Cardelli axiomatization of subtyping for recursive types

1.4 Recursive Subtyping

Amadio and Cardelli [AC93] extend the standard contravariant structural subtyping relation on μ -free types (to be thought of as finite trees) defined by

$$\begin{array}{c}
\perp \leq_{\text{fin}} T \quad (\perp_{\text{fin}}) \qquad \qquad T \leq_{\text{fin}} \top \quad (\top_{\text{fin}}) \\
\\
T \leq_{\text{fin}} T \quad (\text{REF}_{\text{fin}}) \qquad \qquad \frac{S_1 \leq_{\text{fin}} T_1 \quad T_2 \leq_{\text{fin}} S_2}{T_1 \rightarrow T_2 \leq_{\text{fin}} S_1 \rightarrow S_2} \quad (\text{ARROW}_{\text{fin}})
\end{array}$$

in a natural fashion to infinite trees.

Definition 2 (Amadio/Cardelli subtype relation). Let τ, σ be recursive types. Define $\tau \leq_{\text{AC}} \sigma$ if $\text{Tree}(\tau)|_k \leq_{\text{fin}} \text{Tree}(\sigma)|_k$ for all $k \in \mathbb{N}_0$.

In the definition we could replace the lower approximations by upper approximations since both induce the same subtyping relation:

Proposition 3. $T|_k \leq_{\text{fin}} T'|_k$ if and only if $T|_k \leq_{\text{fin}} T'|_k$.

Amadio and Cardelli build on the axiomatization of type equality in Figure 1 and give a sound and complete axiomatization of \leq_{AC} in [AC93], shown in Figure 2.

1.5 The New Axiomatizations

In this paper we show that the Amadio/Cardelli subtype relation can be directly axiomatized by the inference system in Figure 3.

The most noteworthy aspect of the system is rule ARROW/FIX for proving inequality between function types. It can be understood as the composition of the two separate rules

$$\frac{A \vdash \sigma_1 \leq \tau_1 \quad A \vdash \tau_2 \leq \sigma_2}{A \vdash \tau_1 \rightarrow \tau_2 \leq \sigma_1 \rightarrow \sigma_2} \quad (\text{ARROW}) \qquad \frac{A, \tau \leq \tau' \vdash \tau \leq \tau'}{A \vdash \tau \leq \tau'} \quad (\text{FIX})$$

$$\begin{array}{l}
A \vdash \perp \leq \tau \quad (\perp) \qquad\qquad\qquad A \vdash \tau \leq \top \quad (\top) \\
\\
A \vdash \tau \leq \tau \quad (\text{REF}) \qquad\qquad\qquad \frac{A \vdash \tau \leq \delta \quad A \vdash \delta \leq \sigma}{A \vdash \tau \leq \sigma} \quad (\text{TRANS}) \\
\\
A \vdash \mu\alpha.\tau \leq \tau[\mu\alpha.\tau/\alpha] \quad (\text{UNFOLD}) \qquad\qquad A \vdash \tau[\mu\alpha.\tau/\alpha] \leq \mu\alpha.\tau \quad (\text{FOLD}) \\
\\
A, \tau \leq \sigma, A' \vdash \tau \leq \sigma \quad (\text{HYP}) \\
\\
\frac{A, \tau_1 \rightarrow \tau_2 \leq \sigma_1 \rightarrow \sigma_2 \vdash \sigma_1 \leq \tau_1 \quad A, \tau_1 \rightarrow \tau_2 \leq \sigma_1 \rightarrow \sigma_2 \vdash \tau_2 \leq \sigma_2}{A \vdash \tau_1 \rightarrow \tau_2 \leq \sigma_1 \rightarrow \sigma_2} \quad (\text{ARROW/FIX})
\end{array}$$

Fig. 3. Coinductive axiomatization of Amadio/Cardelli subtyping

where the premise of obviously “dangerous” Rule FIX must be proved by Rule ARROW. Rule FIX says that we may actually use as a hypothesis what we want to prove when trying to prove it. We are just not allowed to use it “right away”!

The system is *sound and complete* for Amadio/Cardelli subtyping: $\vdash \tau \leq \tau'$ if and only if $\tau \leq_{\text{AC}} \tau'$. Because of Rule ARROW/FIX, more specifically the part corresponding to Rule FIX, soundness is actually a tricky issue. The proof is accomplished by giving sequents a level stratified interpretation. Completeness is shown by exhibiting an algorithm that builds a derivation for given (τ, τ') and succeeds whenever $\tau \leq_{\text{AC}} \tau'$. The crucial part here is showing that the algorithm terminates. The algorithm not only *decides* whether $\vdash \tau \leq \tau'$, but also returns an explicit proof. (It is relatively easy to see that the algorithm can be implemented in time $O(n^2)$ where n is the number of symbols in the two input types, but this is not elaborated in this paper. See [Car93, KPS93, KPS95] for efficient algorithms for deciding recursive subtyping.)

Given our “innate” axiomatization of \leq_{AC} by \leq in Figure 3 the semantic type equivalence \approx can now be *defined* in terms of subtyping since $\tau \approx \tau'$ if and only if $\tau \leq_{\text{AC}} \tau'$ and $\tau' \leq_{\text{AC}} \tau$ if and only if $\vdash \tau \leq \tau'$ and $\vdash \tau' \leq \tau$. Alternatively, we can provide a direct axiomatization of \approx , see Figure 4. Note that it requires neither Rule CONTRACT nor Rule μ -COMPAT, which are difficult to interpret denotationally and operationally.

The above results for subtyping are presented in Section 2. The corresponding results for type equality are analogous; they are omitted for space reasons.

1.6 Proofs as Programs: Subtyping/Type Equality Proofs as Coercions

The point of our coinductive axiomatizations is not only to support direct coinductive reasoning, but also to provide a natural foundation for a proof theory

$$\begin{array}{c}
A, \tau = \tau', A' \vdash \tau = \tau' \quad A \vdash \tau = \tau \\
\\
\frac{A \vdash \tau = \tau'}{A \vdash \tau' = \tau} \quad \frac{A \vdash \tau = \tau' \quad A \vdash \tau' = \tau''}{A \vdash \tau = \tau''} \\
\\
A \vdash \mu\alpha. \tau = \tau[\mu\alpha. \tau/\alpha] \\
\\
\frac{A, \tau \rightarrow \tau' = \sigma \rightarrow \sigma' \vdash \tau = \sigma \quad A, \tau \rightarrow \tau' = \sigma \rightarrow \sigma' \vdash \tau' = \sigma'}{A \vdash \tau \rightarrow \tau' = \sigma \rightarrow \sigma'}
\end{array}$$

Fig. 4. Coinductive axiomatization of recursive type equality

and operational interpretation of proofs. In Section 3 we briefly introduce the term language of *coercions* for proofs in our subtyping axiomatization. Each rule corresponds to a natural construction on coercions; in particular, the fixpoint rule corresponds to definition by recursion.

Finally, Section 4 concludes with a brief summary and possible future work.

2 Recursive Types: Subtyping

2.1 Simulations on Recursive Types

We give a characterization of \leq_{AC} that highlights the coinductive nature of \leq_{AC} . Its advantages are that it is intrinsically in terms of recursive types, without referring to infinite trees, and it directly reflects the characteristic closure properties of \leq_{AC} . It will be used in the proof of completeness for our axiomatization of \leq_{AC} .

Definition 4 (Simulation on recursive types). A *simulation (on recursive types)* is a binary relation \mathcal{R} on recursive types satisfying:

- (i) $(\tau_1 \rightarrow \tau_2) \mathcal{R} (\sigma_1 \rightarrow \sigma_2) \Rightarrow \sigma_1 \mathcal{R} \tau_1$ and $\tau_2 \mathcal{R} \sigma_2$
- (ii) $\mu\alpha. \tau \mathcal{R} \sigma \Rightarrow \tau[\mu\alpha. \tau/\alpha] \mathcal{R} \sigma$
- (iii) $\tau \mathcal{R} \mu\beta. \sigma \Rightarrow \tau \mathcal{R} \sigma[\mu\beta. \tau/\beta]$
- (iv) $\tau \mathcal{R} \sigma \Rightarrow \mathcal{L}(\tau) \leq \mathcal{L}(\sigma)$

Lemma 5. \leq_{AC} is a simulation.

Proof. We prove the four properties of Definition 4.

- (i) Assume $\tau_1 \rightarrow \tau_2 \leq_{\text{AC}} \sigma_1 \rightarrow \sigma_2$ and let $k \in \mathbb{N}_0$. By Definition 2 we have $\text{Tree}(\tau_1 \rightarrow \tau_2) \upharpoonright_{(k+1)} \leq_{\text{fin}} \text{Tree}(\sigma_1 \rightarrow \sigma_2) \upharpoonright_{(k+1)}$. By definition of $\text{Tree}(\cdot)$

and of the k 'th lower approximation we find

$$\left(\text{Tree}(\tau_1) |^k \rightarrow \text{Tree}(\tau_2) |^k \right) \leq_{\text{fin}} \left(\text{Tree}(\sigma_1) |^k \rightarrow \text{Tree}(\sigma_2) |^k \right)$$

and thus $\text{Tree}(\sigma_1) |^k \leq_{\text{fin}} \text{Tree}(\tau_1) |^k$ and $\text{Tree}(\tau_2) |^k \leq_{\text{fin}} \text{Tree}(\sigma_2) |^k$. Since k was chosen arbitrary and $\text{Tree}(\sigma_1) |^k \leq_{\text{fin}} \text{Tree}(\tau_1) |^k$ if and only if $\text{Tree}(\sigma_1) |^k \leq_{\text{fin}} \text{Tree}(\tau_1) |^k$ (Proposition 3) we finally obtain $\sigma_1 \leq_{\text{AC}} \tau_1$ and $\tau_2 \leq_{\text{AC}} \sigma_2$ as desired.

(ii) Consider $\mu\alpha.\tau \leq_{\text{AC}} \sigma$. By definition of $\text{Tree}(\cdot)$ we have $\text{Tree}(\mu\alpha.\tau) = \text{Tree}(\tau[\mu\alpha.\tau/\alpha])$ and thereby $\tau[\mu\alpha.\tau/\alpha] \leq_{\text{AC}} \sigma$.

(iii) Exactly as (ii).

(iv) Let $\tau \leq_{\text{AC}} \sigma$ and thus $\text{Tree}(\tau) |^k \leq_{\text{fin}} \text{Tree}(\sigma) |^k$ for all $k \in \mathbb{N}_0$. By inspection of \leq_{fin} we get $\mathcal{L}(\text{Tree}(\tau) |^k) \leq \mathcal{L}(\text{Tree}(\sigma) |^k)$. For $k > 0$ we obviously have $\mathcal{L}(\text{Tree}(\tau) |^k) = \mathcal{L}(\text{Tree}(\tau)) = \mathcal{L}(\tau)$ and hence $\mathcal{L}(\tau) \leq \mathcal{L}(\sigma)$.

Lemma 6. *If \mathcal{R} is a simulation then $\tau\mathcal{R}\sigma \Rightarrow \tau \leq_{\text{AC}} \sigma$ for all $\tau, \sigma \in \mu\text{Tp}$.*

Proof. We prove $\forall k \in \mathbb{N}_0. \forall \tau, \sigma \in \mu\text{Tp}. (\tau\mathcal{R}\sigma \Rightarrow \text{Tree}(\tau) |^k \leq_{\text{fin}} \text{Tree}(\sigma) |^k)$ by induction on k .

Case $k = 0$: Trivial, since $\perp \leq_{\text{fin}} \perp$.

Case $k > 0$: Let τ, σ be given such that $\tau\mathcal{R}\sigma$. We perform a case analysis on the syntactic forms of τ, σ where $\mathcal{L}(\tau) \leq \mathcal{L}(\sigma)$.

Case $\tau = \perp$ or $\sigma = \top$ or $\tau = \alpha, \sigma = \alpha$: Trivial.

Case $\tau = \mu\alpha.\tau', \sigma = \sigma_1 \rightarrow \sigma_2$: τ is canonical so $\tau'[\tau/\alpha] = \tau_1 \rightarrow \tau_2$ for some τ_1, τ_2 . By Definition 4 (ii) we have $(\tau_1 \rightarrow \tau_2)\mathcal{R}(\sigma_1 \rightarrow \sigma_2)$ and thus by (i) $\sigma_1\mathcal{R}\tau_1, \tau_2\mathcal{R}\sigma_2$. Our induction hypothesis yields

$$\text{Tree}(\sigma_1) |^{(k-1)} \leq_{\text{fin}} \text{Tree}(\tau_1) |^{(k-1)} \text{ and } \text{Tree}(\tau_2) |^{(k-1)} \leq_{\text{fin}} \text{Tree}(\sigma_2) |^{(k-1)}$$

By Proposition 3 and Rule $\text{ARROW}_{\text{fin}}$ we conclude

$$\text{Tree}(\tau_1) |^{(k-1)} \rightarrow \text{Tree}(\tau_2) |^{(k-1)} \leq_{\text{fin}} \text{Tree}(\sigma_1) |^{(k-1)} \rightarrow \text{Tree}(\sigma_2) |^{(k-1)}$$

which by definition of $\text{Tree}(\cdot)$ and $|^k$ implies

$$\text{Tree}(\tau) = \text{Tree}(\tau_1 \rightarrow \tau_2) |^k \leq_{\text{fin}} \text{Tree}(\sigma_1 \rightarrow \sigma_2) |^k = \text{Tree}(\sigma).$$

The remaining cases follow the same schema as the previous one, since they all have \rightarrow labels.

Theorem 7 (Characterization of Amadio/Cardelli subtyping). $\tau \leq_{\text{AC}} \sigma$ if and only if there exists a simulation \mathcal{R} such that $\tau\mathcal{R}\sigma$.

Proof. Follows by Lemma 5 and Lemma 6.

2.2 Soundness

We might want to interpret a sequent $\sigma_{11} \leq \sigma_{11}, \dots, \sigma_{n1} \leq \sigma_{n2} \vdash \tau \leq \tau'$ conventionally as “if $\sigma_{11} \leq_{AC} \sigma_{11}, \dots, \sigma_{n1} \leq_{AC} \sigma_{n2}$ then $\tau \leq_{AC} \tau'$ ” and prove every inference rule sound under this interpretation.

The problem is that Rule ARROW/FIX — more specifically the part that corresponds to Rule FIX — is *unsound* under this interpretation! To see this, consider for example $(\perp \rightarrow \top) \leq (\top \rightarrow \perp) \vdash \top \leq \perp$. Since $\perp \rightarrow \top \not\leq_{AC} \top \rightarrow \perp$ it is vacuously valid under the conventional interpretation. Application of Rule ARROW/FIX lets us deduce $\vdash \perp \rightarrow \top \leq \top \rightarrow \perp$, which is, however, *not* valid.

This does not mean that our inference system is unsound. The problem is that the interpretation of sequents is *too strong* (in the sense of “too many sequents are valid”): the premise $(\perp \rightarrow \top) \leq (\top \rightarrow \perp) \vdash \top \leq \perp$, which is obviously not derivable anyway, should not be valid. As suggested by Martín Abadi [Aba96] we give sequents a *level stratified* interpretation, under which all inference rules are sound.

Definition 8 (Stratified sequent interpretation). Let k range over the non-negative integers. Define:

1. $\models_k \tau \leq \tau'$ if $\text{Tree}(\tau)|_k \leq \text{Tree}(\tau')|_k$.
2. $\models_k A$ if $\models_k \tau \leq \tau'$ for all $\tau \leq \tau' \in A$.
3. $A \models_k \tau \leq \tau'$ if $\models_k A$ implies $\models_k \tau \leq \tau'$.
4. $A \models \tau \leq \tau'$ if $A \models_k \tau \leq \tau'$ for all $k \in \mathbb{N}_0$.

Note that $\perp \rightarrow \top \leq \top \rightarrow \perp \vdash \top \leq \perp$ does *not* hold under this interpretation; that is, $(\perp \rightarrow \top) \leq (\top \rightarrow \perp) \not\models \top \leq \perp$. To wit, we have $(\perp \rightarrow \top) \leq (\top \rightarrow \perp) \models_0 \top \leq \perp$ and $(\perp \rightarrow \top) \leq (\top \rightarrow \perp) \models_k \top \leq \perp$ for all $k \geq 2$, but $(\perp \rightarrow \top) \leq (\top \rightarrow \perp) \not\models_1 \top \leq \perp$ since $\text{Tree}(\perp \rightarrow \top)|_1 = \top \rightarrow \perp = \text{Tree}(\top \rightarrow \perp)|_1$, yet $\text{Tree}(\top)|_1 = \top \not\leq \perp = \text{Tree}(\perp)|_1$. Intuitively, a sequent $A \vdash \tau \leq \tau'$ that holds vacuously (because the assumptions are false) under the conventional interpretation holds under the stratified interpretation only if $\tau \leq \tau'$ is not wrong “earlier” than an assumption in A when descending into the trees in A and $\tau \leq \tau'$ in lockstep.

Lemma 9 (Soundness of inference rules). *If $A \vdash \tau \leq \tau'$ then $A \models \tau \leq \tau'$.*

Proof. The proof is by rule induction on the inference rules in Figure 3. For all rules but ARROW/FIX it is easy to prove $A \models_k \tau \leq \tau'$ for arbitrary k and then generalize over k . For Rule ARROW/FIX we require induction on k .

Recall Rule ARROW/FIX:

$$\frac{A, \tau_1 \rightarrow \tau_2 \leq \sigma_1 \rightarrow \sigma_2 \vdash \sigma_1 \leq \tau_1 \quad A, \tau_1 \rightarrow \tau_2 \leq \sigma_1 \rightarrow \sigma_2 \vdash \tau_2 \leq \sigma_2}{A \vdash \tau_1 \rightarrow \tau_2 \leq \sigma_1 \rightarrow \sigma_2}$$

Our major induction hypothesis IH1 is $A, \tau_1 \rightarrow \tau_2 \leq \sigma_1 \rightarrow \sigma_2 \models \sigma_1 \leq \tau_1$ and $A, \tau_1 \rightarrow \tau_2 \leq \sigma_1 \rightarrow \sigma_2 \models \tau_2 \leq \sigma_2$. We now prove $\forall k \in \mathbb{N}_0. A \models_k \tau_1 \rightarrow \tau_2 \leq \sigma_1 \rightarrow \sigma_2$ by induction on k .

Base case: $k = 0$. Trivial since $\text{Tree}(\tau)|_0 = \perp$ for all τ .

Inductive case: $k > 0$. Assume $A \models_{(k-1)} \tau_1 \rightarrow \tau_2 \leq \sigma_1 \rightarrow \sigma_2$ (minor induction hypothesis IH2). We need to show $A \models_k \tau_1 \rightarrow \tau_2 \leq \sigma_1 \rightarrow \sigma_2$; that is, $\models_k A$ implies $\models_k \tau_1 \rightarrow \tau_2 \leq \sigma_1 \rightarrow \sigma_2$.

Assume $\models_k A$. This implies that $\models_{(k-1)} A$. Since $A \models_{(k-1)} \tau_1 \rightarrow \tau_2 \leq \sigma_1 \rightarrow \sigma_2$ by IH2 we obtain $\models_{(k-1)} \tau_1 \rightarrow \tau_2 \leq \sigma_1 \rightarrow \sigma_2$ and thus $\models_{(k-1)} A, \tau_1 \rightarrow \tau_2 \leq \sigma_1 \rightarrow \sigma_2$. Invoking IH1 we derive $\models_{(k-1)} \sigma_1 \leq \tau_1$ and $\models_{(k-1)} \tau_2 \leq \sigma_2$, which together are equivalent to $\models_k \tau_1 \rightarrow \tau_2 \leq \sigma_1 \rightarrow \sigma_2$, and we are done.

Theorem 10 (Soundness). *If $\vdash \tau \leq \tau'$ then $\tau \leq_{\text{AC}} \tau'$.*

Proof. Follows immediately from Lemma 9 and the observation that $\models \tau \leq \tau'$ if and only if $\tau \leq_{\text{AC}} \tau'$.

2.3 Completeness

This section is concerned with the completeness of the inference system in Figure 3 with respect to \leq_{AC} . The proof is divided into three parts; 1) an algorithm **S** that produces derivations, 2) a termination proof and finally 3) a correctness proof for **S**.

Algorithm S Consider Algorithm **S** in Figure 5. The first clause in **S** that matches a particular argument tuple is executed. The only cases requiring remarks are those concerning function types. A pair of function types may have been encountered earlier in the computation and is therefore stored in the assumption set. If that is the case, rule HYP is applied and otherwise rule ARROW/FIX. It is of vital importance that assumptions are checked before applying the ARROW/FIX rule, since otherwise we would never be able to use them.

Termination of S

Syntactic subterms We first introduce the concept of syntactic subterms and prove a crucial property about them: every recursive type has only a finite number of syntactic subterms. This is not entirely obvious since recursive types may have syntactic subterms that are larger than themselves. We require a number of preliminary technical results.

Definition 11. A recursive type τ' is a *syntactic subterm* (or just *subterm*) of τ if $\tau' \sqsubseteq \tau$, where \sqsubseteq is defined by the following rules:

$$\begin{array}{c} \tau \sqsubseteq \tau \quad (\text{REF}) \\ \frac{\tau \sqsubseteq \sigma_1}{\tau \sqsubseteq \sigma_1 \rightarrow \sigma_2} \quad (\text{ARROW}_L) \end{array} \qquad \begin{array}{c} \frac{\tau \sqsubseteq \sigma[\mu\alpha.\sigma/\alpha]}{\tau \sqsubseteq \mu\alpha.\sigma} \quad (\text{UNFOLD}) \\ \frac{\tau \sqsubseteq \sigma_2}{\tau \sqsubseteq \sigma_1 \rightarrow \sigma_2} \quad (\text{ARROW}_R) \end{array}$$

<pre> 1: S(A, μα.τ, σ) = 2: let 3: D₁ = UNFOLD 4: D₂ = S(A, τ[μα.τ/α], σ) 5: in 6: TRANS(D₁, D₂) 7: end 8: S(A, τ, μβ.σ) = 9: let 10: D₁ = S(A, τ, σ[μβ.σ/β]) 11: D₂ = FOLD 12: in 13: TRANS(D₁, D₂) 14: end </pre>	<pre> 15: S((A, τ ≤ σ, A'), τ, σ) = HYP 16: S(A, τ₁ → τ₂, σ₁ → σ₂) = 17: let 18: A' = A ∪ {τ₁ → τ₂ ≤ σ₁ → σ₂} 19: D₁ = S(A', σ₁, τ₁) 20: D₂ = S(A', τ₂, σ₂) 21: in 22: ARROW/FIX(D₁, D₂) 23: end 24: S(A, α, α) = REF 25: S(A, ⊥, τ) = ⊥ 26: S(A, τ, ⊤) = ⊤ 27: S(A, τ, σ) = exception </pre>
--	---

Fig. 5. Algorithm S

Lemma 12. *The subterm relation is transitive, i.e. if $\tau \sqsubseteq \delta$, $\delta \sqsubseteq \sigma$ then $\tau \sqsubseteq \sigma$.*

Proof. Induction on the derivation of $\delta \sqsubseteq \sigma$.

We define a subterm closure operation on recursive types. As we shall see the subterm closure contains all subterms of a recursive type.

Definition 13. *The subterm closure τ^* of τ is the set of recursive types defined by*

$$\begin{aligned}
\perp^* &= \{\perp\} & (\tau_1 \rightarrow \tau_2)^* &= \{\tau_1 \rightarrow \tau_2\} \cup \tau_1^* \cup \tau_2^* \\
\top^* &= \{\top\} & (\mu\alpha.\tau_1)^* &= \{\mu\alpha.\tau_1\} \cup \tau_1^*[\mu\alpha.\tau_1/\alpha] \\
\alpha^* &= \{\alpha\}
\end{aligned}$$

Obviously, the subterm closure is finite; indeed $|\tau^*| = O(|\tau|)$.

Proposition 14. $|\tau^*| < \infty$.

An important technical property of the closure operation is its commutation with substitution:

Lemma 15. $(\tau'[\tau/\beta])^* = (\tau')^*[\tau/\beta] \cup \tau^*$ if $\beta \in \text{fv}(\tau')$.

Proof. Induction on the structure of τ' .

Using this property we can show that τ^* contains all syntactic subterms of τ :

Lemma 16. *If $\tau \sqsubseteq \sigma$ then $\tau \in \sigma^*$.*

Proof. Induction on the derivation of $\tau \sqsubseteq \sigma$. The only interesting case is UNFOLD.

Case UNFOLD: So $\sigma = \mu\alpha.\tau'$ and $\tau \sqsubseteq \tau'[\mu\alpha.\tau'/\alpha]$. By IH we get $\tau \in (\tau'[\mu\alpha.\tau'/\alpha])^*$. By Lemma 15 substitution and closure commute and we can conclude

$$\tau \in (\tau')^*[\mu\alpha.\tau'/\alpha] \cup (\mu\alpha.\tau')^* = (\mu\alpha.\tau')^*$$

since $(\tau')^*[\mu\alpha.\tau'/\alpha] \subseteq (\mu\alpha.\tau')^*$ by definition of $(\mu\alpha.\tau')^*$.

Lemma 16 and Proposition 14 together finally give us the desired property:

Theorem 17. *For recursive type τ the set $\{\tau' \mid \tau' \sqsubseteq \tau\}$ is finite.*

Algorithm execution We now study the computations performed by **S**. The main result is that all recursive types encountered in calls to **S** during the computation are syntactic subterms of the initial recursive types. Combined with Theorem 17 we can prove that **S** terminates. To reason about the steps performed by **S** we define the notions of call tree and call path.

Definition 18. The *call tree* of $\mathbf{S}(A_0, \tau_0, \sigma_0)$ is defined to be a root node labeled $\mathbf{S}(A_0, \tau_0, \sigma_0)$ whose subtrees are the call trees of all the recursive calls $\mathbf{S}(A_i, \tau_i, \sigma_i)$ (finitely many) occurring in the first clause in **S** that matches $\mathbf{S}(A_0, \tau_0, \sigma_0)$.

A *call path* in $\mathbf{S}(A_0, \tau_0, \sigma_0)$ is a path in the call tree of $\mathbf{S}(A_0, \tau_0, \sigma_0)$, starting at its root.

Theorem 19. *Let τ_0, σ_0 be recursive types and A_0 an assumption set. For all nodes $\mathbf{S}(A_i, \tau_i, \sigma_i)$ in the call tree of $\mathbf{S}(A_0, \tau_0, \sigma_0)$ we have that τ_i, σ_i are syntactic subterms of either τ_0 or σ_0 .*

Proof. Induction on the depth d of nodes.

Case $d = 0$: Root node $\mathbf{S}(A_0, \tau_0, \sigma_0)$. Trivial from reflexivity of \sqsubseteq .

Case $d > 0$: Case analysis of nodes at depth $d - 1$.

Case $\mathbf{S}(A, \mu\alpha.\tau, \sigma)$: The unique child of $\mathbf{S}(A, \mu\alpha.\tau, \sigma)$ at depth d is then $\mathbf{S}(A, \tau[\mu\alpha.\tau/\alpha], \sigma)$. By induction hypothesis we know that $\mu\alpha.\tau$ and σ are in canonical form and furthermore that

$$(\mu\alpha.\tau \sqsubseteq \tau_0 \wedge \sigma \sqsubseteq \sigma_0) \quad \text{or} \quad (\mu\alpha.\tau \sqsubseteq \sigma_0 \wedge \sigma_i \sqsubseteq \tau_0)$$

It is easily seen that $\tau[\mu\alpha.\tau/\alpha]$ is in canonical form. Assume that $\mu\alpha.\tau \sqsubseteq \tau_0$ (second case similar). By Rule UNFOLD we have $\tau[\mu\alpha.\tau/\alpha] \sqsubseteq \mu\alpha.\tau$ and thus by transitivity (Lemma 12) $\tau[\mu\alpha.\tau/\alpha] \sqsubseteq \tau_0$.

Case $\mathbf{S}(A, \tau, \mu\beta.\sigma)$: Analogous to the above case.

Case $\mathbf{S}(A, \tau_1 \rightarrow \tau_2, \sigma_1 \rightarrow \sigma_2)$: There are two child nodes at depth d :

1. $\mathbf{S}(A', \sigma_1, \tau_1)$. By IH we know that $\tau_1 \rightarrow \tau_2 \sqsubseteq \tau_0$ and $\sigma_1 \rightarrow \sigma_2 \sqsubseteq \sigma_0$, or $\tau_1 \rightarrow \tau_2 \sqsubseteq \sigma_0$ and $\sigma_1 \rightarrow \sigma_2 \sqsubseteq \tau_0$. But then the result follows directly from transitivity (Lemma 12) since $\tau_1 \sqsubseteq \tau_1 \rightarrow \tau_2$ and $\sigma_1 \sqsubseteq \sigma_1 \rightarrow \sigma_2$ by Rule ARROW_L.

2. $\mathbf{S}(A', \tau_2, \sigma_2)$ Exactly as previous case.

Lemma 20. *If $\mathbf{S}(A_0, \tau_0, \sigma_0), \dots, \mathbf{S}(A_i, \tau_i, \sigma_i), \dots$ is a call path of $\mathbf{S}(A_0, \tau_0, \sigma_0)$ then $A_0 \subseteq A_1 \subseteq \dots \subseteq A_i \subseteq \dots$*

Proof. By inspection of Algorithm \mathbf{S} we see that, for every clause $\mathbf{S}(A, \tau, \sigma)$ and every recursive call $\mathbf{S}(A', \tau', \sigma')$ occurring in it, we have $A \subseteq A'$.

Lemma 21. *If $\mathbf{S}(A_0, \tau_0, \sigma_0), \dots, \mathbf{S}(A_n, \tau_n, \sigma_n), \dots$ is a call path of $\mathbf{S}(A_0, \tau_0, \sigma_0)$ then: $\exists N \forall i : (\tau_i, \sigma_i) \in \left\{ (\tau_j, \sigma_j) \mid 0 \leq j \leq N \right\}$.*

The lemma states that every path has only finitely many different type arguments.

Proof. The statement is proved by contradiction. Assume that

$$\forall N \exists i : (\tau_i, \sigma_i) \notin \left\{ (\tau_j, \sigma_j) \mid 0 \leq j \leq N \right\}$$

This fact directly implies that $\{(\tau_j, \sigma_j) \mid j \in \mathbb{N}_0\}$ is an infinite set. Theorem 19 states that all terms in a call tree are subterms of the initial two terms. We thus have

$$\{(\tau_j, \sigma_j) \mid j \in \mathbb{N}_0\} \subseteq (\{\tau_j \mid j \in \mathbb{N}_0\}) \times (\{\sigma_j \mid j \in \mathbb{N}_0\}) \subseteq \{\tau \mid \tau \sqsubseteq \tau_0\} \times \{\sigma \mid \sigma \sqsubseteq \sigma_0\}$$

Theorem 17, however, implies that

$$\left| \{(\tau_j, \sigma_j) \mid j \in \mathbb{N}_0\} \right| \leq \left| \{\tau \mid \tau \sqsubseteq \tau_0\} \cup \{\sigma \mid \sigma \sqsubseteq \sigma_0\} \right|^2 < \infty$$

which contradicts our assumption that $\{(\tau_j, \sigma_j) \mid j \in \mathbb{N}_0\}$ is infinite.

The above results enable us to prove termination of \mathbf{S} .

Theorem 22 (Termination of \mathbf{S}). *If τ, σ are canonical and A an assumption set then $\mathbf{S}(A, \tau, \sigma)$ terminates.*

Proof. The proof is once again by contradiction. Assume that $\mathbf{S}(A, \tau, \sigma)$ does not terminate, i.e. there exists an infinite call path p in the call tree of $\mathbf{S}(A, \tau, \sigma)$. Let N be determined by Lemma 21 such that

$$\forall i : (\tau_i, \sigma_i) \in \left\{ (\tau_j, \sigma_j) \mid 0 \leq j \leq N \right\} \tag{1}$$

Let us consider the calls (τ_i, σ_i) of p where $i > N$. There must exist a call (τ_n, σ_n) with $n > N$ where $\tau_n = \tau_1 \rightarrow \tau_2$ and $\sigma_n = \sigma_1 \rightarrow \sigma_2$, because otherwise all calls would be unfoldings, which is not possible since the terms are in canonical form (Theorem 19). From (1) we conclude that $(\tau_1 \rightarrow \tau_2, \sigma_1 \rightarrow \sigma_2) \in \{(\tau_j, \sigma_j) \mid 0 \leq j \leq N\}$ which implies that there exists $m \leq N < n$ such that $(\tau_n, \sigma_n) = (\tau_m, \sigma_m)$. The assumption set associated with call n inherits all assumptions from its ancestors in p (by Lemma 20), but then call n must be an application of HYP, which corresponds to a leaf in the call tree. Path p is therefore not infinite and the assumption is false.

Correctness of S Finally we show that, whenever $\tau \leq_{AC} \sigma$, $\mathbf{S}(A, \tau, \sigma)$ does not fail (it does not raise an exception) and it returns a proof of $A \vdash \tau \leq \sigma$.

Lemma 23. *Let τ, σ be recursive types in canonical form and A an assumption set. If $\tau \leq_{AC} \sigma$ then $\mathbf{S}(A, \tau, \sigma)$ returns a derivation of $A \vdash \tau \leq \sigma$.*

Proof. The termination theorem (Theorem 22) gives that $\mathbf{S}(A, \tau, \sigma)$ terminates with, say, n recursive calls. Correctness is proved by induction on n . In each case we verify the derivation returned by \mathbf{S} .

Case $n = 0$: No recursive calls performed at all. Case analysis on the clauses in \mathbf{S} with no recursive calls.

Case $\mathbf{S}((A, \tau \leq \sigma, A'), \tau, \sigma)$, $\mathbf{S}(A, \alpha, \alpha)$, $\mathbf{S}(A, \perp, \tau)$ or $\mathbf{S}(A, \tau, \top)$: Obvious.

Case $\mathbf{S}(A, \tau, \sigma)$: If this clause is reached, it means that $\mathcal{L}(\tau) \neq \perp$, $\mathcal{L}(\sigma) \neq \top$ and $\mathcal{L}(\tau) \neq \mathcal{L}(\sigma)$, i.e. $\mathcal{L}(\tau) \not\leq \mathcal{L}(\sigma)$. Since \leq_{AC} is a simulation and $\tau \leq_{AC} \sigma$ it must hold that $\mathcal{L}(\tau) \leq \mathcal{L}(\sigma)$, which contradicts $\mathcal{L}(\tau) \not\leq \mathcal{L}(\sigma)$. Thus this clause is never reached!

Case $n > 0$: Induction hypothesis: Computations $\mathbf{S}(A', \tau', \sigma')$ with fewer than n recursive calls, where $\tau' \leq_{AC} \sigma'$, produces a correct derivation of $A' \vdash \tau' \leq \sigma'$. Case analysis of rules containing recursive calls.

Case $\mathbf{S}(A, \mu\alpha.\tau, \sigma)$: Since \leq_{AC} is a simulation (Lemma 5) it follows that $\tau[\mu\alpha.\tau/\alpha] \leq_{AC} \sigma$. The induction hypothesis does thus apply and gives that $\mathbf{S}(A, \tau[\mu\alpha.\tau/\alpha], \sigma)$ returns a derivation of $A \vdash \tau[\mu\alpha.\tau/\alpha] \leq \sigma$. By UNFOLD and TRANS we then get a proof of $A \vdash \mu\alpha.\tau \leq \sigma$, which is exactly what $\mathbf{S}(A, \mu\alpha.\tau, \sigma)$ returns.

Case $\mathbf{S}(A, \tau, \mu\beta.\sigma)$: Since \leq_{AC} is a simulation (Lemma 5) we get $\tau \leq_{AC} \sigma[\mu\beta.\sigma/\beta]$. Thus the induction hypothesis is applicable: $\mathbf{S}(A, \tau, \sigma[\mu\beta.\sigma/\beta])$ returns a proof of $A \vdash \tau \leq \sigma[\mu\beta.\sigma/\beta]$. We conclude

$$\frac{\frac{\text{(IH)}}{A \vdash \tau \leq \sigma[\mu\beta.\sigma/\beta]} \quad \frac{\text{(FOLD)}}{A \vdash \sigma[\mu\beta.\sigma/\beta] \leq \mu\beta.\sigma}}{A \vdash \tau \leq \mu\beta.\sigma} \text{(TRANS)}$$

which is the result of $\mathbf{S}(A, \tau, \mu\beta.\sigma)$.

Case $\mathbf{S}(A, \tau_1 \rightarrow \tau_2, \sigma_1 \rightarrow \sigma_2)$: Let $A' = A \cup \{\tau_1 \rightarrow \tau_2 \leq \sigma_1 \rightarrow \sigma_2\}$. Two recursive calls are issued from this rule.

1. $\mathbf{S}(A', \sigma_1, \tau_1)$. From the simulation property of \leq_{AC} and IH we get that the call returns a proof of $A' \vdash \sigma_1 \leq \tau_1$.
2. $\mathbf{S}(A', \tau_2, \sigma_2)$. As above, the call returns a proof of $A' \vdash \tau_2 \leq \sigma_2$.

$\mathbf{S}(A, \tau_1 \rightarrow \tau_2, \sigma_1 \rightarrow \sigma_2)$ returns Rule ARROW/FIX applied to the two subproofs above, which is a proof of $A \vdash \tau_1 \rightarrow \tau_2 \leq \sigma_1 \rightarrow \sigma_2$.

Theorem 24 (Completeness). *If $\tau \leq_{AC} \sigma$ then $\vdash \tau \leq \sigma$*

Proof. Follows from Lemma 23 for $A = \emptyset$.

3 Proofs as Coercions

In this section we present a somewhat generalized axiomatization of recursive subtyping in which rules **ARROW** and **FIX** are separated instead of being melded into a single rule as in Figure 3. This is made possible by using an explicit term representation of proofs as *coercions*. Sound application of Rule **FIX** is then guaranteed by requiring the coercion in the premise to be formally *contractive* in a sense to be defined. The coercion constructions for the rules in the axiomatization can be interpreted as natural functional programming constructs. Notably, the fixpoint rule corresponds to definition by recursion. Coercions can then be used as a basis for *proof theory* as well as *operational interpretation* of proofs in the sense of the Curry-Howard isomorphism.

The latter is important where coercions are not only (constructive) evidence of some subsumption relation, but have semantic significance; that is, they denote functions that map an element of one type to an element of its supertype. Furthermore, coercions may have *operational significance*; that is, different proofs of the *same* subtyping statement may yield coercions with different operational characteristics. For example, folding and unfolding to and from a recursive type (corresponding to the axioms $A \vdash \tau[\mu\alpha.\tau/\alpha] \leq \mu\alpha.\tau$ and $A \vdash \mu\alpha.\tau \leq \tau[\mu\alpha.\tau/\alpha]$, respectively) may operationally require execution of a referencing (heap allocation) and (pointer) dereferencing step, respectively. In this case it is important to replace their composition by the identity coercion (corresponding to the axiom of reflexivity), since the latter is obviously operationally more efficient than the former.

In a separate paper we shall explore the semantics and operational interpretation of coercions. Here we shall only briefly describe their operational interpretation in order to demonstrate, in intuitive and nontechnical terms, how each rule corresponds to a natural program construct.

3.1 Coercions and their functional interpretation

Coercions are defined by the grammar

$$C := \iota_\tau \mid f \mid \mathbf{fix} \ f : \tau \leq \tau'.c \mid c; c \mid c \rightarrow c \mid \text{fold}_{\mu\alpha.\tau} \mid \text{unfold}_{\mu\alpha.\tau} \mid \text{abort}_\tau \mid \text{discard}_\tau$$

Each coercion can be interpreted as a function:

- ι_τ denotes the identity on type τ .
- $\mathbf{fix} \ f : \tau \leq \tau'.c$ denotes the function f recursively defined by the equation $f = c$ (note that f may occur in c).
- $c; c'$ denotes the composition of c' with c .
- $c \rightarrow c'$ denotes the functional F defined by $Ffx = c'(f(cx))$.
- The pair $\text{fold}_{\mu\alpha.\tau}$ and $\text{unfold}_{\mu\alpha.\tau}$ denotes the isomorphism between $\tau[\mu\alpha.\tau/\alpha]$ and $\mu\alpha.\tau$.
- abort_τ maps any argument to \perp ; that is, operationally it enters an infinite loop.
- discard_τ discards its argument and returns $()$, the only defined element of type \top .

3.2 Well-typed coercions

Definition 25 (Contractiveness). A coercion c is (*formally*) *contractive* in coercion variable f if: f does not occur in c ; or $c \equiv c_1 \rightarrow c_2$; or $c \equiv c_1; c_2$ and both c_1 and c_2 are contractive in f ; or $c \equiv \mathbf{fix} g : \tau \leq \sigma$, c_1 and c_1 is contractive in f .

Definition 26 (Well-typed coercions, canonical coercions). A coercion c is *well-typed* if $E \vdash c : \tau \leq \sigma$ is derivable for some E, τ, σ in the inference system of Figure 6.

A well-typed coercion c is *canonical* if every \mathbf{fix} -coercion occurring in c has the form $\mathbf{fix} f : \tau' \leq \sigma'. c_1 \rightarrow c_2$.

$$\begin{array}{c}
 E \vdash \iota_\tau : \tau \leq \tau \\
 \\
 E \vdash \mathbf{abort}_\tau : \perp \leq \tau \qquad E \vdash \mathbf{discard}_\tau : \tau \leq \top \\
 \\
 E \vdash \mathbf{unfold}_{\mu\alpha.\tau} : \mu\alpha.\tau \leq \tau[\mu\alpha.\tau/\alpha] \quad E \vdash \mathbf{fold}_{\mu\alpha.\tau} : \tau[\mu\alpha.\tau/\alpha] \leq \mu\alpha.\tau \\
 \\
 \frac{E \vdash c : \tau \leq \delta \quad E \vdash d : \delta \leq \sigma}{E \vdash c; d : \tau \leq \sigma} \qquad \frac{E \vdash c : \tau \leq \tau' \quad d : \sigma \leq \sigma'}{E \vdash (c \rightarrow d) : (\tau' \rightarrow \sigma) \leq (\tau \rightarrow \sigma')} \\
 \\
 E, f : \tau \leq \sigma, E' \vdash f : \tau \leq \sigma \qquad \frac{E, f : \tau \leq \sigma \vdash c_f : \tau \leq \sigma \quad c_f \text{ contr. in } f}{E \vdash \mathbf{fix} f : \tau \leq \sigma. c_f : \tau \leq \sigma}
 \end{array}$$

Fig. 6. Coercion typing rules

Thinking of coercions as a special class of functions, \leq can be understood as a special (coercion) type constructor in Figure 6. Assumptions in E are of the form $f : \tau \leq \sigma$ where coercion variable f occurs at most once in E . Note that the subscripting of coercions guarantees that there is exactly one proof for every derivable $E \vdash c : \tau \leq \sigma$. Let us write \bar{E} for the subtyping assumptions we get from E by erasing all coercion variables in it.

Well-typed coercions are a term interpretation of the axiomatization in Figure 3 in the sense that for every E and every proof of $\bar{E} \vdash \tau \leq \tau'$ there exists a unique canonical coercion c such that $E \vdash c : \tau \leq \tau'$, where every coercion of the form $c_1 \rightarrow c_2$ is the body of some \mathbf{fix} -coercion. Conversely, it is easily seen that every canonical coercion of this form corresponds to a proof using the inference rules of Figure 3.

Theorem 27. $\vdash \tau \leq \tau'$ if and only if there exists a canonical coercion c such that $\vdash c : \tau \leq \tau'$.

Proof. “Only if” is obvious. Let $\vdash c : \tau \leq \tau'$. “If” follows from the observation that every coercion occurrence of the form $c_1 \rightarrow c_2$ can be “wrapped” with $\mathbf{fix} f : \sigma \leq \sigma'$ (f fresh) for suitable recursive types σ, σ' if it is not already the body of a \mathbf{fix} -coercion. Once this is done, the transformed coercion corresponds directly to a derivation in Figure 3.

This theorem holds not only for *canonical* coercions, but also for the larger class of *well-typed* coercions; that is, well-typed coercions give more *proofs*, but not more *theorems* than canonical coercions. Since this requires a rather lengthy and involved proof, however, we omit it here.

4 Conclusion

4.1 Summary

We have given sound and complete axiomatizations of type equality and type containment using a novel fixpoint rule, which represents a coinduction principle. We have argued that this gives rise to a natural interpretation of proofs as coercions where the fixpoint rule corresponds to definition by recursion.

4.2 Future work

Proof theory, semantics and operational interpretation of coercions In continuation of the work reported here we have formulated an equational theory of coercions that is complete in the sense that two coercions are provably equal if and only if they have identical type signatures. Interestingly, this theory is coinductive, too, as it is based on the fixpoint rule, though for coercion equalities instead of type equalities or subtypings. We can show that the equational theory is verified in a number of functional (cpo-based) interpretations of coercions. This shows that, extensionally, any two coercions with the same type signature are equivalent. Conversely, the equational theory codifies the requirements on a semantics of coercions if we demand that any two coercions be extensionally equivalent. The equational theory can be used as a starting point for optimization of coercions by rewriting: even though coercions of equal type signature are extensionally equivalent, they are not necessarily equally *efficient*!

On the theoretical side, we would like to extend the equational theory for coercions to the typed lambda calculus with embedded coercions in the style of [BTCS91, CG90, Hen94, Reh95] in order to obtain a general coherence characterization for simply typed lambda-calculus with recursive subtyping. This should give another approach to comparing the semantics of FPC under type equality on the one hand and under type isomorphism on the other hand [AF96]. Furthermore, the interrelation of our fixpoint rule and the co-induction principle of Pitts [Pit94] needs to be illuminated; see also [Pit96].

On the more practical side, coercion reduction by rewriting appears to be useful in representation optimization (such as *boxing* analysis [Jør95]) for recursively defined types; this may, however, require admitting more powerful transformations to be useful, for example the isomorphism $S \times (T + U) \approx (S \times T) + (S \times U)$.

Coinduction principles in other formal systems Observational congruence [Mor68] of programs is intuitively a coinductive notion since it states that, if it is impossible to provide finitary evidence that two expressions behave differently, then they are observationally congruent. Indeed for many programming languages observational congruence can be characterized by a notion of bisimulation. (Since the literature on this topic is voluminous we make no attempt at completeness; see e.g. [Mil77, Abr90].) It is thus not surprising that coinduction principles play an important role in proving program properties and in particular equivalences [MT91, Gor95, HL95, Len96]. Relatively little, however, seems to have been done on incorporating coinduction principles in formal proof systems. Coquand [Coq93] formulates a *guarded induction principle* for reasoning about infinite objects within Type Theory, to which our fixpoint rule and its contractiveness requirement is a close pendant.

We are interested in applying our coinduction principle (fixpoint rule) to other coinductive notions such as program equivalence. For example, we hope to formulate a λ -theory that is strong enough to capture regular Böhm tree equality. More abstractly, it seems to be possible to characterize when coinductively defined relations can be completely axiomatized using the fixpoint rule.

Extensions to richer type languages and systems We have studied recursive types and subtyping within a type language of simple types. It would be interesting to extend this study to richer type disciplines with polymorphism (predicative and impredicative), intersection types or object typing.

Acknowledgments

We would like to thank Martín Abadi, Luca Cardelli, Andrew Gordon, Furio Honsell, Jakob Rehof, Simona Ronchi della Rocca, Dave Sands, Mads Tofte and the anonymous referees for helpful discussions, feedback, corrections, and pointers to related or interesting literature. Martín Abadi's help has been particularly valuable in several respects: he came up with the notion of a level stratified interpretation of judgements which is the basis of the soundness proof presented in this paper, replacing our original soundness proof; he found a number of mistakes in our submission; and he has provided interesting pointers to relevant topics and literature. Dave Sands pointed out the coinductive nature of recursive type equality in the early stages of this work and provided valuable help during a number of discussions over a period of two years.

References

- [Aba96] Martín Abadi. Personal communication, September 1996. ACM State of the Art Summer School on Functional and Object-Oriented Programming in Sobotka, Poland, September 8-14, 1996.
- [Abr90] Samson Abramsky. The lazy lambda calculus. In D. Turner, editor, *Research Topics in Functional Programming*, pages 65–117. Addison-Wesley, 1990. Also available by anonymous ftp from theory.doc.ic.ac.uk.
- [AC91] R. Amadio and L. Cardelli. Subtyping recursive types. In *Proc. 18th Annual ACM Symposium on Principles of Programming Languages (POPL)*, Orlando, Florida, pages 104–118. ACM Press, January 1991.
- [AC93] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(4):575–631, September 1993.
- [AF96] Martín Abadi and Marcelo P. Fiore. Syntactic considerations on recursive types. In *Proc. 1996 IEEE 11th Annual Symp. on Logic in Computer Science (LICS)*, New Brunswick, New Jersey. IEEE Computer Society Press, June 1996.
- [AK95] Zena M. Ariola and Jan Willem Klop. Equational term graph rewriting. Technical report, University of Oregon, 1995. To appear in *Acta Informatica*.
- [BTCGS91] V. Breazu-Tannen, T. Coquand, C. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93(1):172–221, July 1991. Presented at LICS '89.
- [BY79] Ch. Ben-Yelles. Type assignment in the lambda-calculus: Syntax and semantics. Technical report, Department of Pure Mathematics, University College of Swansea, September 1979. Author's current address: Université des Sciences et de la Technologie Houari Boumediene, Institut D'Informatique, El-Alia B.P. No. 32, Alger, Algeria.
- [Car93] Luca Cardelli. Algorithm for subtyping recursive types (in Modula-3). http://www.research.digital.com/SRC/personal/Luca_Cardelli/Notes/RecSub.txt, 1993. Originally implemented in Quest and released in 1990.
- [CC91] F. Cardone and M. Coppo. Type inference with recursive types: Syntax and semantics. *Information and Computation*, 92(1):48–80, May 1991.
- [CG90] P. Curien and G. Ghelli. Coherence of subsumption. In A. Arnold, editor, *Proc. 15th Coll. on Trees in Algebra and Programming, Copenhagen, Denmark*, pages 132–146. Springer, May 1990.
- [Coq93] Thierry Coquand. Infinite objects in type theory. In *Proc. Int'l Workshop on Types for Proofs and Programs*, Lecture Notes in Computer Science (LNCS), pages 62–78. Springer-Verlag, 1993.
- [Cou83] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.
- [Gor95] Andrew Gordon. Bisimilarity as a theory of functional programming. In *Proceedings of the Eleventh Conference on the Mathematical Foundations of Programming Semantics (MFPS)*, New Orleans, March 29 to April 1, 1995, *Elsevier Electronic Notes in Theoretical Computer Science*, volume 1, 1995.
- [Hen94] Fritz Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming (SCP)*, 22(3):197–230, 1994.

- [Hin69] R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc.*, 146:29–60, December 1969.
- [HL95] Furio Honsell and Marina Lenisa. Final semantics for untyped lambda-calculus. In *Proc. Int'l Conf. on Typed Lambda Calculi and Applications (TLCA)*, volume 902 of *Lecture Notes in Computer Science (LNCS)*, pages 249–265. Springer-Verlag, 1995.
- [Jør95] Jesper Jørgensen. *A Calculus for Boxing Analysis of Polymorphically Typed Languages*. PhD thesis, DIKU, University of Copenhagen, October 1995.
- [KPS93] Dexter Kozen, Jens Palsberg, and Michael Schwartzbach. Efficient recursive subtyping. In *Proc. 20th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 419–428. ACM, ACM Press, January 1993.
- [KPS95] Dexter Kozen, Jens Palsberg, and Michael Schwartzbach. Efficient recursive subtyping. *Mathematical Structures in Computer Science (MSCS)*, 5(1), 1995.
- [Len96] Marina Lenisa. Final semantics for a higher order concurrent language. In H. Kirchner, editor, *Proc. Coll. on Trees in Algebra and Programming (CAAP)*, volume 1059 of *Lecture Notes in Computer Science (LNCS)*, pages 102–118. Springer-Verlag, 1996.
- [Mil77] Robin Milner. Fully abstract models of typed lambda-calculi. *Theoretical Computer Science (TCS)*, 4(1):1–22, February 1977.
- [Mil84] Robin Milner. A complete inference system for a class of regular behaviours. *Journal of Computer and System Sciences (JCSS)*, 28:439–466, 1984.
- [Mor68] J. Morris. *Lambda-Calculus Models of Programming Languages*. PhD thesis, MIT, 1968.
- [MT91] Robin Milner and Mads Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87(1):209–220, 1991. Note.
- [Pit94] Andrew M. Pitts. A co-induction principle for recursively defined domains. *Theoretical Computer Science (TCS)*, 124:195–219, 1994.
- [Pit96] Andrew M. Pitts. Relational properties of domains. *Information and Computation*, 127(2):66–90, June 1996.
- [Reh95] J. Rehof. Polymorphic dynamic typing — aspects of proof theory and inference. Master's thesis, DIKU, University of Copenhagen, March 1995.
- [Sal66] A. Salomaa. Two complete axiom systems for the algebra of regular events. *Journal of the Association for Computing Machinery (JACM)*, 13(1):158–169, 1966.