Coinductive Axiomatization of Recursive Type Equality and Subtyping

Michael Brandt
Prolog Development Center A/S
H. J. Holst Vej 3-5A
DK-2605 Brøndby, Denmark
Email: michael@pdc.dk

Fritz Henglein
DIKU
University of Copenhagen
DK-2100 Copenhagen, Denmark
Email: 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). It states that from $A, P \vdash P$ one may deduce $A \vdash P$, where $P$ is either a type equality $\tau = \tau'$ or type containment $\tau \leq \tau'$ and the proof of the premise must be contractive in a sense we define in this paper. In particular, a proof of $A, P \vdash P$ using the assumption axiom is not contractive. The fixpoint rule embodies a finitary coinduction principle and thus allows us to capture a coinductive relation in the fundamentally inductive framework of inference systems.

The new axiomatizations are more concise 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

---

*This work was partially supported by Danish Research Council Project DART. The results were obtained and written up at DIKU, University of Copenhagen.*
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.

More generally, we show how adding the fixpoint rule makes it possible to characterize inductively a set that is coinductively defined as the kernel (greatest fixed point) of an inference system.

**Keywords:** subtyping, type equality, recursive type, coercion, coinduction, operational interpretation, axiomatization, inference system, inference rule, fixpoint

1. **Introduction**

The simply typed \(\lambda\)-calculus is paradigmatic for both type inference for programming languages and the Curry-Howard isomorphism. Its typing rules are given by

\[
A, x : \tau, B \vdash x : \tau \quad \text{if } x \notin B
\]

\[
\frac{A, x : \tau \vdash e : \tau'}{A \vdash \lambda x : \tau. e : \tau \rightarrow \tau'}
\]

\[
\frac{A \vdash e : \tau \rightarrow \tau' \quad A \vdash e' : \tau}{A \vdash ee' : \tau'}
\]

Whereas adding recursive 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” property of \(\lambda\)-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 1.1.** The *recursive types (in canonical form)* $\mu T\alpha$ 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 $\alpha$ ranges over an infinite set $T\text{Var}$ of *type variables*, $\mu$ binds its type variable, recursive types that differ only in their bound variables are identified, and in every $\mu \alpha . \tau$ the bound variable $\alpha$ occurs freely in $\tau$.

Intuitively, $\mu \alpha . \tau$ denotes the recursive type defined by the type equation $\alpha = \tau$ (note that $\alpha$ occurs in $\tau$), $\bot$ is contained in all other types, and $\top$ contains all other types. We let $\tau, \sigma$ range over recursive types and write $fv(\tau)$ for the set of free type variables in $\tau$.

Our results extend to other types and type constructors such as product and sum, but for clarity we shall not treat these extensions here.

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].) We write $T_1 \rightarrow T_2$ for the tree $T$ with root label $\rightarrow$, left subtree $T_1$ and right subtree $T_2$.

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

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

- $T|_0^0 = \bot$
- $(T' \rightarrow T')|_k^k = T'|_k \rightarrow T'|_k$
- $\bot|_{k+1} = \bot$
- $\top|_{k+1} = \top$
- $\alpha|_{k+1} = \alpha$
- $T|_k^0 = \top$
- $(T' \rightarrow T')|^k = T'|_k \rightarrow T'|_k$
- $\bot|^k = \bot$
- $\top|^k = \top$
- $\alpha|^k = \alpha$

For tree $T$, $\mathcal{L}(T)$, the *label* of $T$, is the label of the root node of $T$: 

$$\mathcal{L}(\bot) = \bot$$
$$\mathcal{L}(\rightarrow) = \rightarrow$$
$$\mathcal{L}(\top) = \top$$
$$\mathcal{L}(\alpha) = \alpha$$

The label of a recursive type $\tau$ is the label of the tree it denotes: $\mathcal{L}(\tau) = \mathcal{L}(\text{Tree}(\tau))$. 
\[
\begin{array}{c}
\vdash \tau = \tau \\
\vdash \tau = \tau' \quad \vdash \tau' = \tau'' \quad \vdash \tau'' = \tau \\
\hline
\vdash \tau = \tau' \\
\vdash \sigma = \sigma' \\
\vdash \tau \rightarrow \sigma = \tau' \rightarrow \sigma' \\
\vdash \mu \alpha. \tau = \mu \alpha. \tau' \\
\end{array}
\]

\[ (\mu\text{-Compat}) \]

\[ (\text{Fold/Unfold}) \]

\[ (\tau \text{ contractive in } \alpha) \]

\[ (\text{Contract}) \]

Figure 1. Classical axiomatization of recursive type equality

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, \( \tau, \tau' \) are \textit{semantically equivalent} (denote the same ideal) if and only if Tree(\( \tau \)) = Tree(\( \tau' \)).

- \textit{Weak type equality}, the congruence generated by axiom \textit{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 (\textit{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 Tree(\( \tau \)) = 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], Milner [Mil84] and Kozen [Koz94].

All these axiomatizations are a variant of the inference system presented in Figure 1.\(^1\) (In Rule \textit{Contract}, recursive type \( \tau \) is contractive in type variable \( \alpha \) if \( \alpha \) occurs in \( \tau \) only under \( \rightarrow \), if at all.)

\(^1\) Instead of Rule \textit{Contract} Ariola and Klop [AK95] use the equivalent rule

\[ \vdash \tau_1 = \tau_1[\alpha/\alpha] \quad \mu \alpha. \tau = \tau_1 \quad \text{\( \tau \) contractive in } \alpha. \]
\[
\Gamma \vdash \bot \leq \tau \quad \Gamma \vdash \tau \leq \top
\]

\[
\Gamma \vdash \tau \leq \tau' \quad \Gamma \vdash \tau \leq \tau'' \quad \Gamma \vdash \tau = \tau' \quad \Gamma \vdash \tau \leq \tau'
\]

\[
\Gamma, \tau \leq \tau', \Gamma' \vdash \tau \leq \tau'
\]

\[
\Gamma \vdash \tau' \leq \tau \quad \Gamma \vdash \sigma \leq \sigma' \quad \Gamma, \alpha \leq \beta \vdash \tau \leq \sigma \quad \Gamma \vdash \mu \alpha. \tau \leq \mu \beta. \sigma \quad (\alpha, \beta \text{ not free in } \sigma, \tau)
\]

Figure 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 \( \mu \)-free types (to be thought of as finite trees) defined by

\[
\bot \leq_{\text{fin}} T \quad (\bot_{\text{fin}}) \\
T \leq_{\text{fin}} \top \quad (\top_{\text{fin}})
\]

\[
T \leq_{\text{fin}} T \quad (\text{REF}_{\text{fin}}) \\
S_1 \leq_{\text{fin}} T_1 \quad T_2 \leq_{\text{fin}} S_2 \quad S_1 \rightarrow S_2 \quad (\text{ARROW}_{\text{fin}})
\]

in a natural fashion to infinite trees.

**Definition 1.2. (Amadio/Cardelli subtype relation)** Let \( \tau, \sigma \) be recursive types. Define \( \tau \leq_{AC} \sigma \) if \( \text{Tree}(\tau)\mid_k \leq_{\text{fin}} \text{Tree}(\sigma)\mid_k \) 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 1.1.** \( T\mid_k \leq_{\text{fin}} T'\mid_k \) if and only if \( T\mid_k \leq_{\text{fin}} T'\mid_k \).

Alternatively, we could simply define depth-\( k \) approximations by mapping every node in a tree at level \( k \) to \( \bot \) (or \( \top \) for that matter). In either case the same subtype relation is defined.

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.
\[ A \vdash \bot \leq \tau \quad (\bot) \]
\[ A \vdash \tau \leq \top \quad (\top) \]
\[ A \vdash \tau \leq \tau \quad (\text{REF}) \]
\[ \frac{A \vdash \tau \leq \delta \quad A \vdash \delta \leq \sigma}{A \vdash \tau \leq \sigma} \quad (\text{TRANS}) \]
\[ A \vdash \mu \alpha.\tau \leq \tau[\mu\alpha.\tau/\alpha] \quad (\text{UNFOLD}) \quad A \vdash [\mu\alpha.\tau/\alpha] \leq \mu\alpha.\tau \quad (\text{FOLD}) \]
\[ A, \tau \leq \sigma, A' \vdash \tau \leq \sigma \quad (\text{HYP}) \]
\[ \frac{A, \tau_1 \rightarrow \tau_2 \leq \sigma_1 \rightarrow \sigma_2 \vdash \sigma_1 \leq \eta \quad A, \tau_1 \rightarrow \tau_2 \leq \sigma_1 \rightarrow \sigma_2 \vdash \tau_2 \leq \sigma_2}{A \vdash \tau_1 \rightarrow \tau_2 \leq \sigma_1 \rightarrow \sigma_2} \quad (\text{ARROW/FIX}) \]

Figure 3. Coinductive axiomatization of Amadio/Cardelli subtyping

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

\[ \frac{A \vdash \sigma_1 \leq \tau_1 \quad A \vdash \tau_2 \leq \sigma_2}{A \vdash \eta_1 \rightarrow \tau_2 \leq \sigma_1 \rightarrow \sigma_2} \quad (\text{ARROW}) \]
\[ \frac{A, \tau \leq \tau' \vdash \tau \leq \tau'}{A \vdash \tau \leq \tau'} \quad (\text{FIX}) \]

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 is \textit{sound and complete} for Amadio/Cardelli subtyping: \( \vdash \tau \leq \tau' \) if and only if \( \tau \leq_{AC} \tau' \). Because of Rule \text{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 \((\tau, \tau')\) and succeeds whenever \( \tau \leq_{AC} \tau' \). 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, 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 \textit{defined} in terms of subtyping since \( \tau \approx \tau' \) if and only if \( \tau \leq_{AC} \tau' \) and \( \tau' \leq_{AC} \tau \) 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 \text{CONTRACT} nor Rule \( \mu\)-\text{COMPAT}, which are difficult to interpret denotationally and operationally.
\[ A, \tau = \tau', A' \vdash \tau = \tau' \quad A \vdash \tau = \tau \]

\[
\frac{A \vdash \tau = \tau'}{A \vdash \tau' = \tau}
\]

\[
\frac{A \vdash \tau = \tau' \quad A \vdash \tau' = \tau''}{A \vdash \tau = \tau''}
\]

\[ A \vdash \mu \alpha. \tau = \tau[\mu \alpha. \tau/\alpha] \]

\[
\frac{A, \tau \rightarrow \tau' = \sigma \rightarrow \sigma' \vdash \tau = \sigma \quad A, \tau \rightarrow \tau' = \sigma \rightarrow \sigma' \vdash \tau' = \sigma'}{A \vdash \tau \rightarrow \tau' = \sigma \rightarrow \sigma'}
\]

Figure 4. Coinductive axiomatization of recursive type equality

1.6. Overview of the paper

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

Our coinductive axiomatizations do not only support direct coinductive reasoning, but also provide a natural foundation for a proof theory 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.

In Section 4 we discuss why and in which precise sense our axiomatization is coinductive. In this process we give a general recipe for axiomatizing finitarily coinductive relations. Finally, Section 5 describes the present, related and 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 2.1. (Simulation on recursive types) A simulation (on recursive types) is a binary relation \( R \) on recursive types satisfying:

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

Lemma 2.1. $\leq_{AC}$ is a simulation.

Proof:
We prove the four properties of Definition 2.1.

(i) Assume $\tau_1 \rightarrow \tau_2 \leq_{AC} \sigma_1 \rightarrow \sigma_2$ and let $k \in \mathbb{N}_0$. By Definition 1.2 we have
\[\text{Tree}(\tau_1 \rightarrow \tau_2)|_{(k+1)} \leq_{\text{fin}} \text{Tree}(\sigma_1 \rightarrow \sigma_2)|_{(k+1)}\]
By definition of Tree(·) 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 to be 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 1.1) we finally obtain $\sigma_1 \leq_{AC} \tau_1$ and $\tau_2 \leq_{AC} \sigma_2$ as desired.

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

(iii) Exactly as (ii).

(iv) Let $\tau \leq_{AC} \sigma$ and thus $\text{Tree}(\tau)|_k \leq_{\text{fin}} \text{Tree}(\sigma)|_k$ for all $k \in \mathbb{N}_0$. By inspection of $\leq_{\text{fin}}$ 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 2.2. If $\mathcal{R}$ is a simulation then $\tau \mapsto \tau \mapsto \tau \sigma \Rightarrow \tau \leq_{AC} \sigma$ for all $\tau, \sigma \in \mu Tp$.

Proof:
We prove $\forall k \in \mathbb{N}_0. \forall \tau, \sigma \in \mu Tp. (\tau \mapsto \tau \mapsto \tau \sigma \Rightarrow \tau \leq_{AC} \sigma)$ by induction on $k$.

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

Case $k > 0$: Let $\tau, \sigma$ be given such that $\tau \mapsto \tau \sigma$. We perform a case analysis on the syntactic forms of $\tau, \sigma$ where $\mathcal{L}(\tau) \leq \mathcal{L}(\sigma)$. The only possible combinations of $\tau$ and $\sigma$ are

<table>
<thead>
<tr>
<th>$\tau$</th>
<th>$\sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\bot$</td>
<td>$\sigma$</td>
</tr>
<tr>
<td>$\tau$</td>
<td>$\top$</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>$\alpha$</td>
</tr>
<tr>
<td>$\tau_1 \rightarrow \tau_2$</td>
<td>$\sigma_1 \rightarrow \sigma_2$</td>
</tr>
<tr>
<td>$\mu\alpha.\tau'$</td>
<td>$\sigma_1 \rightarrow \sigma_2$</td>
</tr>
<tr>
<td>$\tau_1 \rightarrow \tau_2$</td>
<td>$\mu\beta.\sigma'$</td>
</tr>
<tr>
<td>$\mu\alpha.\tau'$</td>
<td>$\mu\beta.\sigma'$</td>
</tr>
</tbody>
</table>
Case $\tau = \bot$ or $\sigma = \top$ or $\tau = \alpha, \sigma = \alpha$: Trivial.

Case $\tau = \mu \alpha. \tau', \sigma = \sigma_1 \rightarrow \sigma_2$: $\tau$ is canonical so $\tau'[\tau/\alpha] = \tau_1 \rightarrow \tau_2$ for some $\tau_1, \tau_2$. By Definition 2.1 (ii) we have $(\tau_1 \rightarrow \tau_2) R (\sigma_1 \rightarrow \sigma_2)$ and thus by (i) $\sigma_1 R \tau_1, \tau_2 R \sigma_2$. Our induction hypothesis yields

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

By Proposition 1.1 and Rule $\text{ARRROW}_{\text{fin}}$ 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. \hfill \Box

**Theorem 2.1. (Characterization of Amadio/Cardelli subtyping)**

$\tau \leq_{\text{AC}} \sigma$ if and only if there exists a simulation $R$ such that $\tau R \sigma$.

**Proof:**

Follows by Lemma 2.1 and Lemma 2.2. \hfill \Box

### 2.2. Soundness

We might want to interpret a sequent $\sigma_1 \leq \sigma_{12}, \ldots, \sigma_{n1} \leq \sigma_{n2} \vdash \tau \leq \tau'$ conventionally as “if $\sigma_1 \leq_{\text{AC}} \sigma_{12}, \ldots, \sigma_{n1} \leq_{\text{AC}} \sigma_{n2}$ then $\tau \leq_{\text{AC}} \tau''$” and prove every inference rule sound under this interpretation.

The problem is that Rule $\text{ARRROW/FIX}$ — more specifically the part that corresponds to Rule $\text{FIX}$ — is unsound under this interpretation! To see this, consider for example $(\bot \rightarrow \top) \leq (\top \rightarrow \bot) \vdash \top \leq \bot$. Since $\bot \rightarrow \top \not\leq_{\text{AC}} \top \rightarrow \bot$ it is vacuously valid under the conventional interpretation. Application of Rule $\text{ARRROW/FIX}$ lets us deduce $\vdash \bot \rightarrow \top \leq \top \rightarrow \bot$, 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 $(\bot \rightarrow \top) \leq (\top \rightarrow \bot) \vdash \top \leq \bot$, 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 2.2. (Stratified sequent interpretation)** Let $k$ range over the nonnegative integers. Define:

1. $\vdash_k \tau \leq \tau'$ 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. $A \models \tau \leq \tau'$ 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 $k \in \mathbb{N}_0$.

We write $\models \tau \leq \tau'$ instead of $A \models \tau \leq \tau'$ if $A$ is empty.

Note that $\bot \rightarrow \top \leq \top \rightarrow \bot \leq \bot$ does not hold under this interpretation; that is, $(\bot \rightarrow \top) \leq (\top \rightarrow \bot) \nvdash \top \leq \bot$. To wit, we have $(\bot \rightarrow \top) \leq (\top \rightarrow \bot) \models_0 \top \leq \bot$ and $(\bot \rightarrow \top) \leq (\top \rightarrow \bot) \models_k \top \leq \bot$ for all $k \geq 2$, but $(\bot \rightarrow \top) \leq (\top \rightarrow \bot) \nmodels_1 \top \leq \bot$ since $\text{Tree}(\bot \rightarrow \top)|_1 = \top \rightarrow \bot = \text{Tree}(\top \\leq \bot)|_1$, yet $\text{Tree}(\bot)|_1 = \top \nmodels \bot = \text{Tree}(\bot)|_1$. Intuitively, a sequent $A \models \tau \leq \tau'$ 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 2.3. (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 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

$$
A, \tau_1 \rightarrow \tau_2 \leq \sigma_1 \rightarrow \sigma_2 \models \sigma_1 \leq \tau_1 \quad \text{and} \quad A, \tau_1 \rightarrow \tau_2 \leq \sigma_1 \rightarrow \sigma_2 \models \tau_2 \leq \sigma_2.
$$

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 $\text{Tree}(\tau)|_0 = \bot$ for all $\tau$.

**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,

$$
\models_k A \text{ implies } \models_k \tau_1 \rightarrow \tau_2 \leq \sigma_1 \rightarrow \sigma_2.
$$

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.

\[ \square \]

**Theorem 2.2. (Soundness for Amadio/Cardelli subtyping)**

If $\vdash \tau \leq \tau'$ then $\tau \leq_{AC} \tau'$.

**Proof:**

Follows immediately from Lemma 2.3 and the observation that $\models \tau \leq \tau'$ if and only if $\tau \leq_{AC} \tau'$.

\[ \square \]
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 \( S \).

2.3.1. 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 \( \text{Hyp} \) is applied and otherwise rule \( \text{Arrow/Fix} \). It is of vital importance that assumptions are checked before applying the \( \text{Arrow/Fix} \) rule, since otherwise we would never be able to use them.

\begin{verbatim}
1:  S(A, \mu\alpha, \tau, \sigma) =
2:    let
3:      D_1 = \text{Unfold}
4:      D_2 = S(A, \tau[\mu\alpha/\alpha], \sigma)
5:    in
6:      \text{Trans}(D_1, D_2)
7:  end
8:  S(A, \tau, \mu\beta, \sigma) =
9:    let
10:   D_1 = S(A, \tau, \sigma[\mu\beta, \sigma/\beta])
11:   D_2 = \text{Fold}
12:  in
13:  \text{Trans}(D_1, D_2)
14: end
15: S((A, \tau \leq \sigma, A'), \tau, \sigma) = \text{Hyp}
16: S(A, \tau_1 \rightarrow \tau_2, \sigma_1 \rightarrow \sigma_2) =
17:    let
18:     A' = A \cup \{ \tau_1 \rightarrow \tau_2 \leq \sigma_1 \rightarrow \sigma_2 \}
19:     D_1 = S(A', \sigma_1, \tau_1)
20:     D_2 = S(A', \tau_2, \sigma_2)
21:   in
22:  \text{Arrow/Fix}(D_1, D_2)
23: end
24: S(A, \alpha, \alpha) = \text{Ref}
25: S(A, \bot, \tau) = \bot
26: S(A, \tau, \top) = \top
27: S(A, \tau, \sigma) = \text{exception}
\end{verbatim}

Figure 5. Algorithm \( S \)

2.3.2. Termination of \( S \)

Syntactic subterms We first introduce the concept of subterm closure. It corresponds to Kozen’s closure operator for \( L\mu \)-formulas [Koz82, p. 352], which, in turn, is motivated by the Fischer-Ladner closure for propositional dynamic logic [FL77]. Every recursive type has only a finite number of syntactic subterms, even though recursive types may have syntactic subterms that are larger than themselves; indeed, the number of syntactic subterms of a recursive type
is bounded by its size. For completeness we prove this fact here. We require a number of preliminary technical results.

**Definition 2.3.** A recursive type $\tau'$ is a *syntactic subterm* (or just *subterm*) of $\tau$ if $\tau' \subseteq \tau$, where $\subseteq$ is defined by the following rules:

\[
\begin{align*}
\tau & \subseteq \tau \quad \text{(REF)} \\
\frac{\tau \subseteq \sigma[\mu\alpha.\sigma/\alpha]}{\tau \subseteq \mu\alpha.\sigma} \quad \text{(UNFOLD)} \\
\frac{\tau \subseteq \sigma_1}{\tau \subseteq \sigma_1 \rightarrow \sigma_2} \quad \text{(ARROW_L)} \\
\frac{\tau \subseteq \sigma_2}{\tau \subseteq \sigma_1 \rightarrow \sigma_2} \quad \text{(ARROW_R)}
\end{align*}
\]

**Lemma 2.4.** The subterm relation is transitive, i.e. if $\tau \subseteq \delta$, $\delta \subseteq \sigma$ then $\tau \subseteq \sigma$.

**Proof:**
Induction on the derivation of $\delta \subseteq \sigma$. $\Box$

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

**Definition 2.4.** The *subterm closure* $\tau^*$ of $\tau$ is the set of recursive types defined by

\[
\begin{align*}
\bot^* & = \{\bot\} \\
\top^* & = \{\top\} \\
\alpha^* & = \{\alpha\}
\end{align*}
\]

\[
(\tau_1 \rightarrow \tau_2)^* = \{\tau_1 \rightarrow \tau_2\} \cup \tau_1^* \cup \tau_2^*
\]

\[
(\mu\alpha.\tau_1)^* = \{\mu\alpha.\tau_1\} \cup \tau_1^*[\mu\alpha.\tau_1/\alpha]
\]

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

**Proposition 2.1.** $|\tau^*| < \infty$.

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

**Lemma 2.5.** $(\tau'[\tau/\beta])^* = (\tau')^*[\tau/\beta] \cup \tau^*$ if $\beta \in \text{fv}(\tau')$.

**Proof:**
Induction on the structure of $\tau'$.

**Case $\tau' = \beta$:** The left-hand side evaluates to

\[
(\beta[\tau/\beta])^* = \tau^*
\]

and the right-hand side to

\[
\beta^*[\tau/\beta] \cup \tau^* = \{\beta\}[\tau/\beta] \cup \tau^* = \{\tau\} \cup \tau^* = \tau^*
\]

where we note that $\tau \in \tau^*$. 

Case $\tau' = \tau_1 \rightarrow \tau_2$: Since $\beta \in \text{fv}(\tau')$ it must occur free in either $\tau_1$ or $\tau_2$. The induction hypothesis is then that
\[(\tau_1[\tau/\beta])^* = \tau_1^*[\tau/\beta] \cup \tau^*\]
if $\beta \in \text{fv}(\tau_1)$. Assume that $\beta \in \text{fv}(\tau_1)$.
\[
((\tau_1 \rightarrow \tau_2)[\tau/\beta])^* = \\
((\tau_1[\tau/\beta]) \rightarrow \tau_2[\tau/\beta])^* = \\
\{ (\tau_1[\tau/\beta]) \rightarrow \tau_2[\tau/\beta] \} \cup (\tau_1[\tau/\beta])^* \cup (\tau_2[\tau/\beta])^* = \\
\{ (\tau_1 \rightarrow \tau_2)[\tau/\beta] \cup \tau_1^*[\tau/\beta] \cup \tau^* \cup \tau_2^*[\tau/\beta] = \\
\{ (\tau_1 \rightarrow \tau_2)[\tau/\beta] \cup \tau_1^*[\tau/\beta] \cup \tau^* = \\
(\tau_1 \rightarrow \tau_2)^*[\tau/\beta] \cup \tau^*. \\

The evaluation when $\beta \in \text{fv}(\tau_2)$ is similar.

Case $\tau' = \mu \alpha.\sigma$: We may assume that $\alpha \notin \text{fv}(\tau)$. The induction hypothesis is
\[(\sigma[\tau/\beta])^* = \sigma^*[\tau/\beta] \cup \tau^*.

Evaluation of the left-hand side
\[
((\mu \alpha.\sigma)[\tau/\beta])^* = \\
(\mu \alpha.\sigma[\tau/\beta])^* = \\
\{ (\mu \alpha.\sigma)[\tau/\beta] \} \cup (\sigma[\tau/\beta])^*[\mu \alpha.\sigma[\tau/\beta]/\alpha] = \\
\{ (\mu \alpha.\sigma)[\tau/\beta] \} \cup (\sigma^*[\tau/\beta] \cup \tau^*)[\mu \alpha.\sigma[\tau/\beta]/\alpha] = \\
\{ (\mu \alpha.\sigma)[\tau/\beta] \} \cup \sigma^*[\tau/\beta][\mu \alpha.\sigma[\tau/\beta]/\alpha] \cup \tau^*[\mu \alpha.\sigma[\tau/\beta]/\alpha] = \\
\{ (\mu \alpha.\sigma)[\tau/\beta] \} \cup \sigma^*[\mu \alpha.\sigma/\alpha][\tau/\beta] \cup \tau^* = \\
(\{ (\mu \alpha.\sigma) \} \cup \sigma^*[\mu \alpha.\sigma/\alpha])[\tau/\beta] \cup \tau^* = \\
(\mu \alpha.\sigma)^* \cup \tau^*.

Using this property we can show that $\tau^*$ contains all syntactic subterms of $\tau$:

**Lemma 2.6.** If $\tau \subseteq \sigma$ then $\tau \in \sigma^*$.

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

Case UNFOLD: So $\sigma = \mu \alpha.\tau'$ and $\tau \subseteq \tau'[\mu \alpha.\tau'/\alpha]$. By IH we get $\tau \in (\tau'[\mu \alpha.\tau'/\alpha])^*$. By Lemma 2.5 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 $(\mu \alpha.\tau')^*$. □
\[
|\tau^*| = 1 + |\tau_1^*| + |\tau_2^*| < \infty.
\]

Lemma 2.6 and Proposition 2.1 together finally give us the desired property:

**Theorem 2.3.** For recursive type $\tau$ the set $\{ \tau' \mid \tau' \subseteq \tau \}$ 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 2.3 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 2.5. The call tree of $S(A_0, \tau_0, \sigma_0)$ is defined to be a root node labeled $S(A_0, \tau_0, \sigma_0)$ whose subtrees are the call trees of all the recursive calls $S(A_i, \tau_i, \sigma_i)$ (finitely many) occurring in the first clause in S that matches $S(A_0, \tau_0, \sigma_0)$.

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

Theorem 2.4. Let $\tau_0, \sigma_0$ be recursive types and $A_0$ an assumption set. For all nodes $S(A_i, \tau_i, \sigma_i)$ in the call tree of $S(A_0, \tau_0, \sigma_0)$ we have that $\tau_i, \sigma_i$ are syntactic subterms of either $\tau_0$ or $\sigma_0$.

Proof:
Induction on the depth $d$ of nodes.

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

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

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

$$(\mu \alpha \cdot \tau \subseteq \tau_0 \land \sigma \subseteq \sigma_0) \text{ or } (\mu \alpha \cdot \tau \subseteq \sigma_0 \land \sigma_i \subseteq \tau_0).$$

It is easily seen that $\tau[\mu \alpha \cdot \tau/\alpha]$ is in canonical form. Assume $\mu \alpha \cdot \tau \subseteq \tau_0 \land \sigma \subseteq \sigma_0$. By Rule UNFOLD we have $\tau[\mu \alpha \cdot \tau/\alpha] \subseteq \mu \alpha \cdot \tau$ and thus by transitivity (Lemma 2.4) $\tau[\mu \alpha \cdot \tau/\alpha] \subseteq \tau_0$. The other subcase, $\mu \alpha \cdot \tau \subseteq \sigma_0 \land \sigma_i \subseteq \tau_0$, is similar.

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

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

1. $S(A', \sigma_1, \tau_1)$. By IH we know that $\tau_1 \rightarrow \tau_2 \subseteq \tau_0$ and $\sigma_1 \rightarrow \sigma_2 \subseteq \sigma_0$, or $\tau_1 \rightarrow \tau_2 \subseteq \sigma_0$ and $\sigma_1 \rightarrow \sigma_2 \subseteq \tau_0$. But then the result follows directly from transitivity (Lemma 2.4) since $\tau_1 \subseteq \tau_1 \rightarrow \tau_2$ and $\sigma_1 \subseteq \sigma_1 \rightarrow \sigma_2$ by Rule ARROWL.

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

Lemma 2.7. For any call path $S(A_0, \tau_0, \sigma_0), \ldots, S(A_i, \tau_i, \sigma_i), \ldots$ of $S(A_0, \tau_0, \sigma_0)$ we have

$$A_0 \subseteq A_1 \subseteq \ldots \subseteq A_i \subseteq \ldots.$$

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

$\square$
Lemma 2.8. For any call path \( S(A_0, \tau_0, \sigma_0), \ldots, S(A_n, \tau_n, \sigma_n) \ldots \) of \( S(A_0, \tau_0, \sigma_0) \) we have

\[
\exists N. \forall i. (\tau_i, \sigma_i) \in \{ (\tau_j, \sigma_j) \mid 0 \leq j \leq N \}.
\]

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

Proof:
The statement is proved by contradiction. Assume that

\[
\forall N \exists i : (\tau_i, \sigma_i) \not\in \{ (\tau_j, \sigma_j) \mid 0 \leq j \leq N \}
\]

This fact directly implies that \( \{ (\tau_j, \sigma_j) \mid j \in \mathbb{N}_0 \} \) is an infinite set. Theorem 2.4 states that all terms in a call tree are subterms of the initial two terms. We thus have

\[
\{ (\tau_j, \sigma_j) \mid j \in \mathbb{N}_0 \} \subseteq (\{ \tau_j \mid j \in \mathbb{N}_0 \}) \times (\{ \sigma_j \mid j \in \mathbb{N}_0 \}) \subseteq (\{ \tau \mid \tau \subseteq \tau_0 \} \cup \{ \sigma \mid \sigma \subseteq \sigma_0 \})^2
\]

Theorem 2.3, however, implies that

\[
|\{ (\tau_j, \sigma_j) \mid j \in \mathbb{N}_0 \}| \leq |\{ \tau \mid \tau \subseteq \tau_0 \} \cup \{ \sigma \mid \sigma \subseteq \sigma_0 \}|^2 < \infty
\]

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

The above results enable us to prove termination of \( S \).

Theorem 2.5. (Termination of \( S \)) If \( \tau, \sigma \) are canonical and \( A \) is an assumption set then \( S(A, \tau, \sigma) \) terminates.

Proof:
The proof is once again by contradiction. Assume that \( S(A, \tau, \sigma) \) does not terminate, i.e. there exists an infinite call path \( p \) in the call tree of \( S(A, \tau, \sigma) \). Let \( N \) be determined by Lemma 2.8 such that

\[
\forall i : (\tau_i, \sigma_i) \in \{ (\tau_j, \sigma_j) \mid 0 \leq j \leq N \}
\]

(1)

Let us consider the calls \((\tau_i, \sigma_i)\) of \( p \) where \( i > N \). There must exist a call \((\tau_n, \sigma_n)\) 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 2.4). 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 \( (\tau_m, \sigma_m) = (\tau_n, \sigma_n) \). The assumption set associated with call \( n \) inherits all assumptions from its ancestors in \( p \) (by Lemma 2.7), but then call \( n \) must be an application of \( \text{HYP} \), which corresponds to a leaf in the call tree. Path \( p \) is therefore not infinite and the assumption is false. \( \square \)
2.3.3. Correctness of $S$

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

**Lemma 2.9.** Let $\tau$, $\sigma$ be recursive types in canonical form and $A$ an assumption set. If $\tau \leq_{AC} \sigma$ then $S(A, \tau, \sigma)$ returns a derivation of $A \vdash \tau \leq \sigma$.

**Proof:**

The termination theorem (Theorem 2.5) gives that $S(A, \tau, \sigma)$ terminates with, say, $n$ recursive calls. Correctness is proved by induction on $n$. In each case we verify the derivation returned by $S$.

**Case** $n = 0$. No recursive calls performed at all. We perform a case analysis on those clauses in $S$ that have no recursive calls.

**Case** $S((A, \tau \leq \sigma, A'), \tau, \sigma)$, $S(A, \alpha, \alpha)$, $S(A, \bot, \tau)$ or $S(A, \tau, \top)$: Obvious.

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

**Case** $n > 0$. Our induction hypothesis is that computations $S(A', \tau', \sigma')$ with fewer than $n$ recursive calls, where $\tau' \leq_{AC} \sigma'$, produce a correct derivation of $A' \vdash \tau' \leq \sigma'$. We perform a case analysis of rules containing recursive calls.

**Case** $S(A, \mu \alpha. \tau, \sigma)$: Since $\leq_{AC}$ is a simulation (Lemma 2.1) we have $\tau[\mu \alpha. \tau/\alpha] \leq_{AC} \sigma$. Thus the induction hypothesis applies and gives that $S(A, \tau[\mu \alpha. \tau/\alpha], \sigma)$ 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 $S(A, \mu \alpha. \tau, \sigma)$ returns.

**Case** $S(A, \tau, \mu \beta. \sigma)$: Since $\leq_{AC}$ is a simulation (Lemma 2.1) we get $\tau \leq_{AC} \sigma[\mu \beta. \sigma/\beta]$. Thus the induction hypothesis is applicable and gives that $S(A, \tau, \sigma[\mu \beta. \sigma/\beta])$ returns a proof of $A \vdash \tau \leq \sigma[\mu \beta. \sigma/\beta]$. We conclude

\[
\frac{A \vdash \tau \leq \sigma[\mu \beta. \sigma/\beta]}{A \vdash \tau \leq \mu \beta. \sigma} \quad \frac{A \vdash \sigma[\mu \beta. \sigma/\beta] \leq \mu \beta. \sigma}{(\text{Fold})}
\]

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

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

1. $S(A', \sigma_1, \tau_1)$. 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. \(S(A', \tau_2, \sigma_2)\). As above, the call returns a proof of \(A' \vdash \tau_2 \leq \sigma_2\).

Consequently, \(S(A, \tau_1 \rightarrow \tau_2, \sigma_1 \rightarrow \sigma_2)\) returns Rule Arrow/Fix applied to the two subproofs above, which is a proof of \(A \vdash \tau_1 \rightarrow \tau_2 \leq \sigma_1 \rightarrow \sigma_2\).

\[\square\]

**Theorem 2.6. (Completeness)** If \(\tau \leq_{\text{AC}} \sigma\) then \(\vdash \tau \leq \sigma\).

**Proof:**

Follows from Lemma 2.9 for \(A = \emptyset\).

\[\square\]

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 a good idea 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

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

Each coercion can be interpreted as a function:
• $\iota_\tau$ denotes the identity on type $\tau$.
• $\text{fix } f : \tau \leq \tau'.c$ 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 $Ffx = c'(f(x))$.
• The pair fold$_{\mu_\alpha,\tau}$ and unfold$_{\mu_\alpha,\tau}$ denotes the isomorphism between $\tau[\mu_\alpha.\tau/\alpha]$ and $\mu_\alpha.\tau$.
• Both $\bot$ and $\top$ denote 1, the single point domain whose sole element we denote by $\bot_1$.
  We can think of 1 as containing only nonterminating computations, no real value. Then abort$_\tau$ maps any argument to a nonterminating computation at type $\tau$ and discard$_\tau$ maps any argument to a nonterminating computation at type $\bot$; that is, operationally both abort$_\tau$ and discard$_\tau$ simply enter an infinite loop for any argument.

Alternatively, we can think of 1 as containing the single value () and make sure that there are no nonterminating computations at $\bot$ or $\top$. This is simple, however, since, operationally, we can evaluate any expression of type $\bot$ or $\top$ by immediately returning ()

3.2. Well-typed coercions

Definition 3.1. (Contractiveness) A coercion $c$ is (formally) contractive in coercion variable $f$ if one of the following conditions holds:

1. $f$ does not occur in $c$.
2. $c \equiv c_1 \rightarrow c_2$.
3. $c \equiv c_1; c_2$ and both $c_1$ and $c_2$ are contractive in $f$.
4. $c \equiv \text{fix } g : \tau \leq \sigma. c_1$ and $c_1$ is contractive in $f$.

Definition 3.2. (Well-typed coercions, canonical coercions) Coercion $c$ is well-typed if $E \vdash c : \tau \leq \sigma$ is derivable for some $E, \tau, \sigma$ in the inference system of Figure 6.

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

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 $\tilde{E}$ 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 $\tilde{E} \vdash \tau \leq \tau'$ 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 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 3.1. We have $\vdash \tau \leq \tau'$ if and only if there exists a canonical coercion $c$ such that $\vdash c : \tau \leq \tau'$. 
$E \vdash \iota : \tau \leq \tau$

$E \vdash \text{abort}_\tau : \bot \leq \tau$

$E \vdash \text{discard}_\tau : \tau \leq \top$

$E \vdash \text{unfold}_{\mu \alpha, \tau} : \mu \alpha. \tau \leq \tau[\mu \alpha. \tau/\alpha]$

$E \vdash \text{fold}_{\mu \alpha, \tau} : \tau[\mu \alpha. \tau/\alpha] \leq \mu \alpha. \tau$

$E \vdash c : \tau \leq \delta. E \vdash d : \delta \leq \sigma$

$E \vdash c; d : \tau \leq \sigma$

$E \vdash (c \rightarrow d) : (\tau' \rightarrow \sigma) \leq (\tau \rightarrow \sigma')$

$E, f : \tau \leq \sigma, E' \vdash f : \tau \leq \sigma$

$E \vdash \text{fix } f : \tau \leq \sigma. c_f : \tau \leq \sigma$

Figure 6. Coercion typing rules

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 \( \text{fix } f : \sigma \leq \sigma' \) (\( f \) fresh) for suitable recursive types \( \sigma, \sigma' \) if it is not already the body of a \( \text{fix} \) -coercion. Once this is done, the transformed coercion corresponds directly to a derivation in Figure 3. \( \square \)

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. From coinductive definition to inductive definition

In this section we take a more general look at axiomatizing coinductively defined sets. In particular, we shall justify why we have called our axiomatization coinductive and why we refer to the fixpoint rule also as a coinduction principle. Our purpose is to present the general ideas, exemplified by transforming a coinductive definition of Amadio/Cardelli-subtyping systematically to an axiomatization featuring the fixpoint rule \( \text{Fix} \). That axiomatization is similar, but not identical to the ones presented in the previous sections. The transformation is intended to shed insight into the role of formal contractiveness and the way the fixpoint rule embodies a minimal, finitary coinduction principle. No proofs are given since corresponding results have been established in the previous sections.
\[ \vdash \bot \leq \tau \quad (\bot) \]

\[ \vdash \tau \leq \top \quad (\top) \]

\[ \vdash \tau \leq \tau \quad (\text{REF}) \]

\[ \vdash \sigma_1 \leq \tau_1 \quad \vdash \tau_2 \leq \sigma_2 \quad (\text{ARROW}) \]

\[ \vdash \tau[\mu \alpha, \tau/\alpha] \leq \sigma \quad (\text{UNFOLD'}) \]

\[ \vdash \mu \alpha \cdot \leq \sigma \quad (\text{FOLD'}) \]

Figure 7. Normalized subtyping inference rules

4.1. Coinductive characterization of Amadio/Cardelli subtyping

A set \( S \) is coinductively definable, or simply coinductive, if it is the greatest fixpoint of a monotonic operator \( \mathcal{F} \) under set containment: \( S = \bigcup \{ X \mid X \subseteq \mathcal{F}(X) \} \). The coinduction principle says that, in order to prove that \( P \subseteq S \) it is sufficient to prove \( P \subseteq \mathcal{F}(P) \).

Every inference system \( I \), viewed as a rule system in the sense of Aczel [Acz77], gives rise to a monotonic operator \( \mathcal{F}_I \). Consider the inference system \( I_\leq \) in Figure 7. Its associated operator \( \mathcal{F}_{\leq} \) maps sets of pairs of recursive types to sets of pairs of recursive types.

We define \( \leq_{\min} \) to be the set of pairs \((\tau, \sigma)\) such that there exists a derivation of \( \vdash \tau \leq \sigma \). Equivalently, we can define \( \leq_{\min} \) as the least fixpoint of \( \mathcal{F}_{\leq} \):

\[ \leq_{\min} = \text{lfp} \mathcal{F}_{\leq} = \bigcap \{ R \mid \mathcal{F}_{\leq}(R) \subseteq R \} \]

Let us call \( \leq_{\min} \) weak subtyping in analogy with weak type equality. (Indeed, if we strike out the rules \( \bot \) and \( \top \) we arrive at an axiomatization of weak type equality.)

The set \( \leq_{\max} \) coinductively defined by \( I_\leq \) is the greatest fixpoint of \( \mathcal{F} \):

\[ \leq_{\max} = \text{gfp} \mathcal{F}_{\leq} = \bigcup \{ R \mid R \subseteq \mathcal{F}_{\leq}(R) \} \]

Note that \( \leq_{\max} \) properly contains \( \leq_{\min} \); in particular, not every pair in \( \leq_{\max} \) has a finite derivation in \( I_\leq \). Aczel calls \( \leq_{\min} \) the set inductively defined by \( I_\leq \) [Acz77, Section 1.1] and \( \leq_{\max} \) the kernel of \( I_\leq \) [Acz77, Section 1.6].

Theorem 4.1. \((\tau, \sigma) \in \leq_{\max} \text{ if and only if } \tau \leq_{AC} \sigma \).

This theorem expresses that Amadio/Cardelli subtyping is definable by the inference system \( I_\leq \), though under its coinductive interpretation, not its standard inductive interpretation.

4.2. Infinitary axiomatization of Amadio/Cardelli subtyping

We can internalize the coinductive nature of \( \leq_{\AC} \) as follows: To prove that \( \tau \leq_{\AC} \sigma \) find a set of subtyping hypotheses \( R \) such that:
\[
\begin{align*}
R \vdash \bot \leq \tau \quad \text{(\bot)} & \quad R \vdash \tau \leq \top \quad \text{(\top)} \\
R \vdash \tau \leq \tau \quad \text{(REF)} & \quad \frac{R \vdash \sigma_1 \leq \tau_1 \quad R \vdash \tau_2 \leq \sigma_2}{R \vdash \tau_1 \rightarrow \tau_2 \leq \sigma_1 \rightarrow \sigma_2} \quad \text{(ARROW)} \\
\frac{R \vdash \tau[\mu\alpha.\tau/\alpha] \leq \sigma}{R \vdash \mu\alpha.\tau \leq \sigma} \quad \text{(UNFOLD')} & \quad \frac{R \vdash \sigma \leq \tau[\mu\alpha.\tau/\alpha]}{R \vdash \sigma \leq \mu\alpha.\tau} \quad \text{(FOLD')} \\
\frac{R \vdash^+ R}{\vdash R} \quad \text{(COIND)} & \quad \frac{R \cup \{\tau \leq \sigma\} \vdash \tau \leq \sigma}{\quad\text{(HYP)}} \\
\frac{R \vdash P \cup \{\tau \leq \sigma\}}{R \vdash \tau \leq \sigma} \quad \text{(^-ELIM)} & \quad \frac{R \vdash \tau \leq \sigma \quad (\forall(\tau \leq \sigma) \in P)}{R \vdash P} \quad \text{(^-INTRO)}
\end{align*}
\]

Figure 8. Normalized subtyping inference rules with coinduction principle

1. \((\tau, \sigma) \in R\), and
2. \(R \subseteq \mathcal{F}_\leq (R)\)

This is obviously a sound and complete proof principle for Amadio/Cardelli subtyping since we can choose \(R = \leq_{AC}\).

Since \(X = \mathcal{F}_\leq (X) \cup \ldots \cup F_i^j (X) \ldots\) for any fixed point of \(\mathcal{F}_\leq\) we can replace the second condition by the following rule-based coinduction principle:

2'. \(R \vdash^+ \tau' \leq \sigma'\) for all \(\tau' \leq \sigma' \in R\).

Here \(R \vdash^+ \tau' \leq \sigma'\) means that \(\tau' \leq \sigma'\) is derivable from \(I_\leq\) and the hypotheses \(R\), using at least one instance of an inference rule in \(I_\leq\). We say that a proof is formally contractive if it contains at least one application of one of the inference rules; that is, a formally contractive proof must not simply consist of a single invocation of one of the hypotheses.

We can add the coinduction principle as an infinitary inference rule to \(I_\leq\); see Figure 8. Here the notation \(\vdash^+\) in Rule COIND expresses that \(R \vdash \tau \leq \sigma\) must have a formally contractive proof for each \(\tau \leq \sigma \in R\); that is, the last step applied to prove each of the subtyping premises in \(R\) must not be by Rule HYP. At the cost of introducing judgements with potentially infinite sets of subtypings and the infinitary rule \(^\land\)-INTRO this inference system is sound and complete for deriving \(\leq_{AC}\)-subtypings: \(\tau \leq_{AC} \sigma\) if and only if there exists a derivation of \(\vdash \tau \leq \sigma\) in the inference system of Figure 8.

4.3. Finitary coinduction principle

We shall now show how we can get rid of infinite sets of subtypings and finally sets of subtypings altogether. This eventually results in an axiomatization similar to the one in Figure 6. The key
property we require is that \( \leq_{AC} \) is finitary in the following sense.

**Theorem 4.2.** \( \tau \leq_{AC} \sigma \) if and only if there exists a finite \( R \) such that:

1. \( (\tau, \sigma) \in R \) and
2. \( R \subseteq \mathcal{F}_{\leq}(R) \).

We can think of such an \( R \) as a (finite) witness to the fact that \( \tau \leq_{AC} \sigma \) for any \( (\tau, \sigma) \in R \). (This proposition also holds if we replace the second condition by requiring that \( R \) be a simulation. Note, however, that \( R \subseteq \mathcal{F}_{\leq}(R) \) for every simulation \( R \), but not every \( R \) such that \( R \subseteq \mathcal{F}_{\leq}(R) \) is a simulation.) An immediate consequence is that the inference system in Figure 8 remains complete when restricted to *finite* sets of subtypings both to the left and to the right of the turnstile. We can thus treat sets of subtypings as *finite conjunctions* of subtypings.

Figure 9 gives the resulting inference system. We have added explicit proof terms to suggest a natural operational interpretation of proofs as coercions. In particular, the coinduction principle is interpreted as a finite tuple of functions that are mutually recursively defined. Formal contractiveness is captured by requiring that in Rule \textit{COIND} none of the \( c_i \) must be a coercion variable \( f_j \).
\[ E \vdash \text{abort}_\tau : \bot \leq \tau \quad (\bot) \]
\[ E \vdash \text{discard}_\tau : \tau \leq \top \quad (\top) \]
\[ E \vdash \iota : \tau \leq \tau \quad \text{(Ref)} \]
\[ \frac{E \vdash c : \sigma_1 \leq \tau \quad E \vdash d : \tau_2 \leq \sigma_2}{E \vdash c \rightarrow d : \sigma_1 \rightarrow \sigma_2} \quad \text{(Arrow)} \]
\[ \frac{E \vdash c : \tau[\mu \alpha.\tau/\alpha] \leq \sigma}{E \vdash \text{unfold}_{\mu \alpha.\tau} c : \mu \alpha.\tau \leq \sigma} \quad \text{(Unfold')} \]
\[ \frac{E \vdash d : \sigma \leq \tau[\mu \alpha.\tau/\alpha]}{E \vdash \text{fold}_{\mu \alpha.\tau} d : \sigma \leq \mu \alpha.\tau} \quad \text{(Fold')} \]
\[ \frac{E, f : \tau \leq \sigma \vdash c : \tau \leq \sigma}{E \vdash \text{fix} f : \tau \leq \sigma} \quad \text{(Fix)} \]
\[ E_1, f : \tau \leq \sigma, E_2 \vdash f : \tau \leq \sigma \quad \text{(Hyp)} \]

Figure 10. Normalized subtyping inference rules with Fixpoint Rule

4.4. Gaussian elimination

The coercion interpretation of the coinduction principle in Figure 9 suggests how we can simplify the coinduction principle even further, without losing completeness. The term formulation of this proof rule reveals that it corresponds to (closed) mutually recursive definitions of coercions. Using Gaussian elimination, as in Bekic’s Theorem, mutually recursive definitions can be eliminated and replaced by nested, not necessarily closed, directly recursive definitions. (Note that the above observation amounts to a proof-theoretic application of Gaussian elimination.) Thus we can restrict the inference system even further without losing completeness: we replace Rule COIND by Rule FIX.

\[ \frac{E, f : \tau \leq \sigma \vdash c : \tau \leq \sigma}{E \vdash \text{fix} f : c : \tau \leq \sigma} \quad \text{(Fix)} \]

where \( c \) must not be a coercion variable. Now the introduction and elimination rules for conjunctions are superfluous for deriving judgements of the form \( E \vdash \tau \leq \sigma \) and can be eliminated completely. We end up with the inference system displayed in Figure 10. Note that this is the inference system \( I_\leq \) of Figure 7, enhanced with finite lists of subtyping assumptions and the attendant rule for invoking hypotheses, plus a single additional rule: the fixpoint rule.

5. Conclusion

5.1. Summary

We have given sound and complete axiomatizations of type equality and type containment using a novel fixpoint rule, Rule FIX. The fixpoint rule can be viewed as embodying a form of coinduction principle. It gives rise to a natural interpretation of proofs as coercions where the
fixpoint rule corresponds to definition by recursion. Finally, we have discussed how coinductively
defined sets can be turned into inductively defined ones using the fixpoint rule.

5.2. Related work

5.2.1. Recursive type equality and subtyping

The present work was inspired by an observation that meant that the standard unification
closure algorithm [HK71, Hue76] (see [ASU86, Section 6.7] for a presentation) can be turned
into an axiomatization of recursive type equality simply by “adding” the type equation to be
proved to the assumptions in the premises; see Figure 4. Unification closure works by building
a bisimulation between two recursive types. This suggested early on considering recursive type
equality and recursive subtyping as coinductive notions characterized by finitary notions of
bisimulation and simulation, respectively. Fiore [Fio96] has made the semantic connections
between bisimulation, final coalgebras, attendant coinduction principles and equality of infinite
trees precise in a category-theoretic setting.

Amadio and Cardelli [AC91, AC93] have defined subtyping for recursive types and given
an axiomatization inspired by work relying on the Contract Rule [Sal66, Mil84]. They also
present an “algorithm”, which is the basis for efficient subtype checking and can be understood
as an inference system based on a finitary coinduction principle (analogous to Figure 9, though
without an operational interpretation). In contrast to our work the types $\mu x. \tau$ and $\tau[\mu x. \tau/x]$ are
identified in their work, which is tantamount to saying that the syntactic objects they operate
on are regular trees and, operationally, coercing between $\mu x. \tau$ and $\tau[\mu x. \tau/x]$ is a “no-op”. In
contrast, we treat $\mu x. \tau$ and $\tau[\mu x. \tau/x]$ as distinct syntactic objects that are semantically and
operationally related by a unique isomorphism, corresponding to the interpretation of the pair
of inference rules Fold and Unfold. In particular, coercions need not be interpreted by the
identity. In this fashion we can model $\mu x. \tau$ and $\tau[\mu x. \tau/x]$ as different data type representations
that can be used interchangeably, though at the cost of applying a coercion.

Abadi and Fiore develop a semantic interpretation of recursive type equality as isomorphisms by induction on the axiomatization given by Amadio and Cardelli [AC93]. They use
this interpretation to relate the semantics of FPC [Gun92] under identification of recursive type
equality to its semantics under isomorphism. Our coinductive axiomatization of recursive type
equality gives rise to an operational interpretation of type equalities as coercions (not pairs of
coercions). In our setting, there is semantically at most one coercion between any two types
and consequently — semantically — exactly one admissible isomorphism between isomorphic
types. In contrast, Abadi and Fiore consider all possible isomorphisms between types. They
are thus not restricted to single isomorphisms nor forced to characterize these in a particular
fashion. However, this generality seems to cause a problem with the semantic soundness of the
Contract rule.
5.2.2. Program logic

The fixpoint rule in its term form (see Rule Fix in Figure 6), is well known as a reasoning principle for proving partial correctness [Hoa70] and, in suitably adapted form, total correctness [Sok77, Nie85] of recursive procedures. In a denotational setting the fixpoint rule corresponds to the principle of Scott induction and is valid for so-called inclusive (also called admissible) relations, which can be thought of as the semantic analogue to partial correctness predicates.

Note that in program logic we start with a program and wish to prove properties about it, using (the term formulation of) the fixpoint rule, amongst other reasoning principles. In contrast to this we have arrived at the recursive definition interpretation of the fixpoint rule from the other direction: we have shown that a finitary coinductive set can be captured inductively using the fixpoint rule and, furthermore, every element in the set can be given a natural operational interpretation by interpreting the fixpoint rule as a recursive definition. We claim that one of our conceptual contributions is not only the fixpoint rule itself for reasoning about coinductive sets, but also the fact that its correct semantic interpretation is as the (least) fixed points of recursive definitions [Bra97].

5.2.3. Type theory

Coquand [Coq93] formulates a guarded induction principle for reasoning about infinite objects within Type Theory. Using Coquand's terminology, an expression of some ground datatype is a productive element if, intuitively, it reduces to a constructor applied to a list of expressions, and each of these expressions is recursively productive. In this sense a productive element can be said to describe a potentially infinite tree without "undefined" (nonterminating) subtrees. An expression of higher type is reducible if it is hereditarily productive; e.g. function \( f : A \rightarrow B \) is reducible for data types \( A, B \) if it maps every productive element of \( A \) to a productive element of \( B \). Requiring expressions to be reducible guarantees consistency since no nonterminating expressions can be defined. A (first-order) recursive definition \( f(x_1, \ldots, x_n) = e[f, x_1, \ldots, x_n] \) is guarded if every occurrence of \( f \) has arguments that do not contain occurrences of \( f \), and \( f \) occurs "under" at least one constructor. Every function definable by guarded recursion is reducible. Thus guarded recursion yields a sound reasoning principle, called guarded induction by Coquand.

Our fixpoint rule and its contractiveness requirement seem to be a close pendant to Coquand's induction principle and its guardedness requirement. A careful and precise comparison, however, remains to be done. On a more speculative note, both Coquand's and our work can be viewed as making steps towards extending the Curry-Howard program to recursively defined proofs and recursively defined propositions.

5.2.4. Coinductive reasoning

Observational congruence [Mor68] of programs is intuitively a coinductive notion since — disregarding nontermination for now — its dual, noncongruence, is an inductive, finitary notion:
two expressions are observationally noncongruent if there is a finite experiment under which the
two expressions give different, observable answers. It is thus not surprising that coinduction
principles play an important role in proving program properties such as program equivalences
[MT91, Gor95, HL95, Len96]. Indeed for many programming languages observational congruence can be characterized by a notion of bisimulation; see e.g. [Mil77, Abr90]. This means that,
both in theory and practice, observational congruences can be proved by establishing bisimulations based directly on the operational semantics of a language. These bisimulations are
usually neither finitary — quite the opposite — nor do they have, suggest or require operational
significance.

5.3. Future work

5.3.1. 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 [Bra97]. 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
show that the equational theory is verified in a standard denotational (cpo-based) interpretation
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. By viewing some of the coercion equations as operational
inequalities, we can show that operationally optimal coercions exist, are unique modulo the
remaining equalities and can be generated efficiently [Bra97].

5.3.2. Coercion theory for simply typed $\lambda$-calculus with recursive types

We would like to extend the equational theory for coercions to the typed lambda calculus
with embedded coercions in the style of [BTCGS91, CG90, Hen94, Reb95] 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].

5.3.3. Other coinduction axiomatizations

Other coinductively defined relations should be amenable to the program laid out in Section 4;
for example, regular Böhm tree equality [Hue96] and regular expression equality [Sal66, Koz94].
Such axiomatizations might not only be interesting in their own right, but could prove useful in
cases where the relations have or can be given a useful operational interpretation.
5.3.4. Data representation optimization

Coercion reduction by rewriting appears to be useful in representation optimization, such as boxing analysis [Jon95], for recursively defined types. To be practically useful this may, however, require admitting more powerful transformations, for example the isomorphism $S \times (T + U) \cong (S \times T) + (S \times U)$. Seeking a coinductive axiomatization of Kleene Algebras appears to be a useful theoretical step in this direction.

5.3.5. Extensions to richer type languages and systems

We have studied recursive types and subtyping within a type language of monomorphic types; that is, simple types extended with regular recursive types. It would be interesting to extend this study to richer type disciplines with polymorphism (predicative and impredicative), intersection types and object typing.

Acknowledgments

We would like to thank Martín Abadi, Luca Cardelli, Andrew Gordon, Furio Honsell, Gérard Huet, 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 suggested the notion of a level-stratified interpretation of judgements which is the basis of the soundness proof presented in this paper. (An alternative soundness proof can be found in Brandt [Bra97].) Furthermore, he has provided interesting and helpful 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


