Recursive Subtyping: Axiomatizations and Computational Interpretations

Master Thesis

Michael Brandt

DIKU, Department of Computer Science
University of Copenhagen

August 1997

ABSTRACT

We give new axiomatizations of type equality and subtyping for a type language with recursive types. The novelty of our approach is the use of coinductive principles and especially the coinductive rules,

$$\frac{A(\tau = \sigma) \vdash \tau = \sigma}{A \vdash \tau = \sigma} \qquad \frac{A(\tau \leq \sigma) \vdash \tau \leq \sigma}{A \vdash \tau \leq \sigma}$$

The essence of these rules is that you may assume what you are trying to prove (some restrictions apply). The intuition being that if you can not find hard evidence proving that the judgement is false then it must be true. Standard axiomatizations of equality use a variant of the contraction rule,

$$\frac{C(\tau) = \tau \quad C(\sigma) = \sigma \quad \text{for some contractive function } C}{\tau = \sigma}$$

Subtype axiomatizations are normally based on the contravariant subtype axiomatization for non–recursive types extended with a rule for recursive type equality exploiting the contraction rule.

Our coinductive approach has several advantages.

- The coinductive rule is conceptually simpler than the contraction rule and it is self-contained and strictly syntax directed.
- Natural operational interpretations (coercions) exist for the coinductive rule.
- Efficient and natural decision algorithms are easily constructed.
- The recursive subtype relation may be axiomatized without type equality.

We prove our axiomatizations sound and complete with respect to Amadio, Cardelli's.

Due to the coinductive rule are we able to construct a coercion language which encodes all subtype derivations and has a natural operational interpretation easily related to coercions in real programming languages. We develop a coherence theory of coercions and subtype derivations which is founded on a coinductive equality relation on coercions. Furthermore, we present a denotational semantics for coercions and we show coercion equality sound under this interpretation. Finally, we formulate an optimality criteria for coercions and show that optimal coercions exist for all subtype derivations and we give an explicit algorithm for finding them. This algorithm also works as a decision algorithm for the subtype relation.

Contents

Preface

This report is my Master Thesis in computer science at DIKU. The work presented here is based on the ideas and initial results published in the paper "Coinductive Axiomatization of Recursive Type Equality and Subtyping", [BH97] which is jointly authored with Fritz Henglein.

Acknowledgements

I would like to thank my supervisor Fritz Henglein for guiding and helping me thru all the difficulties that I have encountered during the work on this report.

Michael Brandt
Copenhagen, Denmark
August 1997

Chapter 1

Introduction

1.1 Recursive Types

Recursive data structures have been known for a long time ([Hoa75]) and have provided very expressive and natural constructs to many programming languages, *e.g.* Pascal, Prolog and ML. The most well-known recursive data structure is probably the recursive list. Below we present declarations of recursive integer lists in Pascal, Prolog[1] and ML.

```
TYPE
  list = ^cons;
  cons = RECORD
           head : integer;
           tail : list;
         END;

DOMAINS
  list = nil ; cons(integer,list)

DATATYPE
  list = nil | cons of integer * list
```

These type declarations are called *recursive* because they are self referencing (directly or indirectly). Note that they are all explicitly declared as new type names in the language in order to make the self reference. This is the way recursive data structures are incorporated in programming languages today, but there is an alternative. The idea is to have a binding constructor in the type system which provides a name for a type. In this way it is possible to reference the type being defined and thus enable recursive types. This type system is called *Recursive Types* and in this report we will use the recursive type system Tp,

$$\tau \equiv \bot \mid \top \mid \alpha \mid \tau \to \tau \mid \mu\alpha.\tau$$

where α denotes type variables and $\to$ binds stronger than μ. The recursive name binder is $\mu\alpha.\tau$ which binds the name α to $\mu\alpha.\tau$ in the scope of τ, such

[1] Typed Prolog, *e.g.* PDCprolog.

that τ may refer to itself. This binding can be described by two operations,

$$\begin{aligned} \mathbf{unfold}(\mu\alpha.\tau) &= \tau[\mu\alpha.\tau/\alpha] \\ \mathbf{fold}_{\mu\alpha.\tau}(\tau[\mu\alpha.\tau/\alpha]) &= \mu\alpha.\tau \end{aligned}$$

The transition from $\mu\alpha.\tau$ to $\tau[\mu\alpha.\tau/\alpha]$ and vice versa is made explicit by the non-trivial **unfold** and **fold** operations, which means that $\mu\alpha.\tau$ and $\tau[\mu\alpha.\tau/\alpha]$ are isomorphic rather than identical.

Example 1.1.1 (Recursive Lists) The recursive list type presented above can easily be expressed with recursive types.

$$\text{list} \equiv \mu\alpha.\top + (\text{integer} \times \alpha)$$

where the recursive type system is extended with sums, pairs and integers. Observe that this is *not* a type declaration, but merely an ordinary type in the type system. ◇

Example 1.1.2 (Recursive Objects) Recursive types play an important role in the field of object theory. Extend Tp with object types in the style of Abadi, Cardelli [AbCa96]

$$\tau \equiv [l_1 : \tau_1, \ldots, l_n : \tau_n] \mid \bot \mid \top \mid \alpha \mid \tau \to \tau \mid \mu\alpha.\tau$$

where $[\{l_i : \tau_i\}_n]$ is the type of an object with members l_i. One of the key features of objects is that object members may reference their host object, but what happens if one decides to create an object with a copy method which copies the object? Recursive types handle the situation nicely: $\mu\alpha.[\texttt{copy} : \alpha]$ ◇

We use a recursive type system as the foundation of our work because the concept is much simpler and it expresses the essence of recursive data structures more directly than type declarations does. Another important difference from most declaration based languages is that type equivalence and subtyping is done structurally instead of handled by declaration names.

We can restrict Tp to a subset called the *canonical forms* μTp,

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

where it is required that $\alpha \in \text{fv}(\tau_1 \to \tau_2)$ in the case of $\mu\alpha.(\tau_1 \to \tau_2)$. The main advantage is that we avoid difficulties concerning $\mu\alpha.\alpha$ and the like. We can justify the use of this restricted system because every recursive type $\tau \in \text{Tp}$ has a corresponding canonical form $\tau' \in \mu\text{Tp}$ such that τ and τ' are equivalent. The translation is based on two basic isomorphisms,

$$\mu\alpha..\alpha = \bot \qquad \mu\alpha..\mu\beta..\tau = \mu\gamma..\tau[\gamma/\alpha, \gamma/\beta]$$

A translation can directly be formulated from these. When we in the sequent refer to recursive types without direct reference to Tp or μTp, we implicitly refer to the canonical forms.

Recursive types are studied extensively by Amadio, Cardelli [AC91], Cardone,Coppo [CC91] and Abadi, Fiore in [AF96].

1.2 Subtyping

Static type checking of programming languages has many useful properties, but it may also create some unreasonable restrictions on the language. A restriction often encountered is that types represent discrete value domains even tough some may be related. Consider for instance the interrelated domains

$$\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$$

One often have a value of type $\mathbb{Z}$ and want to use it in some context requiring a value of type $\mathbb{R}$. It seems reasonable to allow such uses of integers. Subtyping is about expressing interdependencies between types thus enabling more flexible type systems.

Formally, subtyping introduces a hierarchy on the type system, a partial ordering. If τ is a subtype of σ it is denoted with $\tau \leq \sigma$. There are two standard approaches to subtyping which stem from two different interpretations of what subtypes are supposed to express. These interpretations are called embedding/projection subtyping and context adaption subtyping. The first interpretation says that a type τ is a subtype of σ if and only there exists an injective embedding ι from the domain of τ to the domain of σ. Since ι is injective there also exists a projection π from the domain of σ to the domain of τ, such that $\pi(\iota(x)) = x$ for $x : \tau$. This kind of subtyping leads to *covariant* function subtyping (covariant in both positions). Consider the subtyping of functions $\tau \to \sigma \leq \tau' \to \sigma'$ and assume $\tau \leq \tau'$, $\sigma \leq \sigma'$. In order for this to be an embedding/projection subtype we have to verify that there exists an embedding and a projection of the required form.

$$\iota_{\tau\to\sigma\leq\tau'\to\sigma'}(f)(x) = \iota_{\sigma\leq\sigma'}(f(\pi_{\tau\leq\tau'}(x))) \quad \text{for } f : \tau' \to \sigma' \text{ and } x : \tau'$$

$$\pi_{\tau\to\sigma\leq\tau'\to\sigma'}(g)(y) = \pi_{\sigma\leq\sigma'}(g(\iota_{\tau\leq\tau'}(y))) \quad \text{for } g : \tau \to \sigma \text{ and } y : \tau$$

where $\iota_{\tau\leq\tau'}$ and $\pi_{\tau\leq\tau'}$ are the embedding/projection pair for $\tau \leq \tau'$ and similar for $\sigma \leq \sigma'$. The embedding and projection for the function subtyping satisfies

$$\pi_{\tau\to\sigma\leq\tau'\to\sigma'}(\iota_{\tau\to\sigma\leq\tau'\to\sigma'}(f)) = f$$

so we conclude that the conjectured covariant function subtype is indeed an embedding/projection subtype. An example of embedding/projection subtyping is $\mathbb{N} \to \mathbb{N} \leq \mathbb{R} \to \mathbb{R}$.

The other subtype interpretation is called context adaption, which intuitively means that if you can *adapt* a value of type τ to a *context* expecting a value of type σ, then $\tau \leq \sigma$ is a valid subtyping. Such adapters are called coercions, since they coerce values. Function subtyping in this interpretation is *contravariant* (first position contravariant, second covariant) opposed to embedding/projection subtyping. Reconsider the function subtype, but now assume that $\tau' \leq \tau$ and $\sigma \leq \sigma'$. We can then make an adapter.

$$c_{\tau\to\sigma\leq\tau'\to\sigma'}(f)(x) = c_{\sigma\leq\sigma'}(f(c_{\tau'\leq\tau}(x))) \quad \text{for } f : \tau \to \sigma \text{ and } x : \tau'$$

where $c_{\tau'\leq\tau}$ and $c_{\sigma\leq\sigma'}$ are the coercions for $\tau' \leq \tau$ and $\sigma \leq \sigma'$. An example of context adaption subtyping is $\mathbb{R} \to \mathbb{N} \leq \mathbb{N} \to \mathbb{R}$. Note that this subtype does not hold for the embedding/projection style subtyping. The example shown

for embedding/projection is on the other hand not legal with context adaption subtyping, which shows that the two subtyping schemes are very different. It also shows that embedding/projection does not safely fulfill the paradigm saying that values of subtypes may be substitutted for values of supertypes (apply a function of type $\mathbb{N} \to \mathbb{N}$ embedded into $\mathbb{R} \to \mathbb{R}$ to the real value 5.3). For this reason and the fact that most literature on recursive subtyping is based on context adaption subtyping we have decided to use this style.

For Tp without recursive types we have the standard structural contravariant subtype relation,

$$\bot \leq \tau \qquad\qquad \tau \leq \top$$

$$\tau \leq \tau \qquad\qquad \frac{\tau \leq \tau' \qquad \tau' \leq \tau''}{\tau \leq \tau''}$$

$$\frac{\tau' \leq \tau \qquad \sigma \leq \sigma'}{\tau \to \sigma \leq \tau' \to \sigma'}$$

The first two axioms state that $\bot$ and $\top$, respectively, are the least and greatest types in the subtype hierarchy. The middle rules are standard reflexivity and transitivity. The final rule states the contravariant function subtyping as discusses above.

To capture the intuition of subtypes in a type system we need a subtyping rule of the form,

$$\frac{E \vdash e : \tau' \qquad \tau \leq \tau'}{E \vdash e : \tau} \quad \text{(Subtyping)}$$

In a type system you normally do not care about the actual representation of objects and values, but it is a fact that value representations are very tightly connected with value types. The coercions justifying the subtypes may therefore be very handy to state explicitly in the type system in order to translate between various value representations. The explicit subtyping rule is shown below.

$$\frac{E \vdash e : \tau' \qquad c : \tau \leq \tau'}{E \vdash c(e) : \tau} \quad \text{(Subtyping with Coercions)}$$

Many people have investigated subtyping, subtypes and coercions, *e.g.* [FM90, Reh96, Mit91, Ama90].

1.3 Recursive Subtyping

We have now presented both recursive types and subtypes/subtyping, but only separately. Very interesting aspects and problems surface when the two are combined and interact. To illustrate the key problem we actually only need to consider recursive type equivalence, which is a simpler problem than recursive subtyping. Consider the recursive types $\tau \equiv \mu\alpha.\bot \to \alpha$ and $\sigma \equiv \mu\alpha.\bot \to (\bot \to \alpha)$. They are apparently not structural identical and if we unfold them, say, n and m times respectively we get

$$\mathbf{unfold}^n(\tau) = \bot \to \ldots \to (\bot \to (\mu\alpha.\bot \to \alpha))$$

$$\mathbf{unfold}^m(\sigma) = \bot \to \ldots \to (\bot \to (\mu\alpha.\bot \to (\bot \to \alpha)))$$

They still do not agree structurally, but if we let $n, m \to \infty$ they certainly seem equal. For recursive subtyping we would expect $\tau' \equiv \mu\alpha.\top \to \alpha$ to be less than $\sigma' \equiv \mu\alpha.\bot \to (\bot \to \alpha)$, because

$$\mathbf{unfold}^\infty(\tau') = \top \to (\top \to \ldots$$

$$\mathbf{unfold}^\infty(\sigma') = \bot \to (\bot \to \ldots$$

and these infinite unfoldings are related by the finite contravariant subtype relation. As the example demonstrates there is a complicated relation between recursive structures. Many have worked with this interesting blend of recursive types and subtyping. Our primary inspiration comes from Amadio, Cardelli [AC93], but Cardone, Coppo [CC91] and Ariola, Klop [AK95] have also worked with equivalence of recursive types, which is very related as the example shows.

Another intriguing facet of recursive types and subtyping is coercions. For subtypes without recursive types it is rather trivial to construct coercions directly from the subtype relation, but when combining subtypes with recursive types it is not at all trivial.

1.4 Overview

We give an introduction to the field of recursive subtyping, including definitions of the equality and subtyping relations based on infinite regular trees and axiomatizations of these based on a contraction rule. Our primary theoretical contribution to the field is the alternative axiomatizations of equality and subtyping based on coinductive principles derived from simulation theory and graph unification theory. The coinductive rule provides the necessary expressive power to enhance the finite axiomatizations to cope with the recursive equality and subtype relations. It does so in a very direct and natural way which leads to significantly simpler axiomatizations than Amadio, Cardelli's. A consequence of these new axiomatizations is another major contribution of this work, namely the coercion theory derived as a proof theory for recursive subtyping. The coercions derived in this manner have a natural operational interpretation corresponding directly to ordinary programming constructs, thus enabling them as real programming language coercions. The coercion theory is proved reasonable in many ways: 1) A coherence theory relating coercions and subtype derivations is obtained via a coinductively axiomatized equality relation on coercions (same principles as recursive type equality). 2) A denotational interpretation of coercions is given and the coercion equality relation is proven sound in this model. The final contribution of this report is the theory of optimal coercions. We devise formal criteria for coercions to be optimal which intuitively mean that no unnecessary translations are performed (like fold, unfold) hence leading to more efficient coercions. We show that optimal coercions exist for all recursive types related by the subtype relation. We also present an explicit algorithm for computing them. To summarize, the main contributions of this work are

- Novel axiomatizations of recursive type equality and subtyping based on coinductive principles.
- A proof theory for subtype derivations called coercion theory.

- Optimal coercions constructed explicitly by algorithm.

The report is organized as follows.

Chap.	Contents
2	Introduction to finite/infinite regular trees. Recursive types represented as regular trees.
3	Definition of the recursive equality and subtyping relations as given by Amadio, Cardelli. We present a simulation based characterization of recursive subtyping.
4	We motivate and describe the original axiomatization of recursive equality and subtyping as presented by Amadio, Cardelli.
5	We develop coinductive axiomatizations of recursive equality and subtyping. Completeness and soundness is shown for subtype axiomatization.
6	Coercions are introduced as encodings of subtype derivations and as programs.
7	Coinductive coercion equality theory is presented and coherence with subtype derivations is proven.
8	We give denotational interpretation of coercions and prove the equality relation sound with respect to this interpretation.
9	A formal criterion for optimal coercions is formulated and we show existence of optimal coercions for all subtype derivations.
10	Related works are discussed.
11	Conclusion and future work.

1.5 Preliminaries

We introduce notations, conventions and fundamental mathematical results used in this report. Propositions and theorems are not proved. Material in this section should not be unfamiliar to the reader and is included merely for completeness and for fixing notation.

1.5.1 Notation

General notation conventions include

τ, σ, δ	Recursive types or types in general
α, β	Type variables
$\tau[\sigma/\alpha]$	Type substitution of σ for all free occurrences of α in τ
c, d	Coercions
f, g	Coercion variables
$c[d/f]$	Coercion substitution. Analogous to type substitution.

1.5.2 α-equivalence

All terms and types in this report are treated modulo α-equivalence, that is modulo renaming of bound variables (type or program variables). Substitution is of course constructed such that capturing of free variables is avoided. For a detailed discussion of the implications of α-equivalence and precise definitions of substitution we refer to [HS86].

1.5.3 Environments

An environment, usually denoted by E, is a partial mapping from variable names to various kinds of formulas (*e.g.* types or subtype judgements). The domain $\mathrm{dom}(E)$ of an environment is thus all the variable names it binds and the application $E(f)$, $f \in \mathrm{dom}(E)$, yields the formula bound to f in E. Environments are often written as tuples of the form

$$E = (f_1 : F_1, \ldots, f_n : F_n)$$

where $n \geq 0$, f_i are *distinct* variables and F_i are some kind of formulas. Empty environments are denoted by ϵ or $()$. Two environments E and E' may be concatenated to form a new environment EE' consisting of all the bindings in E and E'. In order for this operator to work properly E and E' are *not* allowed to bind the same variable. If they do, the concatenation is undefined. Concatenation is often used to show a part of an environment, *e.g.* $E_1(f : F)E_2$ denotes the environment composed by three environments E_1, $(f : F)$ and E_2. We sometimes need to compare environments so we define a notion of sub environment.

Definition 1.5.1 For environments E and E' we define

$$E \subseteq E' \quad \Leftrightarrow \quad \mathrm{dom}(E) \subseteq \mathrm{dom}(E') \text{ and for } f \in \mathrm{dom}(E) \text{ is } E(f) = E'(f)$$

$\diamond$

Note that this definition implies $(f : F, f' : F') = (f' : F', f : F)$, where $E = E'$ iff $E \subseteq E'$ and $E' \subseteq E$.

1.5.4 Assumption sets

Sets of assumptions, usually denoted by A, are widely used in the axiomatizations. Assumptions are written like environments (without identifiers),

$$A = (F_1, \ldots, F_n)$$

where F_i are some formulas to be assumed true. Concatenation of assumption sets is denoted by juxtaposition, just as for environments.

1.5.5 Domain theory

We give a brief introduction to domain theory and present the most important (to this report) theorems. For details we refer to any standard textbook on domain theory, *e.g.* [Gun92, Win93, Sco72].

Definition 1.5.2 A domain is a pair $(X, \leq)$ where X is a set and $\leq$ a binary relation satisfying,

Reflexive:	$\forall x \in X.\ x \leq x$
Transitive:	$\forall x, y, z \in X.\ (x \leq y\ \wedge\ y \leq z)\ \Rightarrow\ x \leq z$
Anti-Symmetry:	$\forall x, y \in X.\ (x \leq y\ \wedge\ y \leq x)\ \Rightarrow\ x = y$
Least Element:	$\exists \perp_X \in X.\ \forall x \in X.\ \perp_X \leq x$
Least Upper Bound:	$\forall \{x_i\} \in X.\ x_1 \leq x_2 \leq \ldots \leq x_n \leq \ldots$ there exists $\sqcup_i x_i \in X$ satisfying

1. $\forall i.\ x_i \leq \sqcup_i x_i$
2. $\forall y \in X.\ (\forall i.\ x_i \leq y)\ \Rightarrow\ \sqcup_i x_i \leq y$

The ordering of a domain is often left implicit if it is evident from context. Domains are also known as Complete Partial Orders (CPO) or bc-domains ◇

Definition 1.5.3 Let $(X, \leq_X)$ and $(Y, \leq_Y)$ be domains and $f : X \to Y$ be a function. We say that f is continuous if

1. f is monotone; $\forall x, x' \in X.\ (x \leq_X x')\ \Rightarrow\ f(x) \leq_Y f(x')$
2. f preserves limits; for all chains $\{x_i\} \in X.\ f(\sqcup x_i) = \sqcup f(x_i)$

◇

One of domain theory's main results is the theorem of fix-points.

Theorem 1.5.4 ***(Fix-point Theorem)*** *If $(X, \leq_X)$ is a domain and $f : X \to X$ is a continuous function then $\sqcup_i f^i(\perp_X)$ is the least fix-point of f, that is*

$$f(\sqcup_i f^i(\perp_X)) = \sqcup_i f^i(\perp_X) \qquad \forall x \in X.(f(x) = x) \Rightarrow \sqcup_i f^i(\perp_X) \leq x$$

where

$$f^i(x) = \underbrace{f(f(\ldots f(}_{i}x)))$$

Constructions

Domains may be constructed from other domains into more complex domains. Below we list the domain constructions we use.

Construction	Description
$X \to Y$	Domain of continuous functions. Pointwise ordering.
$X \stackrel{s}{\to} Y$	Domain of strict, continuous functions. A function f is strict if and only if $f(\perp) = \perp$. Same ordering as function domain.
$X \times Y$	Cartesian Product domain. Elementwise ordering.
$X + Y$	Disjoint Sum domain with extra $\perp$ element. Elementwise ordering.

1.5.6 Complete Metric Spaces

We give a very short introduction to complete metric spaces. Details are omitted, but interested readers are referred to any textbook on classic mathematical analysis, *e.g.* [Berg92].

Definition 1.5.5 A metric space is a pair (X, d) where X is a set and d is *metric* on this set satisfying for all $x, y, z \in X$,

1. $d(x, y) \geq 0$ and $d(x, y) = 0 \Leftrightarrow x = y$
2. $d(x, y) = d(y, x)$
3. $d(x, z) \leq d(x, y) + d(y, z)$

$\diamond$

Converging sequences are of special importance to metric spaces and our applications of them.

Definition 1.5.6 A sequence $\{x_i\}$ in a metric space (X, d) converges to $x \in X$ if

$$\forall \epsilon > 0.\ \exists N \geq 1.\ \forall n > N.\ d(x_n, x) \leq \epsilon$$

$\diamond$

A variant of general sequences are the so called Cauchy sequences,

Definition 1.5.7 A sequence $\{x_i\}$ is called a Cauchy sequence if

$$\forall \epsilon > 0.\ \exists N \geq 1.\ \forall n, m > N.\ d(x_n, x_m) \leq \epsilon$$

$\diamond$

Cauchy sequences are the foundation for complete metric spaces.

Definition 1.5.8 A metric space (X, d) is called complete if all its cauchy sequences are convergent. $\diamond$

Complete metric spaces have many useful properties. One that in particular serves us concerns fix points of contractive functions.

Definition 1.5.9 A function $f : X \to X$ on a metric space (X, d) is called *contractive* if there exists c such that $0 \leq c < 1$ and

$$d(f(x), f(x')) \leq c\ d(x, x') \text{ for all } x, x' \in X$$

$\diamond$

Theorem 1.5.10 ***(Banach's Fix-point Theorem)*** *Let (X, d) be a complete metric space. If $f : X \to X$ is a contractive function then f has a unique fixpoint, i.e.*

$$f(x) = x \qquad \forall x' \in X.\ (f(x') = x') \Rightarrow x = x'$$

Chapter 2

Recursive Types and Regular Trees

In this chapter we begin to explore the nature of recursive types. It is very common to depict expressions, and in particular type expressions, as trees, so it seems natural to pursue this direction with recursive type expressions as well.

We will first develop a formal understanding of trees (see [Cour83] for further details) and then translate recursive types to trees.

2.1 Trees

Definition 2.1.1 Consider an alphabet Σ and a rank function $\rho : \Sigma \to \mathbb{N}_0$ denoting a rank for each symbol. A tree $t : \mathbb{N}^* \to \Sigma$ is a partial function satisfying the following requirements:

1. If $\pi_1, \pi_2 \in \mathbb{N}^*$ and $\pi_1\pi_2 \in \mathrm{dom}(t)$ then $\pi_1 \in \mathrm{dom}(t)$.
2. Let $\pi \in \mathbb{N}^*$ and $i, j \in \mathbb{N}$. If $i \leq j$ and $\pi j \in \mathrm{dom}(t)$ then $\pi i \in \mathrm{dom}(t)$.
3. Let $\pi \in \mathbb{N}^*$. If $\pi \in \mathrm{dom}(()t)$ and $\rho(t(\pi)) = k$ then for $i \in \mathbb{N}$ we have $\pi i \in \mathrm{dom}(t) \Leftrightarrow i \leq k$

$\diamond$

We say that a tree t is finite (infinite) if $\mathrm{dom}(t)$ is finite (infinite). A subtree t' at node π of tree t is define by $t'(\pi') = t(\pi\pi')$, such that the root of t' becomes $t(\pi)$.

Example 2.1.2 Let $\Sigma = \{\bot_0, \to_2, \top_0\}$ and consider the tree

$$t(\pi) = \begin{cases} \to & \text{if } \pi = \epsilon \\ t(\pi') & \text{if } \pi = 1\pi' \\ \bot & \text{if } \pi = 2 \end{cases}$$

$t'(\pi) = t(1\pi)$ is a subtree of t. Actually $t' = t$. $\diamond$

T^Σ denotes the set of all trees over the ranked alphabet Σ. T^Σ has some interesting topological and order-theoretical properties.

Proposition 2.1.3 *Define d for T^Σ as*

$$d(t,s) = \begin{cases} 0 & t = s \\ 2^{-\delta(t,s)} & t \neq s \end{cases}$$

where

$$\delta(t,s) = \begin{cases} \infty & \text{if } t = s \\ \min\{|\pi| \mid \pi \in \mathrm{dom}(t) \cap \mathrm{dom}(s),\ t(\pi) \neq s(\pi)\} & \text{if } t \neq s \end{cases}$$

d is a metric and (T^Σ, d) is a complete metric space.

PROOF We prove that d is a metric. Requirements 1 and 2 of Definition 1.5.5 are trivially satisfied. To prove 3 assume that $d(t,s) > d(t,r) + d(r,s)$. The distance between two trees is determined by the shortest path to a node mismatch. If $d(t,r) < d(t,s)$ then $\delta(t,r) > \delta(t,s)$. Now, if our assumption is true, then both $d(t,r)$ and $d(s,r)$ are properly less than $d(t,s)$, which implies that $\delta(t,r) > \delta(t,s)$ and $\delta(r,s) > \delta(t,s)$. If no mismatches occur in t, r until below level $\delta(t,s)$ it means that t and r are identical to this level. The same holds for r and s, but then we may conclude that t and s must be identical to and including level $\delta(t,s)$, which indeed is a contradiction, because $\delta(t,s)$ is the level of mismatch in t, s. We thus conclude $d(t,s) \leq d(t,r) + d(r,s)$ and hence that d is a metric on T^Σ.

A metric space is complete if every Cauchy sequence converges. Let (t_i) be a Cauchy sequence, *i.e.*

$$\forall \epsilon > 0. \exists N. \forall n, m \geq N.\ d(t_n, t_m) \leq \epsilon$$

Consider the function $\varphi : \mathbb{N}_0 \to \mathbb{N}$ defined by

$$\varphi(k) = \max\{n_1, \ldots, n_{k+1}\}$$

where n_j are such that $\delta(t_{n_j}, t_m) \geq j$ for all $m \geq n_j$. Such n_j exists: To compute n_j let $\epsilon = 2^{-j}$. Because (t_i) is a Cauchy sequence we can find N such that

$$d(t_n, t_m) \leq \epsilon = 2^{-j}$$

for all $n, m \geq N$. Let $n_j = N$. If $t_{n_j} = t_m$ for some m then $\delta(t_{n_j}, t_m) = \infty$ and the property trivially holds. Assume $t_{n_j} \neq t_m$, then

$$d(t_{n_j}, t_m) = 2^{-\delta(t_{n_j}, t_m)} \leq 2^{-j}$$

and therefore $\delta(t_{n_j}, t_m) \geq j$ as required. Observe that

$$\delta(t_{\varphi(k)}, t_m) \geq k + 1 > k \quad \text{for all } m \geq \varphi(k)$$

and also that $k \leq h \ \Rightarrow\ \varphi(k) \leq \varphi(h)$.

We now present a candidate as limit of (t_i).

$$s(\pi) = t_{\varphi(|\pi|)}(\pi) \quad \text{if } \pi \in \mathrm{dom}(t_{\varphi(|\pi|)})$$

First we show that s actually is a tree and next that $\lim_i t_i = s$.

1. Prefix closed. Let $\pi_1, \pi_2 \in \mathbb{N}^*$ and $\pi_1\pi_2 \in \mathrm{dom}(s)$. Show that $\pi_1 \in \mathrm{dom}(s)$. We directly get that $\pi_1\pi_2 \in \mathrm{dom}(t_{\varphi(|\pi_1\pi_2|)})$ and furthermore that $\pi_1 \in \mathrm{dom}(t_{\varphi(|\pi_1\pi_2|)})$ since $t_{\varphi(|\pi_1\pi_2|)}$ is a tree. Assume $\pi_1 \notin \mathrm{dom}(t_{\varphi(|\pi_1|)})$. From the observation above we get

$$\delta(t_{\varphi(|\pi_1|)}, t_{\varphi(|\pi_1\pi_2|)}) > |\pi_1|$$

since $|\pi_1| \leq |\pi_1\pi_2|$ and hence $\varphi(|\pi_1|) \leq \varphi(|\pi_1\pi_2|)$, which means that no mismatches occur at level $|\pi_1|$. This cannot be true since we assumed that $\pi_1 \notin \mathrm{dom}(t_{\varphi(|\pi_1|)})$ and showed that $\pi_1 \in \mathrm{dom}(t_{\varphi(|\pi_1\pi_2|)})$. The assumption is therefore false and so $\pi_1 \in \mathrm{dom}(t_{\varphi(|\pi_1|)}) \subseteq \mathrm{dom}(s)$.

2. Let $\pi \in \mathbb{N}^*$, $i, j \in \mathbb{N}$ and $i \leq j$. If $\pi j \in \mathrm{dom}(s)$ then $\pi i \in \mathrm{dom}(s)$. Since $\pi j \in \mathrm{dom}(s)$ we know that $\pi j \in \mathrm{dom}(t_{\varphi(|\pi|+1)})$, but because $t_{\varphi(|\pi|+1)}$ is a tree the result follows directly.

3. Let $\pi \in \mathbb{N}^*$ and $\rho(s(\pi)) = k$. Show

$$\pi i \in \mathrm{dom}(s) \quad \Leftrightarrow \quad i \leq k$$

Because $\varphi(|\pi|) \leq \varphi(|\pi| + 1)$ we know that $\delta(t_{\varphi(|\pi|)}, t_{\varphi(|\pi|+1)}) > |\pi|$ and so $s(\pi) = t_{\varphi(|\pi|)}(\pi) = t_{\varphi(|\pi|+1)}(\pi)$. We conclude that $\rho(t_{\varphi(|\pi|+1)}(\pi)) = k$ and from this the result follows.

The final step is to show that $t_i \to s$. Let $\epsilon > 0$ be given. Compute $k \in \mathbb{N}_0$ such that $2^{-k} \leq \epsilon$ (this is definitely possible). Let $N = \varphi(k)$ which then satisfies $\delta(t_N, t_n) > k$ for $n \geq N$. We claim that $\delta(t_n, s) \geq k$ for $n \geq N$. To prove this assume oppositely that $k > \delta(t_n, s)$ for some $n \geq N$. This means that there exists path π such that $t_n(\pi) \neq s(\pi)$ and $k > |\pi|$.

$$\varphi(|\pi|) \leq \varphi(k) = N \leq n$$

By definition of $\varphi(|\pi|)$ we have $\delta(t_{\varphi(|\pi|)}, t_n) > |\pi|$ and thus $t_{\varphi(|\pi|)}(\pi) = t_n(\pi)$. Now, this cannot be true because $t_{\varphi(|\pi|)}(\pi) = s(\pi) \neq t_n(\pi)$. We conclude $k \leq \delta(t_n, s)$ for all $n \geq N$ and finally

$$d(t_n, s) = 2^{-\delta(t_n, s)} \leq 2^{-k} \leq \epsilon$$

If $t_n = s$ it trivially holds. $\diamond$

Recall that in a complete metric space X every contractive function $f : X \to X$ has a unique fix–point (Theorem 1.5.10). We can also view T^Σ as a domain.

Proposition 2.1.4 *Let Σ be an alphabet with an element $\perp_\Sigma$ of rank 0. Consider the order on trees $t, s \in \mathrm{T}^\Sigma$*

$$t \leq s \;\Leftrightarrow\; \mathrm{dom}(t) \subseteq \mathrm{dom}(s) \text{ and for } \pi \in \mathrm{dom}(t)\ (t(\pi) \neq \perp \Rightarrow t(\pi) = s(\pi))$$

With this partial order T^Σ is a domain with least element $\perp_{\mathrm{T}^\Sigma}(\epsilon) = \perp_\Sigma$.

PROOF We prove that $\leq$ on trees is a partial order.

1. Reflexive. Trivial.

2. Transitive. Let $t \leq s$ and $s \leq r$. Show $t \leq r$. $\mathrm{dom}(t) \subseteq \mathrm{dom}(s) \subseteq \mathrm{dom}(r)$. If $t(\pi) \neq \bot_A$ then $t(\pi) = s(\pi)$ since $t \leq s$, but then because $s \leq r$ and $s(\pi) \neq \bot_A$ we get $t(\pi) = s(\pi) = r(\pi)$.

3. Antisymmetry. If $t \leq s \leq t$ then $t = s$. Surely $\mathrm{dom}(t) = \mathrm{dom}(s)$. If $t(\pi) \neq \bot_A$ then $t(\pi) = s(\pi)$ and likewise if $s(\pi) \neq \bot_A$. On the other hand, if $t(\pi) = \bot_A$ then so is $s(\pi)$ because otherwise we infer $\bot_A \neq s(\pi) = t(\pi) = \bot_A$

A domain must have a least element, but $\bot(\epsilon) = \bot_A$ satisfies this requirement.

To prove the completeness property consider a chain $t_1 \leq t_2 \leq \ldots \leq t_i \leq \ldots$. We need to find a least upper bound for the chain.

$$(\sqcup t_i)(\pi) = \begin{cases} a & \text{if } t_j(\pi) = a \text{ for some } t_j \\ \bot_A & \text{if } t_j(\pi) = \bot_A \text{ for all } t_j \text{ with } \pi \in \mathrm{dom}(t_j) \end{cases}$$

$$\mathrm{dom}(\sqcup t_i) = \cup \mathrm{dom}(t_i)$$

It is easy to see that $\sqcup t_i$ is a tree. We show that it is an upper bound for the chain (t_i). For t_n we have $\mathrm{dom}(t_n) \subseteq \mathrm{dom}(\sqcup t_i)$ and if $t_n(\pi) \neq \bot_A$ then surely $(\sqcup t_i)(\pi) = t_n(\pi)$ since it is defined so. Note that $\pi \in \mathrm{dom}(t_n) \subseteq \mathrm{dom}(t_{n+1}) \subseteq \ldots$ implies that $t_n(\pi) = t_{n+1}(\pi) = \ldots$. To see that it is the *least* upper bound consider another upper bound s. We show that $\sqcup t_i \leq s$. Since s is an upper bound is has to include all the domains in the chain and therefore $\mathrm{dom}(\sqcup t_i) \subseteq \mathrm{dom}(s)$. Assume next that $(\sqcup t_i)(\pi) \neq \bot_A$, *i.e.* $(\sqcup t_i)(\pi) = t_j(\pi)$ for some t_j. Because s is an upper bound and $t_j(\pi) \neq \bot_A$ we find that $(\sqcup t_i)(\pi) = t_j(\pi) = s(\pi)$ and we are done. ◇

Even though trees generally are infinite we find that most trees still have a finite structure, especially those arising in natural applications. Such trees are given a distinguished name.

Definition 2.1.5 $t \in \mathrm{T}^\Sigma$ is *regular* if it contains only finitely many different subtrees. We denote the set of regular trees by T^Σ_{reg}. ◇

It is sometimes useful to consider finite approximations of infinite trees.

Definition 2.1.6 Let T^Σ be a domain with least element $\bot$ and the ordinary tree ordering. The k'th approximation of a tree $t \in \mathrm{T}^\Sigma$ is defined by

$$t|_k(\pi) = \begin{cases} \bot & \text{if } |\pi| = k \text{ and } \pi \in \mathrm{dom}(()t) \\ t(\pi) & \text{if } |\pi| < k \text{ and } \pi \in \mathrm{dom}(()t) \end{cases}$$

◇

It is easy to see that $t|_k$ indeed is a tree. To see that it also is an approximation consider the following proposition.

Proposition 2.1.7 $\lim_{k\to\infty} t|_k = t$

PROOF Let $\epsilon > 0$ be given. Find $N \in \mathbb{N}$ such that $d(t|_k, t) \leq \epsilon$ for all $k \geq N$. If t is finite then let N be the depth of t and then $d(t|_k, t) = 0 \leq \epsilon$ for $k \geq N$. If t is infinite then consider

$$\begin{array}{rcl} \delta(t|_k, t) & = & \left\{ \begin{array}{ll} \infty & \text{if } t|_k = t \\ \min\{|\pi| \mid t|_k(\pi) \neq t(\pi)\} & \text{if otherwise} \end{array} \right. \\ & \geq & k \end{array}$$

and hence $d(t|_k, t) = 2^{-\delta(t|k,t)} \leq 2^{-k}$. It is definitely possible to find N such that $2^{-k} \leq \epsilon$ for all $k \geq N$. ◇

2.2 Translation of Recursive Types

We define a translation of recursive types to regular trees and discuss some interesting properties of regular trees.

Proposition 2.2.1 *Let $\Sigma = \{\rightarrow, \bot, \top\} \cup$ Tyvar be an (infinite) alphabet and ρ the rank function yielding 2 for $\rightarrow$ and 0 for all other symbols. There exists a unique function Tree : $Tp \rightarrow \mathrm{T}^\Sigma$ such that,*

$$Tree(\tau_1 \rightarrow \tau_2)(\pi) = \left\{ \begin{array}{ll} \rightarrow & \text{if } \pi = \epsilon \\ Tree(\tau_1)(\pi') & \text{if } \pi = 1\pi' \\ Tree(\tau_2)(\pi') & \text{if } \pi = 2\pi' \end{array} \right.$$
$$Tree(\alpha)(\epsilon) = \alpha$$
$$Tree(\bot)(\epsilon) = \bot$$
$$Tree(\top)(\epsilon) = \top$$
$$Tree(\mu\alpha.\tau)(\pi) = Tree(\tau[\mu\alpha.\tau/\alpha])(\pi)$$

PROOF From Proposition 2.1.4 we learn that T^Σ is a domain, since $\bot \in \Sigma$ has rank 0. Note that Tp can be viewed as a flat domain with $\bot$ as least element. We can therefore construct the function domain $\mathrm{Tp} \rightarrow \mathrm{T}^\Sigma$. Consider the operator $F : (\mathrm{Tp} \rightarrow \mathrm{T}^\Sigma) \rightarrow (\mathrm{Tp} \rightarrow \mathrm{T}^\Sigma)$ defined by

$$\begin{array}{lll} F(f)(\tau_1 \rightarrow \tau_2)(\epsilon) = \rightarrow & & F(f)(\alpha)(\epsilon) = \alpha \\ F(f)(\tau_1 \rightarrow \tau_2)(1\pi) = f(\tau_1)(\pi) & F(f)(\bot)(\epsilon) = \bot & F(f)(\top)(\epsilon) = \top \\ F(f)(\tau_1 \rightarrow \tau_2)(2\pi) = f(\tau_2)(\pi) & F(f)(\mu\alpha.\tau)(\pi) = f(\tau[\mu\alpha.\tau/\tau])(\pi) & \end{array}$$

First, observe that F actually maps to the right domain $(\mathrm{Tp} \rightarrow \mathrm{T}^\Sigma)$ which implicitly means that the produced function is continuous (proof left to interested readers). Next, we show that F is continuous, *i.e.* 1) monotonic: $f \leq g \Rightarrow F(f) \leq F(g)$ and 2) limit conservative: $F(\sqcup_i f_i) = \sqcup_i F(f_i)$ for chain $\{f_i\}_i$.

Ad 1. Let $f, g \in \mathrm{Tp} \rightarrow \mathrm{T}^\Sigma$ be such that $f \leq g$, which means that $f(\tau) \leq g(\tau)$ for all $\tau \in \mathrm{Tp}$. We now need to show that $F(f)(\tau) \leq F(g)(\tau)$ for all $\tau \in \mathrm{Tp}$. Only interesting case is $\tau = \mu\alpha.\tau'$.

$$F(f)(\mu\alpha.\tau') = f(\tau'[\mu\alpha.\tau'/\alpha]) \leq g(\tau'[\mu\alpha.\tau'/\alpha]) = F(g)(\mu\alpha.\tau')$$

Ad 2. Now let $\{f_i\}$ be a chain, *i.e.* $f_1 \leq f_2 \leq \ldots$. Since $\mathrm{Tp} \to \mathrm{T}^\Sigma$ is a domain we know that the least upper bound of $\{f_i\}$ exists and we denote it with $\sqcup_i f_i$. Only $\mu\alpha.\tau'$ case shown.

$$F(\sqcup_i f_i)(\mu\alpha.\tau') = (\sqcup_i f_i)(\tau'[\mu\alpha.\tau'/\alpha]) = \sqcup_i(f_i(\tau'[\mu\alpha.\tau'/\alpha])) = \sqcup_i(F(f_i)(\mu\alpha.\tau'))$$

because the limit of functions is defined element wise and hence $(\sqcup_i f_i)(\tau) = \sqcup_i(f_i(\tau))$.

F is proven continuous and hence in the domain $(\mathrm{Tp} \to \mathrm{T}^\Sigma) \to (\mathrm{Tp} \to \mathrm{T}^\Sigma)$. By the fix–point theorem we get a well-defined fix–point function $\mathrm{Tree} \in \mathrm{Tp} \to \mathrm{T}^\Sigma$ such that $F(\mathrm{Tree}) = \mathrm{Tree}$. Let us examine Tree.

$$\mathrm{Tree}(\tau_1 \to \tau_2)(1\pi) = F(\mathrm{Tree})(\tau_1 \to \tau_2)(1\pi) = \mathrm{Tree}(\tau_1)(\pi)$$

$$\mathrm{Tree}(\mu\alpha.\tau)(\pi) = F(\mathrm{Tree})(\mu\alpha.\tau)(\pi) = \mathrm{Tree}(\tau[\mu\alpha.\tau/\alpha])(\pi)$$

and likewise for the remaining instances of τ and π. ◇

Note that Tree has an inverse for all *finite* trees and that the resulting type is non-recursive. We usually denote this inverse with Tree^{-1}.

Chapter 3

Recursive Equality and Subtyping

In this chapter we present definitions of recursive equality and subtyping based on trees. These definitions were originally formulated by Cardone, Coppo in [CC91] and Amadio, Cardelli in [AC91]. Recursive equality has been examined in [Bra97] and we will thus not get into details here, but merely present the original definition. Recursive subtyping as defined here is rather hard to reason about and therefore we develop a characterization based on simulations, which provide a much better foundation for axiomatizations.

3.1 Equality

As demonstrated in the introduction (Section 1.3) structural syntactical comparison is not adequate to serve as definition of equality – not even if finite unfolding of the types are permitted. This last variation is normally called *weak* equality (for obvious reasons). Cardone and Coppo [CC91] therefore realized that recursive equality should be based on *infinite* unfoldings, that is, on infinite (regular) trees.

Definition 3.1.1 Recursive types τ and σ are equal if and only if they denote the same infinite tree, *i.e.*

$$\tau =_\mu \sigma \;\Leftrightarrow\; \mathrm{Tree}(\tau) = \mathrm{Tree}(\sigma)$$

$\diamond$

With this definition of equality it is easily seen that the example from the introduction holds and thus that $=_\mu$ is properly stronger than weak equality.

3.2 Subtyping

In the introduction we showed a finite contravariant subtype relation for non-recursive types (see Figure 3.1). It will be the basis of the recursive subtype relation. Amadio, Cardelli [AC91] translated recursive types to regular trees

$$\bot \leq \tau \qquad\qquad \tau \leq \top$$

$$\tau \leq \tau \qquad\qquad \frac{\tau \leq \tau' \qquad \tau' \leq \tau''}{\tau \leq \tau''}$$

$$\frac{\tau' \leq \tau \qquad \sigma \leq \sigma'}{\tau \to \sigma \leq \tau' \to \sigma'}$$

Figure 3.1: Subtype relation of non-recursive types

and approximated them by finite trees. They then compared these finite trees or non-recursive types by the finite contravariant subtype relation. If all approximations were in the subtype relation they defined that the recursive types belonged to the recursive subtype relation.

Definition 3.2.1 We define three relations,

$$\begin{array}{lll} t \leq_{\text{fin}} s \;\Leftrightarrow\; \text{Tree}^{-1}(t) \leq \text{Tree}^{-1}(s) & \text{for finite trees } t,\, s. & \\ t \leq_{\infty} s \;\Leftrightarrow\; \forall k.\; t|_k \leq_{\text{fin}} s|_k & \text{for trees } t,\, s. & \text{The recur-} \\ \tau \leq_{\mu} \sigma \;\Leftrightarrow\; \text{Tree}(\tau) \leq_{\infty} \text{Tree}(\sigma) & \text{for recursive types } \tau,\, \sigma. & \end{array}$$

sive subtype relation is defined by $\leq_\mu$. ◇

Our definition is slightly different from Amadio, Cardelli's original, because our tree restriction $t|_k$ is based on $\bot$ cut-offs only. Amadio, Cardelli placed $\bot$ and $\top$ depending on the polarity of the tree path reaching the cut depth, but since we are only interested in the limit of the restricted trees it does not matter and we feel that our definition is simpler. Note that $=_\mu$ is embedded in $\leq_\mu$ since $\lim_{(k\to\infty)}(t|_k) = t$ according to Proposition 2.1.7. We illustrate the relations with an example.

Example 3.2.2 Consider the recursive types $\tau \equiv \mu\alpha.(\top \to \alpha)$ and $\sigma \equiv \mu\alpha.(\bot \to (\bot \to \alpha))$. We have informally

$$\text{Tree}(\tau) = (\top \to (\top \to \ldots$$

$$\text{Tree}(\sigma) = (\bot \to (\bot \to \ldots$$

The approximations of these trees are

k	$\text{Tree}^{-1}(\text{Tree}(\tau)\|_k)$	$\text{Tree}^{-1}(\text{Tree}(\sigma)\|_k)$
0	$\bot$	$\bot$
1	$\bot \to \bot$	$\bot \to \bot$
2	$\top \to (\bot \to \bot)$	$\bot \to (\bot \to \bot)$
n	$\top \to (\top \to \ldots \to (\bot \to \bot)\ldots)$	$\bot \to (\bot \to \ldots \to (\bot \to \bot)\ldots)$

We argue that $\text{Tree}^{-1}(\text{Tree}(\tau)|_k) \leq \text{Tree}^{-1}(\text{Tree}(\sigma)|_k)$ for all k. Approximations $k = 0, 1$ are trivial by reflexivity. For $k = 2$ we apply the contravariant function subtype rule and for $n > 2$ we do induction on n. We thus conclude that $\tau \leq_\mu \sigma$. ◇

3.2.1 Characterization

Kozen, Palsberg and Schwartzbach [KPS93, KPS95] show that $\leq_\mu$ on trees can be characterized by the nonexistence of a path that, in a sense, would witness that two trees cannot possibly be in a partial order relation that respects the basis partial order on labels.

We introduce here yet another characterization of $\leq_\mu$, one that is substantially simpler than all the other characterizations and which is intrinsically in terms of recursive types, without referring to infinite trees. The concepts of simulation and bisimulation originates in concurrency theory [Par81, Mil89] where it is used as a reasoning tool for observational equivalence. Fiore [Fio94] applied the technique to recursive structures and introduced the coinduction principle we use extensively here.

Definition 3.2.3 A *simulation (on recursive types)* is a binary relation $\mathcal{R}$ on $\mu\mathrm{Tp}$ satisfying:

1. $\tau_1 \to \tau_2 \,\mathcal{R}\, \sigma_1 \to \sigma_2 \;\Rightarrow\; \sigma_1 \,\mathcal{R}\, \tau_1$ and $\tau_2 \,\mathcal{R}\, \sigma_2$
2. $\mu\alpha.\tau \,\mathcal{R}\, \sigma \;\Rightarrow\; \tau[\mu\alpha.\tau/\alpha] \,\mathcal{R}\, \sigma$
3. $\tau \,\mathcal{R}\, \mu\beta.\sigma \;\Rightarrow\; \tau \,\mathcal{R}\, \sigma[\mu\beta.\sigma/\beta]$
4. $\tau \,\mathcal{R}\, \sigma \;\Rightarrow\; \mathcal{L}(\tau) \leq \mathcal{L}(\sigma)$

where the label of a recursive type $\mathcal{L}(\tau)$ is the top level constructor. It may be defined as $\mathcal{L}(\tau) = \mathrm{Tree}(\tau)(\epsilon)$. The order on labels are the transitive, reflexive closure of

$$\bot \leq \left\{ \begin{array}{c} \to \\ \alpha \end{array} \right\} \leq \top$$

◇

Note that the label function $\mathcal{L}$ may be defined without reference to regular trees. A relation $\mathcal{R}$ is called a bisimulation if both $\mathcal{R}$ and $\tilde{\mathcal{R}} = \{(\sigma,\tau) \mid (\tau,\sigma) \in \mathcal{R}\}$ are simulations. It is obvious that a bisimulation is a relation satisfying rules 1,2 and 4 of the simulation definition plus an extra symmetry rule.

With simulations we can give an elegant characterization of the subtype relation.

Lemma 3.2.4 *(Characterization of Recursive Subtyping)*
$\tau \leq_\mu \sigma$ if and only if there exists a simulation $\mathcal{R}$ such that $\tau\mathcal{R}\sigma$.

PROOF The direction from left to right is proven by showing that $\leq_\mu$ itself is a simulation. We show that the four simulation conditions hold.

Ad.1 Assume $\tau_1 \to \tau_2 \leq_\mu \sigma_1 \to \sigma_2$ and show $\sigma_1 \leq_\mu \tau_1$, $\tau_2 \leq_\mu \sigma_2$. To prove $\sigma_1 \leq_\mu \tau_1$ we need to show for all k that $\mathrm{Tree}(\sigma_1)|_k \leq_{\mathrm{fin}} \mathrm{Tree}(\tau_1)$. Let k be given. We know that $\mathrm{Tree}(\tau_1 \to \tau_2)|_{k+1} \leq_{\mathrm{fin}} \mathrm{Tree}(\sigma_1 \to \sigma_2)|_{k+1}$ and furthermore

$$\begin{array}{rcl} \mathrm{Tree}(\tau_1 \to \tau_2)|_{k+1} & = & \mathrm{Tree}(\tau_1)|_k \to \mathrm{Tree}(\tau_2)|_k \\ \mathrm{Tree}^{-1}(\mathrm{Tree}(\tau_1 \to \tau_2)|_{k+1}) & = & \mathrm{Tree}^{-1}(\mathrm{Tree}(\tau_1)|_k) \to \mathrm{Tree}^{-1}(\mathrm{Tree}(\tau_2)|_k) \end{array}$$

and similar for $\sigma_1 \to \sigma_2$. By definition of $\leq$ we find that $\mathrm{Tree}^{-1}(\mathrm{Tree}(\sigma_1)|_k) \leq \mathrm{Tree}^{-1}(\mathrm{Tree}(\tau_1)|_k)$ and thereby $\mathrm{Tree}(\sigma_1)|_k \leq_{\mathrm{fin}} \mathrm{Tree}(\tau_1)|_k$. We have hereby shown that $\sigma_1 \leq_\mu \tau_1$. A analogous argument holds for $\tau_2 \leq_\mu \sigma_2$.

Ad.2 Assume next that $\mu\alpha.\tau \leq_\mu \sigma$. Show $\tau[\mu\alpha.\tau/\alpha] \leq_\mu \sigma$, but since $\mathrm{Tree}(\mu\alpha.\tau) = \mathrm{Tree}(\tau[\mu\alpha.\tau/\alpha])$ by Proposition 2.2.1 we can trivially infer the desired result due to reflexivity of $\leq$.

Ad.3 Exactly as condition 2.

Ad.4 Let $\tau \leq_\mu \sigma$ and show $\mathcal{L}(\tau) \leq \mathcal{L}(\sigma)$. Observe first by inspection that $\tau' \leq \sigma'$ implies $\mathcal{L}(\tau') \leq \mathcal{L}(\sigma')$ for non-recursive types τ', σ'. Now, for $k \geq 1$ we have

$$\mathcal{L}(\tau) = \mathcal{L}(\mathrm{Tree}^{-1}(\mathrm{Tree}(\tau)|_k))$$

and hence is the condition satisfied because

$$\mathrm{Tree}^{-1}(\mathrm{Tree}(\tau)|_k) \leq \mathrm{Tree}^{-1}(\mathrm{Tree}(\sigma)|_k)$$

for all k.

The opposite direction is proven as follows. Let $\mathcal{R}$ be a simulation. We prove by induction on k that

$$\tau\mathcal{R}\sigma \;\Rightarrow\; \mathrm{Tree}(\tau)|_k \leq_{\mathrm{fin}} \mathrm{Tree}(\sigma)|_k$$

Case $k = 0$**:** Trivial, since $\bot \leq \bot$.

Case $k > 0$**:** We now have for $\tau'\mathcal{R}\sigma'$ that $\mathrm{Tree}(\tau')|_{(k-1)} \leq_{\mathrm{fin}} \mathrm{Tree}(\sigma')|_{(k-1)}$. Assume $\tau\mathcal{R}\sigma$ holds. We then have $\mathcal{L}(\tau) \leq \mathcal{L}(\sigma)$ by condition 4 of the simulation definition. We perform a case analysis of τ, σ under the restriction that their labels are related (this restriction reduces the number of cases dramatically!),

Case $\tau = \bot$**:** Trivial.

Case $\sigma = \top$**:** Trivial.

Case $\tau = \alpha = \sigma$**:** Trivial.

Case $\tau = \tau_1 \to \tau_2,\ \sigma = \sigma_1 \to \sigma_2$**:** Since $\mathcal{R}$ is a simulation we get from condition 1 that $\sigma_1\mathcal{R}\tau_1$ and thus by induction hypothesis

$$\mathrm{Tree}(\sigma_1)|_{(k-1)} \leq_{\mathrm{fin}} \mathrm{Tree}(\tau_1)|_{(k-1)}$$

and likewise

$$\mathrm{Tree}(\tau_2)|_{(k-1)} \leq_{\mathrm{fin}} \mathrm{Tree}(\sigma_2)|_{(k-1)}$$

We conclude

$$\begin{array}{lcl} \mathrm{Tree}(\tau_1 \to \tau_2)|_k & = & \\ \mathrm{Tree}(\tau_1)|_{(k-1)} \to \mathrm{Tree}(\tau_2)|_{(k-1)} & \leq_{\mathrm{fin}} & \\ \mathrm{Tree}(\sigma_1)|_{(k-1)} \to \mathrm{Tree}(\sigma_2)|_{(k-1)} & = & \mathrm{Tree}(\sigma_1 \to \sigma_2)|_k \end{array}$$

Case $\tau = \mu\alpha.\tau_1 \to \tau_2,\ \sigma = \sigma_1 \to \sigma_2$**:** Again, since $\mathcal{R}$ is a simulation and $\tau\mathcal{R}\sigma$ we get that $(\tau_1 \to \tau_2)[\tau/\alpha]\mathcal{R}\sigma$. Let $\tau_1' \to \tau_2' = (\tau_1 \to \tau_2)[\tau/\alpha]$. The rest of the proof follows the same schema as previous case.

Case $\ldots$**:** The remaining cases are similar to the above case.

We have just proven

$$\forall k.\ \tau \mathcal{R} \sigma \ \Rightarrow\ \mathrm{Tree}(\tau)|_k \leq_{\mathrm{fin}} \mathrm{Tree}(\sigma)|_k$$

and hence

$$\tau \mathcal{R} \sigma \ \Rightarrow\ \forall k.\ \mathrm{Tree}(\tau)|_k \leq_{\mathrm{fin}} \mathrm{Tree}(\sigma)|_k$$

which implies $\tau \leq_\mu \sigma$. ◇

Recursive type equality can, not surprisingly, be characterized by bisimulations.

Lemma 3.2.5 *(Characterization of Recursive Equality)*
$\tau =_\mu \sigma$ if and only if there exists a bisimulation $\mathcal{R}$ such that $\tau \mathcal{R} \sigma$.

Chapter 4

Amadio Cardelli Axiomatizations

4.1 Equality Relation

Cardone and Coppo [CC91] defined recursive equality, but gave no axiomatization of it. Amadio, Cardelli [AC93] and Ariola, Klop [AK95] have later given axiomatizations of $=_\mu$. Axiomatizations similar to theirs have though been known for long, *e.g.* [Sal66, Mil84]. In this section we will present the axiomatization of Amadio, Cardelli.

Weak equality was mentioned earlier but only informally. In Figure 4.1 we give a formal presentation of the weak equality.

$$\tau =_{\mathrm{w}} \tau \quad (\textsc{Ref}) \qquad \frac{\tau_1 =_{\mathrm{w}} \tau_2 \quad \tau_2 =_{\mathrm{w}} \tau_3}{\tau_1 =_{\mathrm{w}} \tau_3} \quad (\textsc{Trans}) \qquad \frac{\tau_2 =_{\mathrm{w}} \tau_1}{\tau_1 =_{\mathrm{w}} \tau_2} \quad (\textsc{Sym})$$

$$\mu\alpha.\tau_1 =_{\mathrm{w}} [\mu\alpha.\tau_1/\alpha]\tau_1 \quad (\textsc{Unfold}) \qquad \frac{\tau_1 =_{\mathrm{w}} \tau_2 \quad \tau_1' =_{\mathrm{w}} \tau_2'}{\tau_1 \to \tau_1' =_{\mathrm{w}} \tau_2 \to \tau_2'} \quad (\textsc{Arrow})$$

Figure 4.1: Axiomatization of weak Equality

Let us review the example from the introduction (Section 1.3), but now with weak recursive equality.

$$\begin{aligned}
\tau &=_{\mathrm{w}} \mu\alpha.\bot \to \alpha \\
&=_{\mathrm{w}} \bot \to (\mu\alpha.\bot \to \alpha) \\
&=_{\mathrm{w}} \bot \to (\bot \to (\mu\alpha.\bot \to \alpha)) \\
&=_{\mathrm{w}} \bot \to (\bot \to \tau)
\end{aligned}$$

$$\begin{aligned}
\sigma &=_{\mathrm{w}} \mu\alpha.\bot \to (\bot \to \alpha) \\
&=_{\mathrm{w}} \bot \to (\bot \to (\mu\alpha.\bot \to (\bot \to \alpha))) \\
&=_{\mathrm{w}} \bot \to (\bot \to \sigma)
\end{aligned}$$

$$A \vdash \bot \leq_{\mathrm{AC}} \tau \qquad A \vdash \tau \leq_{\mathrm{AC}} \top$$

$$A \vdash \tau \leq_{\mathrm{AC}} \tau \qquad \frac{A \vdash \tau \leq_{\mathrm{AC}} \tau' \quad A \vdash \tau' \leq_{\mathrm{AC}} \tau''}{A \vdash \tau \leq_{\mathrm{AC}} \tau''}$$

$$A(\tau \leq_{\mathrm{AC}} \tau')A' \vdash \tau \leq_{\mathrm{AC}} \tau' \qquad \frac{\vdash \tau =_{\mathrm{AC}} \tau'}{A \vdash \tau \leq_{\mathrm{AC}} \tau'}$$

$$\frac{A \vdash \tau' \leq_{\mathrm{AC}} \tau \quad A \vdash \sigma \leq_{\mathrm{AC}} \sigma'}{A \vdash \tau \to \sigma \leq_{\mathrm{AC}} \tau' \to \sigma'} \qquad \frac{A(\alpha \leq_{\mathrm{AC}} \beta) \vdash \tau \leq_{\mathrm{AC}} \sigma}{A \vdash \mu\alpha.\tau \leq_{\mathrm{AC}} \mu\beta.\sigma} \quad (\alpha \notin \mathrm{fv}(\sigma),\ \beta \notin \mathrm{fv}(\tau))$$

Figure 4.2: Amadio, Cardelli Axiomatization of Recursive Subtyping

We find that after some unfolding and applications of the arrow rule we arrive at the initial judgement again, $\tau =_{\mathrm{w}} \sigma$. Still, they do have some common structure. Consider the context $C(X) = \bot \to (\bot \to X)$,

$$C(\tau) =_{\mathrm{w}} \tau \qquad C(\sigma) =_{\mathrm{w}} \sigma$$

that is, τ and σ are fix-points of C. Translate τ, σ to their tree representations and C to a function on trees. Because the space of trees is a complete metric space (Proposition 2.1.3) we know by Banach that Tree(C) has a *unique* fixed point, but since both Tree(τ) and Tree(σ) are fix-points of Tree(C) they must be identical, *i.e.* $\tau =_\mu \sigma$. We have thus proved τ, σ equal by reasoning with $=_{\mathrm{w}}$ and the unique fix point property of T^Σ. Amadio, Cardelli formulated the CONTRACT rule

$$\frac{C(\tau) =_{\mathrm{AC}} \tau \quad C(\sigma) =_{\mathrm{AC}} \sigma \quad \text{Tree(C) contractive}}{\tau =_{\mathrm{AC}} \sigma} \quad (\textsc{Contract})$$

Contractiveness of Tree(C) can easily be formulated as a syntactic restriction on C – it says informally that μ-bound variables must occur *beneath* arrow constructors. It is not a big surprise that this restriction is what motivates our canonical recursive types μTp. Amadio, Cardelli show that $(=_{\mathrm{AC}}) \equiv (=_{\mathrm{w}}) + (\textsc{Contract})$ is a complete and sound axiomatization of recursive equality.

Theorem 4.1.1 *$\tau =_\mu \sigma$ if and only if $\tau =_{AC} \sigma$.*

4.2 Subtype Relation

Amadio, Cardelli show in [AC93] that the key problem concerning recursive subtyping is due to recursive equality (see also Section 1.3). By adding a compatibility rule for μ types and a rule exploiting recursive equality to the standard contravariant subtype axiomatization (see Figure 3.1) they obtain a complete and sound axiomatization of $\leq_\mu$ (see Figure 4.2).

Theorem 4.2.1 *$\tau \leq_\mu \sigma$ if and only if $\tau \leq_{AC} \sigma$.*

Example 4.2.2 Let $\tau \equiv \mu\alpha.\top \to \alpha$ and $\sigma \equiv \mu\beta.\bot \to (\bot \to \beta)$ as in Section 1.3. We show that $\tau \leq_{\text{AC}} \sigma$. We showed in Section 4.1 that $\mu\beta.\bot \to (\bot \to \beta) =_{\text{AC}} \mu\beta.\bot \to \beta$ and hence we only need to show $\mu\alpha.\top \to \alpha \leq_{\text{AC}} \mu\beta.\bot \to \beta$. For this we assume $\alpha \leq_{\text{AC}} \beta$ and attempt to prove $\top \to \alpha \leq_{\text{AC}} \bot \to \beta$, but since $\bot \leq_{\text{AC}} \top$ ($\bot$ axiom) and $\alpha \leq_{\text{AC}} \beta$ (assumption) we conclude the judgment by the arrow rule. Altogether we have proven $\tau \leq_{\text{AC}} \sigma$. $\diamond$

Chapter 5

New Axiomatizations

This chapter presents one of the major contributions of this work, namely the new axiomatizations of recursive type equality and subtyping. These new axiomatizations are based on coinductive principles which seem natural considering the bisimulation and simulation characterizations presented in Section 3.2. The interesting difference between our axiomatizations and those of Amadio, Cardelli and others is that we do not have a contract rule nor a compatibility rule for μ. The contract rule is problematic in at least two ways:

1. It requires a contractive context for which the recursive types are fixpoints. This context has to come right out of the blue. There is no direct relation between syntax of the types and this mysterious context. Theoretically this is not a problem, but in practice one would like an algorithm for determining when two recursive types are in the relation. Such algorithms become quite complicated and involved (see [AC93, KPS93, KPS95]).

2. It is very hard to give an operational interpretation of it, *i.e.* it is hard to construct a reasonable coercion encoding the rule.

First, we present and motivate the equality axiomatization, because it has the same fundamental coinductive features as the subtype relation, but in a more pure and uncluttered form. It is therefore easier to illustrate our intuition and ideas of how to axiomatize the relation. Next, we show the axiomatization of the subtype relation which is actually a very natural extension of the equality axiomatization. Completeness and soundness theorems are proved for the subtype axiomatization, but only stated for equality (for details we refer to [Bra97]).

5.1 Equality Relation

Recursive types are usually represented as infinite regular trees (*e.g.* definition of equality, Definition 3.1.1), but they can also be represented as directed cyclic graphs.

Example 5.1.1 Consider the recursive types $\tau \equiv \mu\alpha.\bot \to \alpha$ and $\sigma \equiv \mu\alpha.\bot \to (\bot \to \alpha)$. Their representation as graphs are shown in Figure 5.1. $\diamond$

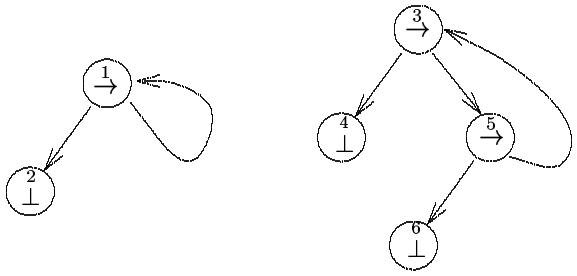


Figure 5.1: Graph representations of τ and σ

Structural equivalence of graphs by unification is a well-known algorithm (see [ASU86]) going back to Hopcroft and Ullman (1971) for FSM equivalence and Huet (1976) for unification. The unification algorithm $\mathbf{unify}(A, m, n)$ takes a set of assumptions A and two graph nodes m, n. Intuitively $\mathbf{unify}(A, m, n)$ does the following:

> If m, n are assumed equal (*i.e.* $(m = n) \in A$) or they are structural identical leafs then return true. If m, n are internal nodes with the same label and rank (arity) then assume them equal (*i.e.* add $(m = n)$ to A) and unify each pair of children. If all children unify return true. Otherwise return false.

If $\mathbf{unify}(A, m, n) = \mathit{true}$ then the graphs rooted at m and n are structurally equivalent under the assumptions in A.

Example 5.1.2 Consider τ and σ of Example 5.1.1 and their graph representation in Figure 5.1. Unification of the root nodes $\mathbf{unify}(\epsilon, 1, 3)$ leads to unification of the children because nodes 1, 3 have the same label and rank. The sub unifications are $\mathbf{unify}(A, 2, 4)$ and $\mathbf{unify}(A, 1, 5)$ with $A \equiv (1 = 3)$. The first unification returns *true* immediately, because nodes 2 and 4 are identical leaf nodes. The second unification produces yet another two sub unifications; $\mathbf{unify}(A', 2, 6)$ and $\mathbf{unify}(A', 1, 3)$ where $A' \equiv (1 = 3, 1 = 5)$. Once again the first unification returns *true* (same reason as before), but now the second unification also yields *true* because nodes 1 and 3 are assumed equal. The entire unification succeeds. ◇

The interesting thing about **unify** is its coinductive nature. It says, "Assume m and n unify and prove that all their children unifies – then m and n unify". You assume what you are trying to prove, or in other words, if you can not find a witness of contradiction then your assumption is true. These are all common characteristics of coinduction. Based on this idea of unification and coinduction we formulate the fundamental rule of our axiomatization,

$$\frac{A(\tau = \sigma) \vdash \tau = \sigma}{A \vdash \tau = \sigma} \quad \text{(Fix)}$$

This rule captures the coinductive principle in that $\tau = \sigma$ is added to the assumption set A. It has, though, an obvious defect,

$$\dfrac{\dfrac{A(\tau = \sigma) \vdash \tau = \sigma}{A(\tau = \sigma) \vdash \tau = \sigma}\ (\text{Hyp})}{A \vdash \tau = \sigma}\ (\text{Fix})$$

We can thus prove $\tau = \sigma$ for all τ, σ! This is not quite what we had in mind. Some restrictions on the derivation in the premise of (Fix) is therefore needed. This situation is analogous to the contractive restriction on type contexts in the (Contract) rule of Amadio, Cardelli. Consider the non-contractive context $\lambda t.t$. This context has the exact same effect as the derivation described above, namely that every judgement can be proven true. With this observation in mind we realize that our restriction on the premise derivation of (Fix) has to be the same as that of (Contract). Loosely speaking, an assumption may only be *fired* if it is "protected" by an application of the arrow rule. Instead of stating this restriction as a side condition on the (Fix) rule we decide to combine (Arrow) and (Fix) such that the side condition can be avoided. What we arrive at is the (Arrow/Fix) rule,

$$\dfrac{\begin{array}{l}A(\tau_1 \to \tau_2 = \sigma_1 \to \sigma_2) \vdash \tau_1 = \sigma_1 \\ A(\tau_1 \to \tau_2 = \sigma_1 \to \sigma_2) \vdash \tau_2 = \sigma_2\end{array}}{A \vdash \tau_1 \to \tau_2 = \sigma_1 \to \sigma_2}\ (\text{Arrow/Fix} =)$$

Note that this corresponds to applying rule (Fix) only after the arrow rule. The entire axiomatization of recursive type equality is presented in Figure 5.2

$$A \vdash \mu\alpha.\tau = [\mu\alpha.\tau/\alpha]\tau \qquad (\text{Unfold} =)$$

$$A(\tau = \sigma)A' \vdash \tau = \sigma \qquad (\text{Hyp} =)$$

$$\dfrac{A' \vdash \tau_1 = \sigma_1 \quad A' \vdash \tau_2 = \sigma_2}{A \vdash \tau_1 \to \tau_2 = \sigma_1 \to \sigma_2} \qquad \begin{array}{c}(\text{Arrow/Fix} =) \\ A' = A(\tau_1 \to \tau_2 = \sigma_1 \to \sigma_2)\end{array}$$

$$A \vdash \tau = \tau \qquad (\text{Ref} =)$$

$$\dfrac{A \vdash \tau_1 = \tau_2 \quad A \vdash \tau_2 = \tau_3}{A \vdash \tau_1 = \tau_3} \qquad (\text{Trans} =)$$

$$\dfrac{A \vdash \tau_2 = \tau_1}{A \vdash \tau_1 = \tau_2} \qquad (\text{Sym} =)$$

Figure 5.2: Coinductive axiomatization of recursive type equality

Example 5.1.3 Consider once again the recursive types τ, σ of Section 1.3. We show that $\epsilon \vdash \tau = \sigma$. First, unfold τ and σ once and obtain the judgement,

$$\epsilon \vdash \bot \to \tau = \bot \to (\bot \to \sigma)$$

By rule (Arrow/Fix =) we need to prove

$$A \vdash \bot = \bot \qquad A \vdash \tau = \bot \to \sigma$$

where $A = (\bot \to \tau = \bot \to (\bot \to \sigma))$. The leftmost judgement is proven directly by rule (Ref =). The other judgement requires yet another unfolding of τ,

$$A \vdash \bot \to \tau = \bot \to \sigma$$

Another application of (Arrow/Fix =) introduces (besides the trivial $\bot = \bot$) the judgement $A' \vdash \tau = \sigma$ with $A' = A(\bot \to \tau = \bot \to \sigma)$. By repeating the initial unfoldings we arrive at

$$A' \vdash \bot \to \tau = \bot \to (\bot \to \sigma)$$

which we can prove directly with (Hyp =) because A' contains this very judgement as an assumption. ◇

The equality axiomatization is complete and sound with respect to the definition of equality.

Theorem 5.1.4 *(Brandt'97)*

$$\tau =_\mu \sigma \qquad \Leftrightarrow \qquad \epsilon \vdash \tau = \sigma$$

5.2 Subtype Relation

We now turn our attention to the recursive subtype relation. The basic idea is to axiomatize recursive subtyping with the same coinductive principle as with equality. Most of the material presented in this section has appeared in proceedings to the 1997 TLCA conference ([BH97]) in Nancy, France.

The coinductive nature of recursive equality is captured by the rules (Arrow/Fix =) and (Hyp =) in Figure 5.2. For recursive subtyping we simple direct the (Arrow/Fix =) rule in the ordinary contravariant fashion,

$$\frac{A' = A(\tau_1 \to \tau_2 \le \sigma_1 \to \sigma_2) \qquad A' \vdash \sigma_1 \le \tau_1 \quad A' \vdash \tau_2 \le \sigma_2}{A \vdash \tau_1 \to \tau_2 \le \sigma_1 \to \sigma_2} \qquad \text{(Arrow/Fix } \le\text{)}$$

Next, consider the (Unfold =) rule. Since this is an axiom we need to include two subtype axioms – one in each direction, such that the equality is preserved in the subtype relation (*i.e.* both $\le$ and $\ge$ holds). The reflexivity axiom (Ref =) does of course only result in a single axiom in the subtype axiomatization, because the types are identical. Besides the axioms of = (without symmetry) we need to express how $\bot$ and $\top$ relate to other types, which is straightforward. The entire axiomatization is shown in Figure 5.3.

Example 5.2.1 We can show $\epsilon \vdash \tau' \le \sigma'$ with τ', σ' from Section 1.3. Since the derivation is practically identical to the one from Example 5.1.3 we will not go into details. ◇

5.2.1 Soundness

This section focuses on the soundness of $\le$ with respect to $\le_\mu$, which means that $\vdash \tau \le \sigma$ implies $\tau \le_\mu \sigma$. Soundness is hard to prove and the standard strategy of proving $(\tau_1 \le \sigma_1, \ldots, \tau_n \le \sigma_n) \vdash \tau \le \sigma$ by assuming $\tau_1 \le_\mu \sigma_1, \ldots, \tau_n \le_\mu \sigma_n$

$$A \vdash \bot \leq \tau \quad (\bot \leq) \qquad A \vdash \tau \leq \top \quad (\top \leq)$$

$$A \vdash \tau \leq \tau \quad (\text{Ref} \leq) \qquad \frac{A \vdash \tau \leq \delta \quad A \vdash \delta \leq \sigma}{A \vdash \tau \leq \sigma} \quad (\text{Trans} \leq)$$

$$A \vdash \mu\alpha.\tau \leq \tau[\mu\alpha.\tau/\alpha] \quad (\text{Unfold} \leq) \qquad A \vdash \tau[\mu\alpha.\tau/\alpha] \leq \mu\alpha.\tau \quad (\text{Fold} \leq)$$

$$A(\tau \leq \sigma)A' \vdash \tau \leq \sigma \quad (\text{Hyp} \leq) \qquad \frac{\begin{array}{c}A' = A(\tau_1 \to \tau_2 \leq \sigma_1 \to \sigma_2) \\ A' \vdash \sigma_1 \leq \tau_1 \quad A' \vdash \tau_2 \leq \sigma_2\end{array}}{A \vdash \tau_1 \to \tau_2 \leq \sigma_1 \to \sigma_2} \quad (\text{Arrow/Fix} \leq)$$

Figure 5.3: Coinductive axiomatization of recursive subtyping

and proving $\tau \leq_\mu \sigma$ does not work. The reason is the (Arrow/Fix) rule and specifically the Fix part of it. It says that to prove $A \vdash \tau \leq \sigma$ simply prove $A(\tau \leq \sigma) \vdash \tau \leq \sigma$, but the problem is that $\tau \leq_\mu \sigma$ might not hold and then to add to the assumptions would be unsound. This does not mean that our system is unsound, but only that the inductive step used in the standard proof strategy is too strong. We present instead a proof based on the idea that derivations in our axiomatization correspond to simulations. Another proof using stratified interpretations is given in the TLCA article [BH97].

Simulation

We first devise a new axiom system, $\leq_{\mathrm{A}}$, based on a fixed assumption set A.

Definition 5.2.2 For an assumption set A the fixed subtype relation $\leq_{\mathrm{A}}$ is defined by

$$\bot \leq_{\mathrm{A}} \tau \qquad (\textsc{Fixed-}\bot)$$

$$\tau \leq_{\mathrm{A}} \top \qquad (\textsc{Fixed-}\top)$$

$$\tau \leq_{\mathrm{A}} \tau \qquad (\textsc{Fixed-Ref})$$

$$\frac{(\tau, \sigma) \in A}{\tau \leq_{\mathrm{A}} \sigma} \qquad (\textsc{Fixed-Hyp})$$

$$\mu\alpha.\tau \leq_{\mathrm{A}} \tau[\mu\alpha.\tau/\alpha] \qquad (\textsc{Fixed-Unfold})$$

$$\tau[\mu\alpha.\tau/\alpha] \leq_{\mathrm{A}} \mu\alpha.\tau \qquad (\textsc{Fixed-Fold})$$

$$\frac{\sigma_1 \leq_{\mathrm{A}} \tau_1,\ \tau_2 \leq_{\mathrm{A}} \sigma_2}{\tau_1 \to \tau_2 \leq_{\mathrm{A}} \sigma_1 \to \sigma_2} \qquad (\textsc{Fixed-Arrow})$$

$$\frac{\tau \leq_{\mathrm{A}} \delta \quad \delta \leq_{\mathrm{A}} \sigma}{\tau \leq_{\mathrm{A}} \sigma} \qquad (\textsc{Fixed-Trans})$$

$\diamond$

Assumption sets in this system is not constrained to arrows only.

Before defining our simulation candidate we have to create an appropriate assumption set to be used with $\leq_A$.

Definition 5.2.3 The assumption closure of a pair of canonical recursive types is defined by

$$\begin{aligned}(\tau_1 \to \tau_2, \sigma_1 \to \sigma_2)^* &= \{(\tau_1 \to \tau_2, \sigma_1 \to \sigma_2)\} \cup (\sigma_1, \tau_1)^* \cup (\tau_2, \sigma_2)^* \\ (\mu\alpha.\tau, \sigma)^* &= \{(\mu\alpha.\tau, \sigma)\} \cup (\tau[\mu\alpha.\tau/\alpha], \sigma)^* \\ (\sigma, \mu\alpha.\tau)^* &= \{(\mu\alpha.\tau, \sigma)\} \cup (\sigma, \tau[\mu\alpha.\tau/\alpha])^* \\ (\tau, \sigma)^* &= \{(\tau, \sigma)\}\end{aligned}$$

◇

Observe by inspection that

1. $(\tau, \sigma) \in (\tau, \sigma)^*$
2. Equation (4) never includes pairs where both recursive types have arrow labels.

Assumption closures satisfy the fundamental arrow property required of a simulation.

Lemma 5.2.4 *Let p be a pair of canonical recursive types. If $(\tau, \sigma) \in p^*$ and $\mathcal{L}(\tau) = (\to, \tau_1, \tau_2)$, $\mathcal{L}(\sigma) = (\to, \sigma_1, \sigma_2)$ then*

$$(\sigma_1, \tau_2) \in p^* \quad \textit{and} \quad (\tau_2, \sigma_2) \in p^*$$

PROOF Case analysis of (τ, σ).

Case $(\tau_1 \to \tau_2, \sigma_1 \to \sigma_2)$**:** From observation 2 above we can conclude that it is the first equation which has actually included the pair in p^*. A consequence of this is that

$$(\sigma_1, \tau_1)^* \subseteq p^*$$

and thus $(\sigma_1, \tau_1) \in p^*$ by observation 1. Analogously is $(\tau_2, \sigma_2) \in p^*$.

Case $(\mu\alpha.\tau, \sigma_1 \to \sigma_2)$**:** If this pair sits in p^* it means that equation 2 added it to p^*. Again this means that

$$(\tau[\mu\alpha.\tau/\alpha], \sigma_1 \to \sigma_2)^* \subseteq p^*$$

Since the types are canonical, we get that $\tau[\mu\alpha.\tau/\alpha] = \tau_1 \to \tau_2$. We thus have

$$(\tau_1 \to \tau_2, \sigma_1 \to \sigma_2) \in p^*$$

and as we saw in the first case, this implies that

$$(\sigma_1, \tau_1) \in p^*$$

and similar for (τ_2, σ_2).

Case $(\tau_1 \to \tau_2, \mu\beta.\sigma)$**:** Similar to above case.

Case $(\mu\alpha.\tau, \mu\beta.\sigma)$**:** Follows exact same schema as previous cases. ◇

Lemma 5.2.5 *Let p be a pair as above and $A = p^*$. If $\tau \leq_A \sigma$ and $\mathcal{L}(\tau) = (\rightarrow, \tau_1, \tau_2)$, $\mathcal{L}(\sigma) = (\rightarrow, \sigma_1, \sigma_2)$ then*

$$\sigma_1 \leq_A \tau_1 \quad \textit{and} \quad \tau_2 \leq_A \sigma_2$$

PROOF The only interesting case is FIXED-HYP.

Case FIXED-HYP**:** We have that $(\tau \leq \sigma) \in p^*$. Lemma 5.2.4 then gives that

$$(\sigma_1, \tau_2) \in p^* \quad \text{and} \quad (\tau_2, \sigma_2) \in p^*$$

and by FIXED-HYP we get the desired result. ◇

Lemma 5.2.6 *For any pair of canonical recursive types (τ', σ') where $\tau' \leq \sigma'$ it holds for $A = (\tau', \sigma')^*$ that*

$$\tau \leq_A \sigma \;\Rightarrow\; \mathcal{L}(\tau) \leq \mathcal{L}(\sigma)$$

PROOF Induction on derivation. Only interesting case is FIXED-HYP, which implies showing that the assumption closure is label consistent. This is easily seen by inspection since the pair (τ, σ) is label consistent by completeness of $\leq$. ◇

The soundness theorem is finally within reach.

Theorem 5.2.7 *(Soundness) For canonical recursive types τ, σ*

$$\tau \leq \sigma \Rightarrow \tau \leq_\mu \sigma$$

PROOF Let $A = (\tau, \sigma)^*$ and $\mathcal{R} = \leq_A$. If $\mathcal{R}$ is a simulation containing the pair (τ, σ) then by 3.2.4 we get $\tau \leq_\mu \sigma$.

We first show that $\mathcal{R}$ is a simulation.

1. Assume $\tau_1 \rightarrow \tau_2 \leq_A \sigma_1 \rightarrow \sigma_2$. By 5.2.5 we thus get $\sigma_1 \leq_A \tau_1$ and $\tau_2 \leq_A \sigma_2$ which proves the property.

2. Assume $\mu\alpha.\tau' \leq_A \sigma'$. By (FIXED-FOLD) and (FIXED-TRANS) we get

$$\frac{\tau'[\mu\alpha.\tau'/\alpha] \leq_A \mu\alpha.\tau' \quad \mu\alpha.\tau' \leq_A \sigma'}{\tau'[\mu\alpha.\tau'/\alpha] \leq_A \sigma'}$$

3. Assume $\tau' \leq_A \mu\beta.\sigma'$. Result follows from (FIXED-UNFOLD) and (FIXED-TRANS) since

$$\frac{\tau \leq_A \mu\beta.\sigma' \quad \mu\beta.\sigma' \leq_A \sigma'[\mu\beta.\sigma'/\beta]}{\tau' \leq_A \sigma'[\mu\beta.\sigma'/\beta]}$$

4. Assume $\tau' \leq_A \sigma'$. By 5.2.6 we immediately get the desired result, $\mathcal{L}(\tau') \leq \mathcal{L}(\sigma')$.

We thus have that $\mathcal{R}$ is a simulation. It is easily shown from FIXED-HYP that $\tau \,\mathcal{R}\, \sigma$ since $(\tau, \sigma) \in (\tau, \sigma)^* = A$. ◇

5.2.2 Completeness

This section is concerned with the completeness of our axiomatization with respect to $\leq_\mu$, that is, whenever $\tau \leq_\mu \sigma$ then the same conclusion is derivable in our axiomatization: $\vdash \tau \leq \sigma$. The proof is divided into three parts; 1) an algorithm **S** that produces derivations in $\leq$, 2) a termination proof of **S** and finally 3) a correctness proof of **S** based on the simulation characterization of $\leq_\mu$.

Algorithm S

Consider Algorithm **S** in Figure 5.2.2. The most interesting part of the algorithm concerns function types. A pair of functions may have been encountered earlier in the computation and are therefore stored in the assumption set. If that is the case, rule HYP is applied. Otherwise the computation is continued (by rule ARROW/FIX) until the assumption set contains sufficiently many function pairs. Termination of the algorithm garantees that the assumption sets are always finite. One might therefore say that the algorithm builds a minimal simulation containing the pair (τ, σ) and thereby proving them related by the recursive subtype relation.

1: $\mathbf{S}(A, \mu\alpha.\tau, \sigma) =$
2: **let**
3: $\mathcal{D}_1 =$ UNFOLD
4: $\mathcal{D}_2 = \mathbf{S}(A, \tau[\mu\alpha.\tau/\alpha], \sigma)$
5: **in**
6: TRANS$(\mathcal{D}_1, \mathcal{D}_2)$
7: **end**
8: $\mathbf{S}(A, \tau, \mu\beta.\sigma) =$
9: **let**
10: $\mathcal{D}_1 = \mathbf{S}(A, \tau, \sigma[\mu\beta.\sigma/\beta])$
11: $\mathcal{D}_2 =$ FOLD
12: **in**
13: TRANS$(\mathcal{D}_1, \mathcal{D}_2)$
14: **end**
15: $\mathbf{S}(A(\tau \leq \sigma), \tau, \sigma) =$ HYP
16: $\mathbf{S}(A, \tau_1 \to \tau_2, \sigma_1 \to \sigma_2) =$
17: **let**
18: $A' = A(\tau_1 \to \tau_2 \leq \sigma_1 \to \sigma_2)$
19: $\mathcal{D}_1 = \mathbf{S}(A', \sigma_1, \tau_1)$
20: $\mathcal{D}_2 = \mathbf{S}(A', \tau_2, \sigma_2)$
21: **in**
22: ARROW$(\mathcal{D}_1, \mathcal{D}_2)$
23: **end**
24: $\mathbf{S}(A, \alpha, \alpha) =$ REF
25: $\mathbf{S}(A, \bot, \tau) = \bot$
26: $\mathbf{S}(A, \tau, \top) = \top$
27: $\mathbf{S}(A, \tau, \sigma) =$ **exception**

Figure 5.4: Algorithm **S**

Termination of S

In this section we prove that **S** terminates.

Syntactic subterms We first introduce the concept of syntactic subterms and prove some properties about them. A series of technical lemmas finally

establishes the main property of the notion of subterms: every recursive type has only a finite number of subterms.

Definition 5.2.8 A recursive type τ' is said to be a syntactic subterm, or just subterm of τ if $\tau' \sqsubseteq \tau$, where $\sqsubseteq$ is defined by the following rules.

$$\tau \sqsubseteq \tau \quad (\textsc{Ref})$$

$$\frac{\tau \sqsubseteq \sigma_1}{\tau \sqsubseteq \sigma_1 \to \sigma_2} \quad (\textsc{Arrow}_L) \qquad \frac{\tau \sqsubseteq \sigma_2}{\tau \sqsubseteq \sigma_1 \to \sigma_2} \quad (\textsc{Arrow}_R)$$

$$\frac{\tau \sqsubseteq \sigma[\mu\alpha.\sigma/\alpha]}{\tau \sqsubseteq \mu\alpha.\sigma} \quad (\textsc{Unfold})$$

$\diamond$

Lemma 5.2.9 *The subterm relation is transitive, i.e. if $\tau \sqsubseteq \delta$, $\delta \sqsubseteq \sigma$ then $\tau \sqsubseteq \sigma$.*

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

Case $\textsc{Ref}$**:** We have $\tau \sqsubseteq \delta$ and $\delta = \sigma$, but then $\tau \sqsubseteq \delta = \sigma$.

Case $\textsc{Arrow}_L$**:** The rule states that $\delta \sqsubseteq \sigma_1$ where $\sigma = \sigma_1 \to \sigma_2$. Since $\tau \sqsubseteq \delta$ we can apply the induction hypothesis and get $\tau \sqsubseteq \sigma_1$. Another application of rule $\textsc{Arrow}_L$ results in $\tau \sqsubseteq \sigma$.

Case $\textsc{Arrow}_R$**:** Similar to $\textsc{Arrow}_L$.

Case $\textsc{Unfold}$**:** $\sigma = \mu\alpha.\sigma'$ and $\delta \sqsubseteq \sigma'[\mu\alpha.\sigma'/\alpha]$ from rule $\textsc{Unfold}$. By IH we get that $\tau \sqsubseteq \sigma'[\mu\alpha.\sigma'/\alpha]$ which directly leads to the desired result. $\diamond$

We define a subterm closure operation on type terms. The intuition of this operation is that it produces the set of all subterms of a recursive type.

Definition 5.2.10 The subterm closure τ^* of τ is the set of recursive types defined by

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

where substitution is applied element wise to sets of recursive types. $\diamond$

A very important property of the closure operation is its commutation with substitution.

Lemma 5.2.11

$$(\tau'[\tau/\beta])^* = (\tau')^*[\tau/\beta] \cup \tau^*$$

where $\beta \in fv(\tau')$.

PROOF Induction on the structure of τ'.

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

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

and the right hand side

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

Where we noted that $\tau \in \tau^*$.

Case $\tau' = \tau_1 \to \tau_2$**:** Since $\beta \in \mathrm{fv}(\tau')$ it must occur free in either τ_1 or τ_2. Induction hypothesis is then that

$$(\tau_1[\tau/\beta])^* = \tau_1^*[\tau/\beta] \cup \tau^*$$

if $\beta \in \mathrm{fv}(\tau_1)$. Assume that $\beta \in \mathrm{fv}(\tau_1)$.

$$\begin{aligned}
&((\tau_1 \to \tau_2)[\tau/\beta])^* = \\
&((\tau_1[\tau/\beta]) \to \tau_2[\tau/\beta])^* = \\
&\{(\tau_1[\tau/\beta]) \to \tau_2[\tau/\beta]\} \cup (\tau_1[\tau/\beta])^* \cup (\tau_2[\tau/\beta])^* = \\
&\{\tau_1 \to \tau_2\}[\tau/\beta] \cup \tau_1^*[\tau/\beta] \cup \tau^* \cup \tau_2^*[\tau/\beta] = \\
&(\{\tau_1 \to \tau_2\} \cup \tau_1^* \cup \tau_2^*)[\tau/\beta] \cup \tau^* = \\
&(\tau_1 \to \tau_2)^*[\tau/\beta] \cup \tau^*
\end{aligned}$$

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

Case $\tau' = \mu\alpha.\sigma$**:** We can assume that $\alpha \notin \mathrm{fv}(\tau)$. Induction hypothesis is

$$(\sigma[\tau/\beta])^* = \sigma^*[\tau/\beta] \cup \tau^*$$

Evaluation of the left hand side

$$\begin{aligned}
&((\mu\alpha.\sigma)[\tau/\beta])^* = \\
&(\mu\alpha.\sigma[\tau/\beta])^* = \\
&\{\mu\alpha.\sigma[\tau/\beta]\} \cup (\sigma[\tau/\beta])^*\left[\mu\alpha.\sigma[\tau/\beta]/_\alpha\right] = \\
&\{\mu\alpha.\sigma\}[\tau/\beta] \cup (\sigma^*[\tau/\beta] \cup \tau^*)\left[\mu\alpha.\sigma[\tau/\beta]/_\alpha\right] = \\
&\{\mu\alpha.\sigma\}[\tau/\beta] \cup \sigma^*[\tau/\beta]\left[\mu\alpha.\sigma[\tau/\beta]/_\alpha\right] \cup \tau^*\left[\mu\alpha.\sigma[\tau/\beta]/_\alpha\right] = \\
&\{\mu\alpha.\sigma\}[\tau/\beta] \cup \sigma^*[\mu\alpha.\sigma/\alpha][\tau/\beta] \cup \tau^* = \\
&(\{\mu\alpha.\sigma\} \cup \sigma^*[\mu\alpha.\sigma/\alpha])[\tau/\beta] \cup \tau^* = \\
&(\mu\alpha.\sigma)^* \cup \tau^*
\end{aligned}$$

◇

With this property, we are able to show that the subterm closure operation is sound with respect to the subterm definition, *i.e.* fulfills our intuition.

Lemma 5.2.12 *If* $\tau \sqsubseteq \sigma$ *then* $\tau \in \sigma^*$.

PROOF Induction on the derivation of $\tau \sqsubseteq \sigma$.

Case REF **:** That is $\tau = \sigma$. By inspection the result immediately follows.

Case ARROW$_L$ **:** We have that $\sigma = \sigma_1 \to \sigma_2$ and that $\tau \sqsubseteq \sigma_1$. Induction hypothesis is that $\tau \in \tau_1^*$. Observe that

$$\tau_1^* \subseteq (\tau_1 \to \tau_2)^* = \{\tau_1 \to \tau_2\} \cup \tau_1^* \cup \tau_2^*$$

We thus get that $\tau \in \sigma^*$.

Case ARROW$_R$ **:** Similar to the above case.

Case UNFOLD **:** So $\sigma = \mu\alpha.\tau'$ and $\tau \sqsubseteq \tau'[\mu\alpha.\tau'/\alpha]$. By IH we get $\tau \in (\tau'[\mu\alpha.\tau'/\alpha])^*$. Since substitution and closure commutes (5.2.11) we 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. ◇

Furthermore, every recursive type has only a finite number of subterms.

Lemma 5.2.13 $|\tau^*| < \infty$.

PROOF Induction on the structure of τ.

Case $\tau = \alpha$ **:** $|\tau^*| = |\{\alpha\}| = 1 < \infty$.

Case $\tau = \tau_1 \to \tau_2$ **:** Induction hypothesis is that $|\tau_1^*| < \infty$ and $|\tau_2^*| < \infty$. The closure of τ is $\tau^* = \{\tau_1 \to \tau_2\} \cup \tau_1^* \cup \tau_2^*$, but then $|\tau^*| = 1 + |\tau_1^*| + |\tau_2^*| < \infty$.

Case $\tau = \mu\alpha.\sigma$ **:** By IH $|\sigma^*| < \infty$ and $\tau^* = \{\mu\alpha.\sigma\} \cup \sigma^*[\mu\alpha.\sigma/\alpha]$ from which we directly conclude the desired result. ◇

Lemmas 5.2.13 and 5.2.12 together finally give us the main property of subterms:

Corollary 5.2.14 *For recursive type τ the set*

$$\{\tau' \mid \tau' \sqsubseteq \tau\}$$

is finite.

PROOF From 5.2.13 we know that $|\tau^*| < \infty$ and 5.2.12 gives that $\{\tau' \mid \tau' \sqsubseteq \tau\} \subseteq \tau^*$ which implies that

$$\left|\{\tau' \mid \tau' \sqsubseteq \tau\}\right| \leq |\tau^*| < \infty$$

◇

Algorithm execution We now concentrate on the computations performed by **S**. The main result is that all terms used during the computation are subterms of the initial terms. Joined with the result obtained above, we can finally prove the termination property of **S**. To reason about the steps performed by **S** we define some general notions of execution trees, chains and paths for algorithms.

Definition 5.2.15 The call tree of a computation $f(\vec{x})$ is the multi branched and labeled tree defined by

$$\text{Call}(f(\vec{x})) \;=\; f(\vec{x}) : \Big[\text{Call}(g_1(\vec{x_1})), \ldots, \text{Call}(g_n(\vec{x_n}))\Big]$$

where $g_1(\vec{x_1}), \ldots, g_n(\vec{x_n})$ are function calls (in chronological order) issued by $f(\vec{x})$. $\diamond$

Definition 5.2.16 The call chain of $f_0(\vec{x_0})$ is the preorder traversal of $\text{Call}(f_0(\vec{x_0}))$, *i.e.* the list of all calls (node labels) in chronological order

$$\Big[f_0(\vec{x_0}), \ldots, f_i(\vec{x_i}), \ldots\Big]$$

A call path of $f_0(x_0)$ is a list of calls corresponding to the labels along a tree path in $\text{Call}(f_0(x_0))$. $\diamond$

Observe that call trees might be infinitely deep if the function never terminates. Infinite call trees naturally results in infinite chains and paths.

Theorem 5.2.17 *Let τ_0, σ_0 be canonical types and A_0 an assumption set. If*

$$\mathbf{S}(A_1, \tau_1, \sigma_1), \ldots, \mathbf{S}(A_n, \tau_n, \sigma_n), \ldots$$

is the call chain of $\mathbf{S}(A_0, \tau_0, \sigma_0)$ then for all $i \geq 0$ $\tau_i, \sigma_i \in \mu Tp$ and

$$(\tau_i \sqsubseteq \tau_0 \;\wedge\; \sigma_i \sqsubseteq \sigma_0) \quad \textit{or} \quad (\tau_i \sqsubseteq \sigma_0 \;\wedge\; \sigma_i \sqsubseteq \tau_0)$$

The requirement of canonical terms is very important since $\mathbf{S}$ surely would run infinitely on $\mu\alpha.\alpha$.

PROOF Induction on i.

$\boxed{i = 0}$ Trivial from reflexivity of $\sqsubseteq$.

$\boxed{i > 0}$ Case analysis of rules producing the i'th execution step.

Case $\mathbf{S}(A, \mu\alpha.\tau, \sigma)$: Step i is then $\mathbf{S}(A, \tau[\mu\alpha.\tau/\alpha], \sigma)$. By IH we know that $\mu\alpha.\tau$ and σ are canonical 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 $\tau[\mu\alpha.\tau/\alpha]$ also is canonical. If we assume that $\mu\alpha.\tau \sqsubseteq \tau_0$ (second case similar) then by 5.2.9 we get

$$\dfrac{\dfrac{\dfrac{}{\tau[\mu\alpha.\tau/\alpha] \sqsubseteq \tau[\mu\alpha.\tau/\alpha]}\ (\text{Ref})}{\tau[\mu\alpha.\tau/\alpha] \sqsubseteq \mu\alpha.\tau}\ (\mu) \qquad \dfrac{}{\mu\alpha.\tau \sqsubseteq \tau_0}\ (\text{IH})}{\tau[\mu\alpha.\tau/\alpha] \sqsubseteq \tau_0}\ (\text{Trans})$$

and of course $\sigma \sqsubseteq \sigma_0$.

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

Case $\mathbf{S}(A, \tau_1 \to \tau_2, \sigma_1 \to \sigma_2)$: Two steps arises from this rule.

1. $\mathbf{S}(A', \sigma_1, \tau_1)$. By IH we know that $\tau_1 \to \tau_2 \sqsubseteq \tau_0$ and $\sigma_1 \to \sigma_2 \sqsubseteq \sigma_0$ or opposite. But then the result follows directly from transitivity (5.2.9) since $\tau_1 \sqsubseteq \tau_1 \to \tau_2$ and $\sigma_1 \sqsubseteq \sigma_1 \to \sigma_2$ from rule ARROW_L.
2. $\mathbf{S}(A', \tau_2, \sigma_2)$ Exactly as previous case.

◇

Lemma 5.2.18 *If* $p = [\mathbf{S}(A_0, \tau_0, \sigma_0), \ldots, \mathbf{S}(A_i, \tau_i, \sigma_i), \ldots]$ *is a call path of* $\mathbf{S}(A_0, \tau_0, \sigma_0)$ *then*

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

PROOF Call $i+1$ in p corresponds to a node in the call tree at a deeper level that call i. A call at a deeper level than another in a call tree of $\mathbf{S}$, means that the deepest call inherits all the assumptions of its ancestors, since assumption sets always are *expanded*, never contracted. The property follows directly from this observation. ◇

We can now give a very strong characterization of call paths in $\mathbf{S}$

Lemma 5.2.19 *If* τ_0, σ_0 *are canonical types,* A_0 *an assumption set and*

$$\mathbf{S}(A_0, \tau_0, \sigma_0), \ldots, \mathbf{S}(A_n, \tau_n, \sigma_n), \ldots$$

a call path of $\mathbf{S}(A_0, \tau_0, \sigma_0)$ *then*

$$\exists N \forall i : (\tau_i, \sigma_i) \in \bigcup_{0 \le j \le N} (\tau_j, \sigma_j)$$

The lemma states that every path only consists of a finite set of different call patterns, even though the paths may be infinite.

PROOF The statement is proven by contradiction. Assume that

$$\forall N \exists i : (\tau_i, \sigma_i) \notin \bigcup_{0 \le j \le N} (\tau_j, \sigma_j)$$

This fact directly implies that

$$\bigcup_{\infty} (\tau_j, \sigma_j)$$

is an infinite set. 5.2.17 states that all terms in the call chain (or call tree for that matter) are subterms of the initial terms. We thus have

$$\bigcup_{\infty} (\tau_j, \sigma_j) \subseteq (\bigcup_{\infty} \tau_j) \times (\bigcup_{\infty} \sigma_j) \subseteq \{\tau \mid \tau \sqsubseteq \tau_0\} \times \{\sigma \mid \sigma \sqsubseteq \sigma_0\}$$

From Corollary 5.2.14 we get an upper size bound

$$\left| \bigcup_{\infty} (\tau_j, \sigma_j) \right| \le \left| \{\tau \mid \tau \sqsubseteq \tau_0\} \right| \left| \{\sigma \mid \sigma \sqsubseteq \sigma_0\} \right| < \infty$$

Which contradicts the fact that $\cup_\infty (\tau_j, \sigma_j)$ is infinite — our assumption is thus false and the proposition therefore true. ◇

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

Theorem 5.2.20 *(Termination of* $\mathbf{S}$*) If* τ*,* σ *are canonical and* A *an assumption set then the call* $\mathbf{S}(A, \tau, \sigma)$ *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 $\text{Call}(\mathbf{S}(A, \tau, \sigma))$. Let N be determined by 5.2.19 such that

$$\forall i : (\tau_i, \sigma_i) \in \bigcup_{0 \leq j \leq N} (\tau_j, \sigma_j) \tag{5.1}$$

Let us consider the calls (τ_i, σ_i) of p where $i > N$. There must exist a call of the form $(\tau_1 \to \tau_2, \sigma_1 \to \sigma_2)_n$ with $n > N$, because otherwise all the calls would be unfoldings, which is not possible since the terms are canonical (thm. 5.2.17). From (5.1) we conclude that $(\tau_1 \to \tau_2, \sigma_1 \to \sigma_2)_n \in \bigcup_{0 \leq j \leq N} (\tau_j, \sigma_j)$ which implies that there was an earlier ARROW call in p. The assumption set associated with call n inherits all assumptions from its ancestors in p (by 5.2.18), but then call n must be a HYP call, which corresponds to a leaf in the call tree. Path p is therefore not infinite and the assumption is false. $\diamond$

Correctness of S

This final part shows that the derivation produced by $\mathbf{S}$ are valid.

Theorem 5.2.21 *Let* τ*,* σ *be canonical types and* A *an assumption set. If* $\tau \leq_\mu \sigma$ *then* $\mathbf{S}(A, \tau, \sigma)$ *returns a derivation of* $A \vdash \tau \leq \sigma$*.*

PROOF The termination theorem Theorem 5.2.20 gives that $\mathbf{S}(A, \tau, \sigma)$ terminates with, say, n recursive calls. Correctness is proven by induction on recursive steps n.

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

Case $\mathbf{S}(A(\tau \leq \sigma), \tau, \sigma)$**:** HYP.

Case $\mathbf{S}(A, \alpha, \alpha)$**:** REF

Case $\mathbf{S}(A, \bot, \tau)$**:** $\bot$

Case $\mathbf{S}(A, \tau, \top)$**:** $\top$

Case $\mathbf{S}(A, \tau, \sigma)$**:** If this rule is reached it means that $\mathcal{L}(\tau) \neq \bot$, $\mathcal{L}(\sigma) \neq \top$ and $\mathcal{L}(\tau) \neq \mathcal{L}(\sigma)$, *i.e.* $\mathcal{L}(\tau) \not\leq \mathcal{L}(\sigma)$. Since $\leq_\mu$ is a simulation and $\tau \leq_\mu \sigma$ it must hold that $\mathcal{L}(\tau) \leq \mathcal{L}(\sigma)$, but this contradicts the fact $\mathcal{L}(\tau) \not\leq \mathcal{L}(\sigma)$. This rule is never reached!

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

Case $\mathbf{S}(A, \mu\alpha.\tau, \sigma)$**:** Next call is $\mathbf{S}(A, \tau[\mu\alpha.\tau/\alpha], \sigma)$, which surely consists of less recursive calls. It is easily seen that $\tau[\mu\alpha.\tau/\alpha]$, σ are canonical and since $\leq_\mu$ is a simulation it follows that $\tau[\mu\alpha.\tau/\alpha] \leq_\mu \sigma$. The induction hypothesis does thus apply and gives that $\mathbf{S}(A, \tau[\mu\alpha.\tau/\alpha], \sigma)$ is a derivation of $A \vdash \tau[\mu\alpha.\tau/\alpha] \leq$

σ. By UNFOLD and TRANS we then get $A \vdash \mu\alpha.\tau \leq \sigma$ which exactly is what $\mathbf{S}(A, \mu\alpha.\tau, \sigma)$ returns.

Case $\mathbf{S}(A, \tau, \mu\beta.\sigma)$: Next call $\mathbf{S}(A, \tau, \sigma[\mu\beta.\sigma/\beta])$ is of $n-1$ recursive calls and the term are, as previously noted, canonical. Now that $\leq_\mu$ is a simulation (3.2.4) we get $\tau \leq_\mu \sigma[\mu\beta.\sigma/\beta]$. By IH we get that $\mathbf{S}(A, \tau, \sigma[\mu\beta.\sigma/\beta])$ proves $A \vdash \tau \leq \sigma[\mu\beta.\sigma/\beta]$. We conclude

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

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

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

1. $\mathbf{S}(A', \sigma_1, \tau_1)$. From simulation property of $\leq_\mu$ and IH we get that the call proves $A' \vdash \sigma_1 \leq \tau_1$.
2. $\mathbf{S}(A', \tau_2, \sigma_2)$. As above we conclude by simulation property and IH that $A' \vdash \tau_2 \leq \sigma_2$ holds.

$\mathbf{S}(A, \tau_1 \to \tau_2, \sigma_1 \to \sigma_2)$ returns rule ARROW with the two valid subproofs shown above. We thus have a correct proof of

$$A \vdash \tau_1 \to \tau_2 \leq \sigma_1 \to \sigma_2$$

◇

Theorem 5.2.22 *For canonical recursive types τ, σ*

$$\tau \leq_\mu \sigma \quad \Rightarrow \quad \vdash \tau \leq \sigma$$

PROOF If $A = \emptyset$ Theorem 5.2.21 directly proves the theorem. ◇

Chapter 6

Coercions

When dealing with subtyping in programming languages one is often faced with the problem of value representation. Values of a specific type are typically associated with a specific machine representation. For instance integers are normally represented as 32-bit twos complement, while reals are represented as 80-bit IEEE floating point structures. When introducing subtyping you would like integers to become a subtype of reals and hence valid real values, but the representation scheme for the two is not quite the same. What you need is a translation from values represented as integers to values represented as reals. Such translations are normally called *coercions*. Coercions for subtyping have been described and investigated by [Ama90, BrTa89, AC93]. In this chapter we develop a theory of coercions based on the subtype relation discussed in the previous chapters.

6.1 Coercions-as-Proofs

This section aims at constructing a language of coercions powerful enough to translate all instances of the subtype relation. We achieve this by defining a term language which encodes proofs in the subtype relation. In this way there exists a coercion for every pair of types related by the subtype relation. Consider for instance the rule (UNFOLD)

$$A \vdash \mu\alpha.\tau \leq \tau[\mu\alpha.\tau/\alpha]$$

Since the rule is an axiom it seems natural that the corresponding coercion should be a primitive. We have decided to name it **unfold**, because it is exactly the operation we require. Now, if v is a value of type $\mu\alpha.\tau$ then **unfold**(v) is a value of type $\tau[\mu\alpha.\tau/\alpha]$, *i.e.* v is coerced from $\mu\alpha.\tau$ to $\tau[\mu\alpha.\tau/\alpha]$.

A more interesting judgement is transitivity,

$$\frac{A \vdash \tau \leq \delta \quad A \vdash \delta \leq \sigma}{A \vdash \tau \leq \sigma}$$

Assume that c_1, c_2 are coercions representing the proofs of $A \vdash \tau \leq \delta$ and $A \vdash \delta \leq \sigma$ respectively. To obtain a coercion from τ to σ you simply compose c_1 and c_2, which we choose to write in standard categorical manner as $c_1; c_2$ instead of $c_2 \circ c_1$.

None of the two presented subtype rules use the assumption set A which is our way of encoding the coinductive nature of the subtype relation. How this coinductiveness is expressed in the coercions is quite interesting. Consider first rule (ARROW/FIX),

$$\frac{A(\tau_1 \to \tau_2 \leq \sigma_1 \to \sigma_2) \vdash \sigma_1 \leq \tau_1 \quad A(\tau_1 \to \tau_2 \leq \sigma_1 \to \sigma_2) \vdash \tau_2 \leq \sigma_2}{A \vdash \tau_1 \to \tau_2 \leq \sigma_1 \to \sigma_2}$$

By adding the conclusion to the assumptions we allow the sub derivations to exploit what we are attempting to prove. In coercion terms this behavior means that the subcoercions can refer to the coercion they are defining. This concept is normally known as definition by recursion and we formalize it by annotating the assumption with a fresh coercion variable. Whenever the assumption is applied in the proof we just use the associated coercion variable to make the recursive reference. In order for this variable reference to make sense we have to bind it somewhere and since we are modeling recursion it seems appropriate to name this binder **fix** . Besides the **fix** construction we need to encode the proof of the two arrow types being in the subtype relation. For this we simply invent a special coercion arrow construct. The entire encoding is presented in Figure 6.1. Note that the assumption sets are now environments mapping coercion variables to type assumptions. We have included an encoding of $c_1 \to c_2$ without a **fix** binder. This coercion encodes an arrow subtype proof in which the assumption is not applied.

To summarize we give a grammar for the coercions,

$$C := \iota_\tau \mid f \mid \mathbf{fix}\, f.C \to C \mid C \to C \mid C;C \mid \mathbf{fold}_{\mu\alpha.\tau} \mid \mathbf{unfold}_{\mu\alpha.\tau} \mid \bot_\tau \mid \top_\tau$$

One might have expected $\mathbf{fix}\, f.C$ instead of the above, but then all the technical issues of contractivity would resurface and give us the same difficulties as they did for recursive types.

The advantage of encoding proofs as terms is that it is much easier to formulate proof theories, transformations and analysis. We will exploit this property in the following chapters.

6.2 Coercions-as-Typed Programs

In the last section we viewed coercions as subtype proofs. In this section we will view coercions from another perspective, namely as typed programs with an operational semantics.

6.2.1 Type System

The type system for coercions is very simple. The only constructor is $\leq$, which takes two recursive types and builds a coercion type. In order to avoid confusion we call coercion types for type signatures. The type system is actually already familiar, because it is just the encoding of subtype proofs as coercions viewed as a type system (Figure 6.1). Below we show a few standard results about typings which will be useful in later chapters.

Lemma 6.2.1 *(Environment Extension) If* $fv(c) \subseteq \mathrm{dom}(E)$ *and* $E \subseteq E'$ *then*

$$E \vdash c : \tau \leq \sigma \quad \Leftrightarrow \quad E' \vdash c : \tau \leq \sigma$$

$$E \vdash \iota_\tau : \tau \leq \tau \qquad \text{(C1)}$$

$$E \vdash \bot_\tau : \bot \leq \tau \qquad \text{(C2)} \qquad E \vdash \top_\tau : \tau \leq \top \qquad \text{(C3)}$$

$$E \vdash \mathbf{unfold}_{\mu\alpha.\tau} : \mu\alpha.\tau \leq \tau[\mu\alpha.\tau/\alpha] \qquad \text{(C4)}$$

$$E \vdash \mathbf{fold}_{\mu\alpha.\tau} : \tau[\mu\alpha.\tau/\alpha] \leq \mu\alpha.\tau \qquad \text{(C5)}$$

$$\frac{E \vdash c : \tau \leq \delta \quad E \vdash d : \delta \leq \sigma}{E \vdash c; d : \tau \leq \sigma} \qquad \text{(C6)}$$

$$\frac{E \vdash c : \tau \leq \tau' \quad E \vdash d : \sigma \leq \sigma'}{E \vdash (c \to d) : (\tau' \to \sigma) \leq (\tau \to \sigma')} \qquad \text{(C7)}$$

$$\frac{E' = E(f : (\tau' \to \sigma) \leq (\tau \to \sigma')) \quad f \text{ fresh} \qquad E' \vdash c : \tau \leq \tau' \quad E' \vdash d : \sigma \leq \sigma'}{E \vdash \mathbf{fix}\, f.(c \to d) : (\tau' \to \sigma) \leq (\tau \to \sigma')} \qquad \text{(C8)}$$

$$E(f : \tau \leq \sigma)E' \vdash f : \tau \leq \sigma \qquad \text{(C9)}$$

Figure 6.1: Coercions (C) encoding Subtype proofs

Observe that this lemma indirectly shows that the typing rules are invariant for reordering of bindings in the environment.

PROOF The $\Rightarrow$ direction goes by induction on c. The only noteworthy case is fix.

Case $c = \mathbf{fix}\, f.c_1 \to c_2$: Assume $f \notin \mathrm{dom}(E')$. The type rule of fix states that $E(f : (\tau_2 \to \sigma_1) \leq (\tau_1 \to \sigma_2)) \vdash c_1 : \tau_1 \leq \tau_2$ and similar for c_2. We know that $E \subseteq E'$, but then $E(f : (\tau_2 \to \sigma_1) \leq (\tau_1 \to \sigma_2)) \subseteq E'(f : (\tau_2 \to \sigma_1) \leq (\tau_1 \to \sigma_2))$ and by induction hypothesis

$$E'(f : (\tau_2 \to \sigma_1) \leq (\tau_1 \to \sigma_2)) \vdash c_1 : \tau_1 \leq \tau_2$$

$$E'(f : (\tau_2 \to \sigma_1) \leq (\tau_1 \to \sigma_2)) \vdash c_2 : \sigma_1 \leq \sigma_2$$

The fix typing rule thus gives $E' \vdash \mathbf{fix}\, f.(c_1 \to c_2) : (\tau_2 \to \sigma_1) \leq (\tau_1 \to \sigma_2)$. If $f \in \mathrm{dom}(E')$ then we rename f to some fresh variable and repeat the above argument.

The reverse direction follows by straight forward induction. $\diamond$

Lemma 6.2.2 *(Substitution) Let $E'' \vdash d : \tau' \leq \sigma'$. For all E, E' and c where $E'' \subseteq EE'$ and $f \notin \mathrm{dom}(EE')$*

$$E(f : \tau' \leq \sigma')E' \vdash c : \tau \leq \sigma \quad \Leftrightarrow \quad EE' \vdash c[d/f] : \tau \leq \sigma$$

PROOF The $\Rightarrow$ direction is shown by induction on c.

Case $c = f$: In this case $\tau = \tau'$ and $\sigma = \sigma'$. We have that $E'' \vdash d : \tau' \leq \sigma'$ and by Lemma 6.2.1 also $EE' \vdash d : \tau' \leq \sigma'$ since $E'' \subseteq EE'$, which concludes the case because $f[d/f] = d$.

Case $c = g$: We know that $E(f : \tau' \leq \sigma')E' \vdash g : \tau \leq \sigma$, but again by Lemma 6.2.1 we conclude $EE' \vdash g : \tau \leq \sigma$.

Case $c = c_1; c_2$: From the typing rule of composition we know that $E(f : \tau' \leq \sigma')E' \vdash c_1 : \tau \leq \delta$ and $E(f : \tau' \leq \sigma')E' \vdash c_2 : \delta \leq \sigma$ for some δ. By induction hypothesis we get $EE' \vdash c_1[d/f] : \tau \leq \delta$ and $EE' \vdash c_2[d/f] : \delta \leq \sigma$. Apply the composition typing rule again,

$$EE' \vdash (c_1[d/f]); (c_2[d/f]) : \tau \leq \sigma$$

and because $(c_1; c_2)[d/f] = (c_1[d/f]); (c_2[d/f])$ we are finished.

Case $c = c_1 \to c_2$: As composition case.

Case $c = \mathbf{fix}\, g.(c_1 \to c_2)$: Here $\tau = \tau_2 \to \sigma_1$, $\sigma = \tau_1 \to \sigma_2$. The fix typing rule states that

$$E(f : \tau' \leq \sigma')E'(g : \tau \leq \sigma) \vdash c_1 : \tau_1 \leq \tau_2$$

and likewise for c_2. Because $E'' \subseteq EE' \subseteq EE'(g : \tau \leq \sigma)$ we may apply the induction hypothesis,

$$EE'(g : \tau \leq \sigma) \vdash c_1[d/f] : \tau_1 \leq \tau_2$$

$$EE'(g : \tau \leq \sigma) \vdash c_2[d/f] : \sigma_1 \leq \sigma_2$$

and since $\mathbf{fix}\ g.(c_1[d/f] \to c_2[d/f]) = (\mathbf{fix}\ g.c_1 \to c_2)[d/f]$ we conclude the case by the fix typing rule.

The opposite direction ($\Leftarrow$) follows the exact same schema as the above shown. $\diamond$

Another type theoretically property is uniqueness of typing.

Lemma 6.2.3 *If* $E \vdash c : \tau \leq \sigma$ *and* $E \vdash c : \tau' \leq \sigma'$ *then* $\tau \leq \sigma \equiv \tau' \leq \sigma'$.

PROOF Easy induction proof on derivation of $E \vdash c : \tau \leq \sigma$. $\diamond$

6.2.2 Operational Semantics

This section discusses the operational semantics of the coercion language. As mentioned in the introduction coercions are supposed to translate between value representations. We will not provide a formal operational semantics, but merely give an intuition and hopefully thereby a better understanding and motivation for the remaining theoretical chapters on coercions. From the intuitive presentation should it be easy to give an operational semantics (translate coercions to lambda expressions and use some standard operational semantics for lambda calculus).

$C := \iota_\tau$ The identity coercion is simply the identity function, that is it does not change the representation at all.

$C := \mathbf{unfold}_{\mu\alpha.\tau}$ Values of the recursive type $\mu\alpha.\tau$ are usually represented as pointers to values of type τ which in turn may reference them selves. Unfolding the recursive value means to access its body, that is dereference its pointer. The unfold coercion thus corresponds to pointer dereferencing which is also known as *unboxing*.

$C := \mathbf{fold}_{\mu\alpha.\tau}$ The fold coercion is the inverse of unfold. Its operational semantics is therefore to create a reference to the value applied to, or in other words *boxing*.

$C := \bot_\tau$ Normally the type $\bot$ is interpreted as non-termination. Under this interpretation the $\bot_\tau$ coercion can be thought of as an arbitrary function, since it will *never* be applied because the preceding computation will never terminate.

$C := \top_\tau$ The only value of type $\top$ is the top value $\top$, sometimes named unit or $()$. The $\top_\tau$ coercion returns the top value whenever applied to a value of type τ.

$C := C; C$ Composition of coercions is straight forward.

$C := C \to C$ The arrow coercion is somewhat more interesting. Let $c = c_1 \to c_2$ be such that $c : (\tau_2 \to \sigma_1) \leq (\tau_1 \to \sigma_2)$ and $c_1 : \tau_1 \leq \tau_2$, $c_2 : \sigma_1 \leq \sigma_2$. We now have to construct a function which given a value of type $\tau_2 \to \sigma_1$ produces a value of type $\tau_1 \to \sigma_2$. Assume x is a value of the require input type and let $v : \tau_1$. If we apply c_1 on v we get $c_1(v) : \tau_2$ and hence we can apply x to obtain $x(c_1(v)) : \sigma_1$. To finally get the right result type we must apply c_2 which then gives $c_2(x(c_1(v))) : \sigma_2$. The arrow coercion can thus be formulated as

$$c_1 \to c_2 = \lambda x.c_1; x; c_2$$

that is apply c_1 on the input and c_2 on the output.

$C := \mathbf{fix}\ f.C \to C$ The body of the fix has of course the same semantics as described above. Operationally viewed fix gives a name to the arrow coercion such that it may reference itself, that is call itself recursively.

$C := f$ This coercion is merely a recursive call to the coercion named f by a fix construct as described in the previous paragraph.

Chapter 7

Coercion Coherence

In the last chapter we viewed coercions as subtype proofs, that is to every subtype proof there was a unique coercion encoding it. An interesting observation in this respect is that subtype proofs are *not* unique in the sense that two different derivations may proof the same judgement. This conjecture is easily demonstrated by example,

$$\vdash \mathbf{unfold}; \mathbf{fold} : \mu\alpha.\tau \qquad \vdash \iota_{\mu\alpha.\tau}$$

both encode a subtype proof of $\mu\alpha.\tau \leq \mu\alpha.\tau$. The fact that subtype proofs are not unique raises the question of coercion coherence:

> Are coercions corresponding to derivations proving the same subtype judgement related?

This chapter is devoted to answer this particular question. Since coercions are subtype proofs this chapter might as well be read as a chapter on subtype proof theory, where we investigate the relationship between derivations proving the same subtype judgement. It will be evident during the chapter that a proof encoding mechanism is indeed required in order to come thru.

7.1 Coercion Equality Relation

In this section we seek to relate coercions proving the same subtype judgement. It seems reasonable to name the relation coercion equality.

When we investigated coercion interaction we quickly realized that coercions and recursive types are not that different considering their fundamental properties. What intrigued us when comparing recursive types is also present when comparing recursive coercions, namely the coinductive interaction due to the recursion constructor. Fortunately, we have already developed a technique for capturing this aspect in axiomatizations – the coinductive rule. Though similar to recursive types, coercions still possess major challenges. The big difference is that the coercion language has a constructor which is not free. We say that a constructor $\cdot$ is free if it satisfy a structural equivalence property,

$$\tau \cdot \sigma = \tau' \cdot \sigma' \;\Leftrightarrow\; \tau = \tau \;\wedge\; \sigma = \sigma'$$

Coercion composition is not a free constructor, because it is associative,

$$c_1; (c_2; c_3) = (c_1; c_2); c_3$$

which obviously results in conflicts with the structural equivalence property stated above. Another cause of difficulty is due to the fact that composition distributes over arrows,

$$(c' \to d); (c \to d') = (c; c') \to (d; d')$$

These equalities are definitely needed since the left and right hand side both encode the same judgement (inspect). The identity coercion also plays a special role. Consider the following intuitively reasonable equalities,

$$\iota_\tau; c = c \qquad \iota_{\tau_1 \to \tau_2} = \iota_{\tau_1} \to \iota_{\tau_2} \qquad \iota_{\mu\alpha.\tau} = \mathbf{unfold}; \mathbf{fold}$$

The first equality says that an identity coercion really is nothing but a no-op operation. The two others state that coercions with identity type signature have no more operational contents than that of an identity coercion (with same type signature, of course). The $\bot$ and $\top$ coercions also interact with composition. Considering the discussion on operational semantics in Section 6.2.2 the following equalities seem natural,

$$\bot; c = \bot \quad c; \top = \top \quad \bot_\top = \top_\bot \quad \bot_\bot = \iota_\bot \quad \top_\top = \iota_\top$$

The entire equality relation is shown in Figure 7.1.

The formulation of the equality relation is a little imprecise about types of the coercions. The intention is of course that all coercions are well-typed with the same type signature. We should, to be precise, have formulated the rules with type signatures, *e.g.*

$$\frac{A \vdash d = c : \tau \leq \sigma}{A \vdash c = d : \tau \leq \sigma} \quad \text{(CE19)}$$

and also included type signatures in the assumption sets A. A rule especially affected by this imprecise formulation is the hypothesis rule (CE14). It should have been formulated like this

$$\frac{\vdash c : \tau \leq \sigma \qquad \vdash d : \tau \leq \sigma}{A(c = d : \tau \leq \sigma)A' \vdash c = d : \tau \leq \sigma} \quad \text{(CE14)}$$

The well-typed conditions on c, d are assumed implicitly in our formulation, but are in fact very crucial. The rationale for this imprecise formulation is that everyone know what we mean and that all the extra type information would clutter theorems and proofs unnecessary.

7.1.1 Subtype Proof Theory

We take here a moment to reflect over the equality relation as a proof theory of the subtype relation. Below we pick out some of the more interesting rules of the equality relation and explain them in a subtype proof setting.

Rules (CE6,7) state that composition with identity does not prove more than the coercion itself. In proof terms composition is transitivity and identity

$$A \vdash \bot; c = \bot \quad \text{(CE1)} \qquad A \vdash c; \top = \top \quad \text{(CE2)}$$

$$A \vdash \bot_\bot = \iota_\bot \quad \text{(CE3)} \qquad A \vdash \top_\top = \iota_\top \quad \text{(CE4)}$$

$$A \vdash \bot_\top = \top_\bot \quad \text{(CE5)}$$

$$A \vdash \iota_\tau; c = c \quad \text{(CE6)} \qquad A \vdash c; \iota_\tau = c \quad \text{(CE7)}$$

$$A \vdash \iota_{\tau \to \tau'} = \iota_\tau \to \iota_{\tau'} \quad \text{(CE8)}$$

$$A \vdash \mathbf{unfold}; \mathbf{fold} = \iota_{\mu\alpha.\tau} \quad \text{(CE9)} \qquad A \vdash \mathbf{fold}; \mathbf{unfold} = \iota_{\tau[\mu\alpha.\tau/\alpha]} \quad \text{(CE10)}$$

$$A \vdash c; (c'; c'') = (c; c'); c'' \quad \text{(CE11)}$$

$$A \vdash (c' \to d); (c \to d') = (c; c') \to (d; d') \quad \text{(CE12)}$$

$$\frac{A \vdash c = c' \quad A \vdash d = d'}{A \vdash c; d = c'; d'} \quad \text{(CE13)}$$

$$A(c = d)A' \vdash c = d \quad \text{(CE14)} \qquad A \vdash \mathbf{fix}\, f.c = c[\mathbf{fix}\, f.c/f] \quad \text{(CE15)}$$

$$\frac{A(c \to d = c' \to d') \vdash c = c' \quad A(c \to d = c' \to d') \vdash d = d'}{A \vdash (c \to d) = (c' \to d')} \quad \text{(CE16)}$$

$$\frac{A \vdash c = c' \quad A \vdash c' = c''}{A \vdash c = c''} \quad \text{(CE17)}$$

$$A \vdash c = c \quad \text{(CE18)} \qquad \frac{A \vdash d = c}{A \vdash c = d} \quad \text{(CE19)}$$

Figure 7.1: Coercion Equality (CE) Relation

is reflexivity. Identity compositions thus mean transitivity combined with reflexivity. The rule then says that reflexivity might as well be skipped. Rules (CE3,4,8-10) say that reflexivity is just as good a proof as the one obtained by structurally decomposing or unfolding/folding the types. Rule (CE15) is a bit more deep. It says that a proof may be unfolded meaning that the hypothesis rule is substituted with the entire proof from which the assumption arose. The rule can also be read from right to left and then it means that a proof may be folded if the same derivation is repeated twice (or more times). Finally rules (CE14,16) tell that we may use coinductive assumptions when comparing subtype proofs.

7.2 Completeness

We prove in this section that two coercions encoding the same subtype judgements are related by our equality relation. Completeness is proven by a method similar to the one used in our completeness proof for $\leq$ in Section 5.2.2, though with additional technical complications due to the non-free constructor and the interaction between constructors. In order to handle these problems properly we transform coercions into a restricted form. With precise knowledge of the coercion syntax we can develop an algorithm for determining equality. We prove that the algorithm always terminates and furthermore answers affirmative on coercions with identical type signatures (*i.e.* coercions encoding same subtype judgement).

Reduction system We present a reduction system in which all the non-free constructors of a coercion are forced into fixed positions. In this way we achieve simpler and thus more controllable coercions. To reduce as much as possible we choose to unfold recursive fix coercions, but then we are faced with the problem of strong normalization (termination). Since fix coercions by definition have arrow coercions as bodies an easy way of regaining strong normalization is to prohibit reduction under arrow constructors, that is leave out compatibility on arrow coercions.

Congruence rules:

$$(c_1; c_2); c_3 \;=\; c_1; (c_2; c_3)$$

Reduction rules:

$$\begin{aligned}
\iota_{\tau_1} \to \iota_{\tau_2} &\;\Rightarrow\; \iota_{\tau_1 \to \tau_2} \\
\iota; c &\;\Rightarrow\; c \\
c; \iota &\;\Rightarrow\; c \\
\mathbf{fold}; \mathbf{unfold} &\;\Rightarrow\; \iota \\
\mathbf{unfold}; \mathbf{fold} &\;\Rightarrow\; \iota \\
\bot; c &\;\Rightarrow\; \bot \\
c; \top &\;\Rightarrow\; \top
\end{aligned}$$

$$
\begin{array}{rcl}
\bot_\bot & \Rightarrow & \iota_\bot \\
\top_\top & \Rightarrow & \iota_\top \\
\bot_\top & \Rightarrow & \top_\bot \\
(c_1 \to c_2);(d_1 \to d_2) & \Rightarrow & (d_1;c_1) \to (c_2;d_2) \\
\textbf{fix}\ f.d & \Rightarrow & d[\textbf{fix}\ f.d/f]
\end{array}
$$

$$
\frac{c \Rightarrow c'}{c;d \Rightarrow c';d} \qquad \frac{d \Rightarrow d'}{c;d \Rightarrow c;d'}
$$

The transitive, reflexive closure of $\Rightarrow$ is denoted with $\Rightarrow^*$. It follows by inspection of the relation that normal forms of well-formed coercions are given by the grammar

$$
\begin{array}{rl}
C_{nf} = & \iota_\tau \mid \bot_\tau \mid \top_\tau \mid \textbf{fold} \mid \textbf{unfold} \mid C \to C \mid (C \to C);\textbf{fold} \mid \\
& \textbf{unfold};(C \to C) \mid \textbf{unfold};(C \to C);\textbf{fold}
\end{array}
$$

with the following syntactic restrictions,

1. $C \to C \neq \iota \to \iota$
2. $\bot_\tau \neq \bot_\bot$
3. $\bot_\tau \neq \bot_\top$
4. $\top_\tau \neq \top_\top$

C refers to the ordinary grammar on coercions (Section 6.1). Sometimes we relax one of the restrictions on the grammar. In these situations we denote the grammar with C_{nf}^{-r} where r is one of the four restrictions.

Proposition 7.2.1 *(Strong Normalization) For all coercions c there exists normal form c' such that $c \Rightarrow^* c'$. We denote the normal form of c with $nf(c)$.*

PROOF

We devise a measure on coercions (though not in a standard mathematical fashion). For coercion c is $\mid c \mid \in \mathbf{N}^5$ defined by

$$
\mid c \mid \ = \ \left(\mid c \mid^{\textbf{fix}},\ \mid c \mid^{\to},\ \mid c \mid^{;},\ \mid c \mid^{\bot},\ \mid c \mid^{\top} \right)
$$

where $\mid c \mid^x$ measures the number of top level occurrences of constructor x in coercion c,

$$
\begin{array}{lcllcl}
\mid \textbf{fix}\ f.c \mid^x & = & \binom{\textbf{fix}}{x} & \mid f \mid^x & = & \binom{f}{x} \\
\mid c;d \mid^x & = & \binom{;}{x} + \mid c \mid^x + \mid d \mid^x & \mid c \to d \mid^x & = & \binom{\to}{x} \\
 & & & \mid \iota \mid^x & = & \binom{\iota}{x}
\end{array}
$$

where $\binom{x}{x} = 1$ or $\binom{y}{x} = 0$ if $y \neq x$. The intuition is that $\mid c \mid$ counts the number of possible reductions. We have to assert that the measure $0 = (0,0,0,0,0)$ denotes that no more reductions are possible. Case analysis on reduction rules

Case $\iota_{\tau_1} \to \iota_{\tau_2} \Rightarrow \iota_{\tau_1 \to \tau_2}$**:** Since $|\, c \,|^{\to} = 0$ this rule is not applicable.

Case $\iota; c' \Rightarrow c'$**:** We have that $|\, c \,|^{;} = 0$ which means that no top level ; exists. This rule does therefore not apply to c.

Case $c'; \iota \Rightarrow c'$, $\bot; c \Rightarrow \bot$, $c; \top \Rightarrow \top$**:** As above case.

Case $\bot_\bot \Rightarrow \iota_\bot$, $\bot_\top \Rightarrow \top_\bot$**:** The fourth component of the measure is zero thus preventing these reductions.

Case $\top_\top \Rightarrow \iota_\top$**:** Fifth component zero, so no top level $\top$ coercions are present.

Case $(c_1 \to c_2); (d_1 \to d_2) \Rightarrow (d_1; c_1) \to (c_2; d_2)$**:** Not applicable since $|\, c \,|^{;} = 0$.

Case fold; **unfold** $\Rightarrow \iota$**:** As previous case not possible since ; measure is zero.

Case unfold; **fold** $\Rightarrow \iota$**:** Similar to above.

Case fix $f.d \Rightarrow d[\textbf{fix}\ f.d/f]$**:** When $|\, c \,|^{\textbf{flx}} = 0$ then no top level fix expressions exist and therefore rule can not be used.

Case $\dfrac{c' \Rightarrow c''}{c'; d' \Rightarrow c''; d'}$**:** Since $|\, c \,|^{;} = 0$ then c will never match a composition like $c'; d'$.

Case $\dfrac{d' \Rightarrow d''}{c'; d' \Rightarrow c'; d''}$**:** As previous case.

It holds for arbitrary coercion c that $|\, c \,|^{x} < \infty$ for all constructors x, since c has finite syntactically size. Any standard mathematical measure on $\mathbf{N}^5$ then yields that $|\, c \,|$ is finite. Likewise is $|\, c \,|$ always non-negative. We can now prove an interesting property of the reduction system

$$\text{If } c \Rightarrow c' \text{ then } |\, c' \,| < |\, c \,|, \text{ where } \mathbf{N}^5 \text{ is lexically ordered.} \qquad (*)$$

This property actually states that it is only possible to reduce c by finitely many steps. If c was to be reduced infinitely then $|\, c \,|$ would eventually reach zero, which prohibits further reduction and thus contradicts the assumption.

Recall the lexical order on A^n:

$$a < b \;\Leftrightarrow\; \exists i \leq n : a_i < b_i \;\wedge\; \forall j < i : a_j = b_j$$

for $a = (a_1, \ldots, a_n)$, $a = (b_1, \ldots, b_n)$ in A^n. Note $<$ represents both the ordering of A^n and A.

Property $(*)$ is proven by induction on the derivation of $c \Rightarrow c'$.

Case $\iota; c \Rightarrow c$**:** We have by definition that $|\, \iota; c \,|^{\textbf{flx}} = |\, c \,|^{\textbf{flx}}$ and $|\, \iota; c \,|^{\to} = |\, c \,|^{\to}$, but for the third component we get

$$|\, c \,|^{;} < |\, c \,|^{;} + 1 = |\, \iota; c \,|^{;}$$

which concludes the case.

Case $(c; \iota) \Rightarrow c$, (**unfold**; **fold**) $\Rightarrow \iota$, (**fold**; **unfold**) $\Rightarrow \iota$, $(\bot; c) \Rightarrow \bot$, $(c; \top) \Rightarrow \top$**:** All similar to the above case.

Case $\bot_\bot \Rightarrow \iota_\bot$**:** The fourth component of the measure is decreased while the first three remain unaltered.

Case $\top_\top \Rightarrow \iota_\top$ **:** As bottom case above.

Case $\bot_\top \Rightarrow \top_\bot$ **:** Though fifth component is increased the fourth is decreased thus leading to a total decrease of measure.

Case $\iota_\tau \to \iota_{\tau'} \Rightarrow \iota_{\tau\to\tau'}$ **:** The first measure component $|\cdot|^{\mathbf{fix}}$ is definitely zero for both expressions. Consider second component

$$|\,\iota_{\tau\to\tau'}\,|^{\to} = 0 < 1 = |\,\iota_\tau \to \iota_{\tau'}\,|^{\to}$$

Case fix $f.c \Rightarrow c[\mathbf{fix}\ f.c/f]$ **:** Since $c = c_1 \to c_2$ we know that $c[\mathbf{fix}\ f.c/f]$ can not contain a top level fix constructor. In other words

$$|\,c[\mathbf{fix}\ f.c/f]\,|^{\mathbf{fix}} < |\,\mathbf{fix}\ f.c\,|^{\mathbf{fix}}$$

and thus $|\,c[\mathbf{fix}\ f.c/f]\,| < |\,\mathbf{fix}\ f.c\,|$.

Case $(c_1 \to c_2);(d_1 \to d_2) \Rightarrow (d_1;c_1) \to (c_2;d_2)$ **:** No top level **fix** constructors exist in neither expression, so their **fix**-measures are both zero. Their arrow measures satisfy

$$|\,(d_1;c_1) \to (c_2;d_2)\,|^{\to} = 1 < 2 = |\,(c_1 \to c_2);(d_1 \to d_2)\,|^{\to}$$

Case $\dfrac{c \Rightarrow c'}{c;d \Rightarrow c';d}$ **:** By induction hypothesis we conclude that $|\,c'\,| < |\,c\,|$. If $|\,c'\,|^{\mathbf{fix}} < |\,c\,|^{\mathbf{fix}}$ then $|\,c';d\,|^{\mathbf{fix}} < |\,c;d\,|^{\mathbf{fix}}$ and hence the required conclusion. Similarly if it is one of the other measures that determines inequality of c' and c.

Case $\dfrac{d \Rightarrow d'}{c;d \Rightarrow c;d'}$ **:** Exactly as above. ◇

For the reduction system to be useful it must be conservative with respect to equality.

Proposition 7.2.2 *If* $c \Rightarrow^* c'$ *then* $c = c'$.

PROOF Assume that $c \Rightarrow c'$. Observe by inspection that each of the reduction rules correspond directly to a judgement in the equality axiomatization. We thus have $\vdash c = c'$. Now, if $c \Rightarrow^* c'$ we do a simple induction proof on the number of reduction steps and conclude that $\vdash c = c'$. ◇

Algorithm E We define an algorithm which given two normal form coercions determines if they are equal. Since normal form coercions have a simple structure it is easy to determine whether or not they are equivalent. The algorithm is purely syntax directed and if it encounters a coercion which it can not decompose syntactically it tries to expand the coercion by rule (CE8) or (CE9) and (CE10). Note that the coercions generated in this way are not in normal form, but they do fit the normal form grammar syntactically. The algorithm can therefore handle coercions generated from C_{nf}^{-1}, *i.e.* disregarding the side condition about arrow identity coercions.

1: $\mathbf{E}(A, \iota_{\mu\alpha.\tau_1\to\tau_2}, c) = \mathbf{E}(A, (\mathbf{unfold}; \iota_{(\tau_1[T/\alpha])} \to \iota_{(\tau_2[T/\alpha])}; \mathbf{fold}), c)$
2: $\mathbf{E}(A, c, \iota_{\mu\alpha.\tau_1\to\tau_2}) = \mathbf{E}(A, c, (\mathbf{unfold}; \iota_{(\tau_1[T/\alpha])} \to \iota_{(\tau_2[T/\alpha])}; \mathbf{fold}))$
3: $\mathbf{E}(A, \iota_{\tau_1\to\tau_2}, c) = \mathbf{E}(A, \iota_{\tau_1} \to \iota_{\tau_2}, c)$
4: $\mathbf{E}(A, c, \iota_{\tau_1\to\tau_2}) = \mathbf{E}(A, c, \iota_{\tau_1} \to \iota_{\tau_2})$
5: $\mathbf{E}(A, c \to c', d \to d') =$ **if** $(c \to c' = d \to d') \in A$ **then** TRUE **else**
6: $\quad \mathbf{E}(A(c \to c' = d \to d'), \mathrm{nf}(c), \mathrm{nf}(d)) \ \wedge \ \mathbf{E}(A(c \to c' = d \to d'), \mathrm{nf}(c'), \mathrm{nf}(d'))$
7: $\mathbf{E}(A, (\mathbf{unfold}; c \to c'), (\mathbf{unfold}; d \to d')) = \mathbf{E}(A, c \to c', d \to d')$
8: $\mathbf{E}(A, (\mathbf{unfold}; c \to c'), \mathbf{unfold}) = \mathbf{E}(A, c \to c', \iota)$
9: $\mathbf{E}(A, \mathbf{unfold}, (\mathbf{unfold}; c \to c')) = \mathbf{E}(A, \iota, c \to c')$
10: $\mathbf{E}(A, (c \to c'; \mathbf{fold}), (d \to d'; \mathbf{fold})) = \mathbf{E}(A, c \to c', d \to d')$
11: $\mathbf{E}(A, (c \to c'; \mathbf{fold}), \mathbf{fold}) = \mathbf{E}(A, c \to c', \iota)$
12: $\mathbf{E}(A, \mathbf{fold}, (c \to c'; \mathbf{fold})) = \mathbf{E}(A, \iota, c \to c')$
13: $\mathbf{E}(A, (\mathbf{unfold}; c \to c'; \mathbf{fold}), (\mathbf{unfold}; d \to d'; \mathbf{fold})) =$
14: $\quad \mathbf{E}(A, c \to c', d \to d')$
15: $\mathbf{E}(A, c, c) = \mathrm{TRUE}$
16: $\mathbf{E}(\_, \_, \_) = \mathrm{FALSE}$

Figure 7.2: Coercion Equality Algorithm

Algorithm Termination. We prove termination by defining a set which is finite and contains all coercions occurring during execution. As in Section 5.2.2 we can then conclude that execution must terminate since the assumption set eventually will contain *enough* assumptions.

Definition 7.2.3 Define the set of coercions constructed from a type by

$$\mathrm{T}(\tau) \ = \ \{\iota_{\tau'}, \bot_{\tau'}, \top_{\tau'}, \mathbf{unfold}_{\tau'}, \mathbf{fold}_{\tau'} \mid \tau' \sqsubseteq \tau\}$$

where $\sqsubseteq$ is the subterm relation defined in Definition 5.2.8.

$$\mathrm{T}(c : \tau \leq \tau') \ = \ \mathrm{T}(\tau) \cup \mathrm{T}(\tau')$$

First level closure $\mathrm{C}'(c) \ = \ \mathrm{T}(c) \ \cup \ \mathrm{C}'_0(c)$ with

$\mathrm{C}'_0(f) = \{f\}$

$\mathrm{C}'_0(\iota) = \emptyset$

$\mathrm{C}'_0(\bot) = \emptyset$

$\mathrm{C}'_0(\top) = \emptyset$

$\mathrm{C}'_0(\mathbf{unfold}) = \{\mathbf{unfold}\}$

$\mathrm{C}'_0(\mathbf{fold}) = \{\mathbf{fold}\}$

$\mathrm{C}'_0(\mathbf{fix}\ f.d) = \{\mathbf{fix}\ f.d\} \ \cup \ \mathrm{C}'(d)[\mathbf{fix}\ f.d/f]$

$\mathrm{C}'_0(c_1 \to c_2) = \mathrm{C}''(c_1) \ \cup \ \mathrm{C}''(c_2)$

$\mathrm{C}'_0(c_1; c_2) = \mathrm{C}'(c_1) \ \cup \ \mathrm{C}'(c_2)$

Second level closure $\mathrm{C}''(c) \ = \ \mathrm{T}(c) \ \cup \ \mathrm{C}''_0(c)$ with

$$\mathrm{C}_0''(f) = \{f\}$$
$$\mathrm{C}_0''(\iota) = \emptyset$$
$$\mathrm{C}_0''(\bot) = \emptyset$$
$$\mathrm{C}_0''(\top) = \emptyset$$
$$\mathrm{C}_0''(\mathbf{unfold}) = \{\mathbf{unfold}\}$$
$$\mathrm{C}_0''(\mathbf{fold}) = \{\mathbf{fold}\}$$
$$\mathrm{C}_0''(\mathbf{fix}\ f.d) = \{\mathbf{fix}\ f.d\} \cup \mathrm{C}''(d)[\mathbf{fix}\ f.d/f]$$
$$\mathrm{C}_0''(c_1 \to c_2) = \{c_1 \to c_2\} \cup \mathrm{C}''(c_1) \cup \mathrm{C}''(c_2)$$
$$\mathrm{C}_0''(c_1; c_2) = \{c_1; c_2\} \cup \mathrm{C}''(c_1) \cup \mathrm{C}''(c_2)$$

Finally we define the closure of a coercion to be

$$\mathrm{C}^*(c) = \left\{(c_1;..;c_n) \mid n > 0,\ c_i \in \mathrm{C}'(c),\ (c_1;..;c_n) \text{ is well-typed and cycle free}\right\}$$

A composition $c_1;..;c_m$ of non-compositions $c_i : \tau_i \leq \tau_{i+1}$ is called cycle free if and only if

$$k \neq l \Rightarrow \tau_k \neq \tau_l$$

for all k, l. ◇

The closure definitions satisfy our first requirement

Proposition 7.2.4 *For all canonical types τ and coercions c*

1. $\mathrm{T}(\tau)$, $\mathrm{T}(c)$ *finite*
2. $\mathrm{C}_0'(c)$, $\mathrm{C}'(c)$, $\mathrm{C}_0''(c)$, $\mathrm{C}''(c)$ *and* $\mathrm{C}^*(c)$ *finite*

PROOF Ad 1). By Corollary 5.2.14 we know that $\{\tau' \mid \tau' \sqsubseteq \tau\}$ is finite. From this we immediately obtain the results.

Ad 2). Consider $\mathrm{C}_0''(c)$. An easy induction reveals that it must be finite and therefore $\mathrm{C}''(c)$ is also finite. A consequence of this result is that $\mathrm{C}_0'(c)$ and $\mathrm{C}'(c)$ are finite. Now the only missing set is $\mathrm{C}^*(c)$. If $\mathrm{C}^*(c)$ was to include a composition of more than $|\mathrm{C}'(c)|$ coercions then at least *one* has to be included twice and hence introduces a type cycle, which conflicts the definition. We conclude that all compositions are of size less than or equal to $|\mathrm{C}'(c)|$, which is finite. Disregarding type-ability $\mathrm{C}^*(c)$ is thus bounded by the total number of permutations in the finite set $\mathrm{C}'(c)$. ◇

Some useful properties include

Lemma 7.2.5 *For all canonical coercions c, d*

1. $\mathrm{C}'(c) \subseteq \mathrm{C}^*(c)$
2. $\mathrm{C}'(c) \subseteq \mathrm{C}''(c)$
3. $(\mathrm{C}'(c) \subseteq \mathrm{C}'(d)) \Rightarrow (\mathrm{C}^*(c) \subseteq \mathrm{C}^*(d))$
4. $c \in \mathrm{C}''(c)$

PROOF
Ad 1). Trivial since $\mathrm{C}^*(c)$ is constructed from $\mathrm{C}'(c)$.
Ad 2). By inspection.
Ad 3). Assume $(\mathrm{C}'(c) \subseteq \mathrm{C}'(d))$ and let $c_1;..;c_n \in \mathrm{C}^*(c)$. Since $c_i \in \mathrm{C}'(c) \subseteq \mathrm{C}'(d)$ for all i then $c_1;..;c_n \in \mathrm{C}^*(d)$.
Ad 4). By inspection. ◇

The first level coercion closure satisfies a substitution lemma.

Lemma 7.2.6 *(Substitution) If* $E \vdash c : \tau \leq \tau'$ *and* $(f : \sigma \leq \sigma') \in E$, $E \vdash d : \sigma \leq \sigma'$ *then*

$$\mathrm{C}'(c[d/f]) \subseteq \mathrm{C}'(c)[d/f] \cup \mathrm{C}'(d)$$

where substitution is extended to sets point wise.

PROOF By induction on coercion c.

Case $c = g \neq f$**:**

$$\mathrm{C}'(g[d/f]) = \mathrm{C}'(g) = \mathrm{C}'(g)[d/f] \subseteq \mathrm{C}'(g)[d/f] \cup \mathrm{C}'(d)$$

Case $c = f$**:**

$$\mathrm{C}'(f[d/f]) = \mathrm{C}'(d) \subseteq \{d\} \cup \mathrm{C}'(d) \cup \mathrm{T}(f) = \mathrm{C}'(f)[d/f] \cup \mathrm{C}'(d)$$

Case $c = c_1; c_2$**:** Consider

$$\begin{aligned}
\mathrm{C}'((c_1;c_2)[d/f]) &= \mathrm{C}'((c_1[d/f]);(c_2[d/f])) \\
&= \mathrm{C}'(c_1[d/f]) \cup \mathrm{C}'(c_2[d/f]) \cup \mathrm{T}((c_1[d/f]);(c_2[d/f])) \\
&\subseteq \mathrm{C}'(c_1)[d/f] \cup \mathrm{C}'(c_2)[d/f] \cup \mathrm{C}'(d) \cup \mathrm{T}((c_1[d/f]);(c_2[d/f])) \\
&= \mathrm{C}'(c_1)[d/f] \cup \mathrm{C}'(c_2)[d/f] \cup \mathrm{C}'(d) \cup \mathrm{T}(c_1;c_2)[d/f] \\
&= \mathrm{C}'(c_1;c_2)[d/f] \cup \mathrm{C}'(d)
\end{aligned}$$

where we applied IH twice and the fact that $\mathrm{T}(c[d/f]) = \mathrm{T}(c)[d/f]$.

Case $c = c_1 \to c_2$**:** Similar to $c = c_1; c_2$.

Case $c = \mathbf{fix}\ g.c'$**:** We can assume that $g \neq f$

$$\begin{aligned}
\mathrm{C}'((\mathbf{fix}\ g.c')[d/f]) &= \mathrm{C}'(\mathbf{fix}\ g.(c'[d/f])) \\
&= \mathrm{T}(\mathbf{fix}\ g.(c'[d/f])) \cup \mathrm{C}'_0(\mathbf{fix}\ g.(c'[d/f])) \\
&= \mathrm{T}(\mathbf{fix}\ g.(c'[d/f])) \cup \{\mathbf{fix}\ g.(c'[d/f])\} \cup \mathrm{C}'(c'[d/f]) \\
&\subseteq \mathrm{T}(\mathbf{fix}\ g.c')[d/f] \cup \{\mathbf{fix}\ g.c'\}[d/f] \cup \mathrm{C}'(c')[d/f] \cup \mathrm{C}'(d) \\
&= (\mathrm{T}(\mathbf{fix}\ g.c') \cup \{\mathbf{fix}\ g.c'\} \cup \mathrm{C}'(c'))[d/f] \cup \mathrm{C}'(d) \\
&= \mathrm{C}'(\mathbf{fix}\ g.c')[d/f] \cup \mathrm{C}'(d)
\end{aligned}$$

◇

A very important aspect of the closure operation is its relation to reduction.

Lemma 7.2.7 *If* $c \Rightarrow^* c'$ *then* $\mathrm{C}^*(c') \subseteq \mathrm{C}^*(c)$.

PROOF We prove lemma for single step reduction which then implies multi step reduction. Induction on derivation of $c \Rightarrow c'$. Not all cases are shown.

Case $\iota_{\tau_1} \to \iota_{\tau_2} \Rightarrow \iota_{\tau_1 \to \tau_2}$ **:**

$$\mathrm{C}'(\iota_{\tau_1 \to \tau_2}) = \mathrm{T}(\tau_1 \to \tau_2)$$

$$\mathrm{C}'(\iota_{\tau_1} \to \iota_{\tau_2}) = \mathrm{T}(\tau_1 \to \tau_2) \cup \mathrm{C}''(\iota_{\tau_1}) \cup \mathrm{C}''(\iota_{\tau_2})$$

Case is concluded by lemma 7.2.5 3).

Case $(\iota; c) \Rightarrow c$ **:** Obvious.

Case $(c_1 \to c_2); (d_1 \to d_2) \Rightarrow (d_1; c_1) \to (c_2; d_2)$ **:** Consider the coercion signatures:

Coercion	*Signature*	*Coercion*	*Signature*
c_1	$\delta_1 \leq \tau_1$	d_1	$\sigma_1 \leq \delta_1$
c_2	$\tau_2 \leq \delta_2$	d_2	$\delta_2 \leq \sigma_2$
$(c_1 \to c_2)$	$(\tau_1 \to \tau_2) \leq (\delta_1 \to \delta_2)$	$(d_1; c_1)$	$\sigma_1 \leq \tau_1$
$(d_1 \to d_2)$	$(\delta_1 \to \delta_2) \leq (\sigma_1 \to \sigma_2)$	$(c_2; d_1)$	$\tau_2 \leq \sigma_2$
$\Big((c_1 \to c_2); (d_1 \to d_2)\Big)$	$(\tau_1 \to \tau_2) \leq (\sigma_1 \to \sigma_2)$		
$\Big((d_1; c_1) \to (c_2; d_2)\Big)$	$(\tau_1 \to \tau_2) \leq (\sigma_1 \to \sigma_2)$		

First level closure of right hand side.

$$\begin{aligned}
&\mathrm{C}'((d_1; c_1) \to (c_2; d_2)) = \\
&\quad = T \cup \mathrm{C}'_0((d_1; c_1) \to (c_2; d_2)) \\
&\quad = T \cup \mathrm{C}''(d_1; c_1) \cup \mathrm{C}''(c_2; d_2) \\
&\quad = T \cup \{(d_1; c_1),\ (c_2; d_2)\} \cup \mathrm{C}''(c_1) \cup \mathrm{C}''(c_2) \cup \mathrm{C}''(d_1) \cup \mathrm{C}''(d_2)
\end{aligned}$$

where $T = \mathrm{T}(\tau_1 \to \tau_2) \cup \mathrm{T}(\sigma_1 \to \sigma_2)$. Left hand side

$$\begin{aligned}
&\mathrm{C}'((c_1 \to c_2); (d_1 \to d_2)) = \\
&\quad = T \cup \mathrm{C}'_0((c_1 \to c_2); (d_1 \to d_2)) \\
&\quad = T \cup \mathrm{C}'(c_1 \to c_2) \cup \mathrm{C}'(d_1 \to d_2) \\
&\quad = T \cup \mathrm{T}(\delta_1 \to \delta_2) \cup \mathrm{C}''(c_1) \cup \mathrm{C}''(c_2) \cup \mathrm{C}''(d_1) \cup \mathrm{C}''(d_2)
\end{aligned}$$

Let $(c_1; ..; c_n) \in \mathrm{C}^*((d_1; c_1) \to (c_2; d_2))$. If $c_i \notin \{(d_1; c_1),\ (c_2; d_2)\}$ for all i then $c_i \in \mathrm{C}'((c_1 \to c_2); (d_1 \to d_2))$ and therefore $(c_1; ..; c_n) \in \mathrm{C}^*((c_1 \to c_2); (d_1 \to d_2))$. Assume now that $c_j = (d_1; c_1)$ for some j. Since $c_1 \in \mathrm{C}''(c_1)$ and $d_1 \in \mathrm{C}''(d_1)$ (Lemma 7.2.5 (4)) then $(d_1; c_1) \in \mathrm{C}^*((c_1 \to c_2); (d_1 \to d_2))$. In this way we again obtain $(c_1; ..; c_n) \in \mathrm{C}^*((c_1 \to c_2); (d_1 \to d_2))$.

Case $(\mathbf{fold}; \mathbf{unfold}) \Rightarrow \iota$ **:**

$$\mathrm{C}'(\iota_\tau) = \mathrm{T}(\tau)$$

$$\mathrm{C}'(\mathbf{fold}; \mathbf{unfold}) = \mathrm{T}(\tau) \cup \mathrm{T}(\mu\alpha.\tau') \cup \{\mathbf{fold}, \mathbf{unfold}\}$$

Since $(\mathbf{fold}; \mathbf{unfold}) : \tau \leq \tau$. By Lemma 7.2.5 (3) we get $\mathrm{C}^*(\iota_\tau) \subseteq \mathrm{C}^*(\mathbf{fold}; \mathbf{unfold})$.

Case $\mathbf{fix}\, f.c \Rightarrow c[\mathbf{fix}\, f.c/f]$**:** Since $\mathbf{fix}\, f.c$ is well-formed it is also typable, that is there exists E such that

$$\frac{E(f : \tau \leq \tau') \vdash c : \tau \leq \tau'}{E \vdash \mathbf{fix}\, f.c : \tau \leq \tau'}$$

for some τ, τ'. By substitution lemma (Lemma 6.2.2) we get

$$\mathrm{C}'(c[\mathbf{fix}\, f.c/f]) \subseteq \mathrm{C}'(c)[\mathbf{fix}\, f.c/f] \cup \mathrm{C}'(\mathbf{fix}\, f.c)$$

The left hand side of the reduction yields

$$\mathrm{C}'(\mathbf{fix}\, f.c) = \mathrm{T}(\mathbf{fix}\, f.c) \cup \{\mathbf{fix}\, f.c\} \cup \mathrm{C}'(c)[\mathbf{fix}\, f.c/f]$$

and we conclude $\mathrm{C}'(c[\mathbf{fix}\, f.c/f]) \subseteq \mathrm{C}'(\mathbf{fix}\, f.c)$.

Case $\dfrac{c \Rightarrow c'}{c; d \Rightarrow c'; d}$ **:** Consider

$$\mathrm{C}'(c'; d) = \mathrm{T}(\tau) \cup \mathrm{T}(\tau') \cup \mathrm{C}'(c') \cup \mathrm{C}'(d)$$

Let $c_1; ..; c_n \in \mathrm{C}^*(c'; d)$ with $c_i \in \mathrm{C}'(c'; d)$. We show that $c_i \in \mathrm{C}^*(c; d)$ for all i:

> **Case** $c_i \in \mathrm{T}(\tau)$**:** Since $(c'; d)$ and $(c; d)$ are equally typed (Lemma 7.2.2) then $\mathrm{T}(\tau) \subseteq \mathrm{C}'(c; d) \subseteq \mathrm{C}^*(c; d)$.
>
> **Case** $c_i \in \mathrm{T}(\tau')$**:** As above.
>
> **Case** $c_i \in \mathrm{C}'(c')$**:** By Lemma 7.2.5 and IH
>
> $$\mathrm{C}'(c') \subseteq \mathrm{C}^*(c') \subseteq \mathrm{C}^*(c) \subseteq \mathrm{C}^*(c; d)$$
>
> since obviously $\mathrm{C}'(c) \subseteq \mathrm{C}'(c; d)$.
>
> **Case** $c_i \in \mathrm{C}'(d)$**:** Obvious.

Because $c_i \in \mathrm{C}^*(c; d)$ we can find $c_{i,j} \in \mathrm{C}'(c; d)$ such that $c_i = c_{i,1}; ..; c_{i,m_i}$ for all i. Consider

$$c_1; ..; c_n \equiv c_{1,1}; ..; c_{1,m_1}; ..; c_{n,1}; ..; c_{n,m_n}$$

which is cycle free, because $c_1; ..; c_n \in \mathrm{C}^*(c'; d)$. We thereby have that $c_1; ..; c_n \in \mathrm{C}^*(c; d)$ and hence that $\mathrm{C}^*(c'; d) \subset \mathrm{C}^*(c; d)$. ◇

We can now prove the main property of the algorithm which leads to the termination theorem.

Lemma 7.2.8 *Let*

$$(A_0, c_0, d_0), \ldots, (A_n, c_n, d_n), \ldots$$

be a call path of **E**. *For all* $i > 0$

$$\mathrm{C}^*(c_i) \cup \mathrm{C}^*(d_i) \subseteq \mathrm{C}^*(c_{i-1}) \cup \mathrm{C}^*(d_{i-1})$$

and $A_{i-1} \subseteq A_i$. *From this we also conclude for all* i

$$\mathrm{C}^*(c_i) \cup \mathrm{C}^*(d_i) \subseteq \mathrm{C}^*(c_0) \cup \mathrm{C}^*(d_0)$$

PROOF Let i be given. Case analysis of algorithm rules producing call i. Only interesting cases are shown.

Case Line 1: Obviously $A \subseteq A$ and furthermore

$$\begin{aligned}
&\mathrm{C}'(\mathbf{unfold}; \iota_{\tau_1[T/\alpha]} \to \iota_{\tau_2[T/\alpha]}; \mathbf{fold}) \\
&\quad = \mathrm{T}(T) \cup \mathrm{C}'_0(\mathbf{unfold}; \iota_{\tau_1[T/\alpha]} \to \iota_{\tau_2[T/\alpha]}; \mathbf{fold}) \\
&\quad = \mathrm{T}(T) \cup \mathrm{C}'(\mathbf{unfold}) \cup \mathrm{C}'(\iota_{\tau_1[T/\alpha]} \to \iota_{\tau_2[T/\alpha]}; \mathbf{fold}) \\
&\quad = \mathrm{T}(T) \cup \mathrm{T}((\tau_1 \to \tau_2)[T/\alpha]) \cup \{\mathbf{unfold}_T\} \cup \mathrm{C}'(\iota_{\tau_1[T/\alpha]} \to \iota_{\tau_2[T/\alpha]}) \cup \mathrm{C}'(\mathbf{fold}) \\
&\quad = \mathrm{T}(T) \cup \mathrm{T}(\tau_1[T/\alpha]) \cup \mathrm{T}(\tau_2[T/\alpha]) \cup \{\mathbf{fold}_T\} \\
&\quad = \mathrm{T}(T) \\
&\quad = \mathrm{C}'(\iota_T)
\end{aligned}$$

where $T = \mu\alpha.\tau_1 \to \tau_2$. By Lemma 7.2.5 (3) is $\mathrm{C}^*(\mathbf{unfold}; \iota_{\tau_1[T/\alpha]} \to \iota_{\tau_1[T/\alpha]}; \mathbf{fold}) \subseteq \mathrm{C}^*(\iota_T) \subseteq \mathrm{C}^*(\iota_T) \cup \mathrm{C}^*(c)$.

Case Line 3: Assumption set condition trivially satisfied. Consider

$$\mathrm{C}'(\iota_{\tau_1} \to \iota_{\tau_2}) = \mathrm{T}(\tau_1 \to \tau_2) \cup \mathrm{T}(\tau_1) \cup \mathrm{T}(\tau_2) = \mathrm{T}(\tau_1 \to \tau_2) = \mathrm{C}'(\iota_{\tau_1 \to \tau_2})$$

which proves case.

Case Line 5: Consider assumption set $A_{i-1} = A \subseteq A \cup \{c \to c' = d \to d'\} = A_i$. Since

$$\mathrm{C}'(c \to c') = \mathrm{T}(c \to c') \cup \mathrm{C}''(c) \cup \mathrm{C}''(c')$$

then by Lemma 7.2.5 (2)

$$\mathrm{C}'(c) \subseteq \mathrm{C}''(c) \subseteq \mathrm{C}'(c \to c')$$

and thus $\mathrm{C}^*(c) \subseteq \mathrm{C}^*(c \to c')$. Lemma 7.2.7 gives that $\mathrm{C}^*(\mathrm{nf}(c)) \subseteq \mathrm{C}^*(c)$. Likewise for c', d and d' so the two possible calls from line 5 both satisfy the closure requirement.

Case Line 8: Easily seen that $\mathrm{C}^*(c_1 \to c_2) \subseteq \mathrm{C}^*(\mathbf{unfold}; c_1 \to c_2)$ and $\mathrm{C}'(\iota_T) = \mathrm{T}(T) \subseteq \mathrm{C}'(\mathbf{unfold}_T)$.

The second result is obtained by simple induction on i. ◇

A final lemma concerning distribution of arrow calls during algorithm execution.

Lemma 7.2.9 *If $p = (A_0, c_0, d_0), \ldots$ is an infinite call path of* **E** *then*

$$\forall i \, \exists j > i \; : \; c_j = c \to c', \; d_j = d \to d'$$

The lemma states that in every infinite call path there exists infinitely many arrow calls.

PROOF All rules doing recursive calls, except line 5, introduce an arrow coercion in the recursive call. An infinite sequence of calls, where only one of the coercion arguments are processed, can not occur (inspect!) and since the only elimination rule for arrow coercions is line 5 we conclude that there are finitely many calls in between instances of line 5. In other words, line 5 is called infinitely often in an infinite call path. ◇

Theorem 7.2.10 *(Termination) For all assumption sets A and coercions $c, d \in C_{nf}^{-1}$ $\mathbf{E}(A, c, d)$ terminates.*

PROOF Assume that $\mathbf{E}(A, c, d)$ does *not* terminate. Two situations may occur: 1) the local processing of a rule is nonterminating (*i.e.* without doing recursive calls), 2) an infinite call-path p exists in the call tree.

Ad 1). The only rule of concern is at line 5 where the normal forms of four coercions are computed, but since $\Rightarrow$ is (SN) this can not be the case.

Ad 2). Assume p is an infinite call path and let A_i, $c_i = c' \to c''$ and $d_i = d' \to d''$ be a call of p (which exists by Lemma 7.2.9). If $(c_i = d_i) \in A_i$ then processing terminates which means that p is not infinite – contradicting assumption. If on the other hand $(c_i = d_i) \notin A_i$ then they are added and $\mathbf{E}$ called recursively. By proposition 7.2.8

$$\mathrm{C}^*(c' \to c'') \cup \mathrm{C}^*(d' \to d'') \subseteq \mathrm{C}^*(c) \cup \mathrm{C}^*(d)$$

By Definition 7.2.3 and Lemma 7.2.5

$$\mathrm{T}(c' \to c'') \cup \mathrm{C}''(c') \cup \mathrm{C}''(c'') = \mathrm{C}'(c' \to c'') \subseteq \mathrm{C}^*(c' \to c'')$$

and likewise for $d' \to d''$. Furthermore is $c' \in \mathrm{C}''(c')$ and $c'' \in \mathrm{C}''(c'')$ and similar for d', d''. Altogether we get

$$c', c'', d', d'' \in \mathrm{C}^*(c) \cup \mathrm{C}^*(d)$$

which by Proposition 7.2.4 is a finite set. It is thus only possible to construct finitely many *different* arrow pairs (assumptions). Proposition 7.2.8 states that the assumption sets in p increases monotonically and Lemma 7.2.9 gives that line 5 of $\mathbf{E}$ is called infinitely often. $\mathbf{E}$ will therefore eventually be invoked with a pair of coercions which already exists in the assumption set. Our initial assumption is proven false and the theorem true. ◇

Algorithm correctness. We prove that whenever algorithm $\mathbf{E}$ returns true the coercions are provable equal in our axiomatization and furthermore we prove that it always returns true for coercions encoding the same subtype judgement.

Lemma 7.2.11 *If $c, d \in C_{nf}^{-1}$ and $\mathbf{E}(A, c, d) = \textsc{True}$ then $A \vdash c = d$.*

PROOF We know that $\mathbf{E}$ terminates so we prove the lemma by induction on the number of recursive calls. Most cases correspond directly to rules in the equality axiomatization. We therefore only show some of the less obvious cases.

Case Line 1 : By rules (CE9,CE13,CE7) we get that $\iota_{\mu\alpha.\tau} = \mathbf{unfold}; \iota_{\tau[\mu\alpha.\tau]}; \mathbf{fold}$ and since the recursive types are canonical we know that $\tau = \tau_1 \to \tau_2$ and hence by (CE8) is $\iota_{\tau[\mu\alpha.\tau]} = \iota_{\tau_1[\mu\alpha.\tau]} \to \iota_{\tau_2[\mu\alpha.\tau]}$. Compatibility gives $A \vdash \iota_\tau = \mathbf{unfold}; \iota_{\tau_1[\mu\alpha.\tau]} \to \iota_{\tau_2[\mu\alpha.\tau]}; \mathbf{fold}$ and thus by induction hypothesis and transitivity $A \vdash \iota_\tau = d$.

Case Line 5 : If the coercions are part of the assumption set it follows directly from the hypothesis rule (CE14) that $A \vdash c = d$. Assume therefore that the two recursive calls both return $\textsc{True}$. By induction hypothesis we get

$$A(c \to c' = d \to d') \vdash \mathrm{nf}(c) = \mathrm{nf}(d)$$

$$A(c \to c' = d \to d') \vdash \mathrm{nf}(c') = \mathrm{nf}(d')$$

From Lemma 7.2.2 we know that $\vdash c = \mathrm{nf}(c)$ so by transitivity,

$$A(c \to c' = d \to d') \vdash c = d$$

$$A(c \to c' = d \to d') \vdash c' = d'$$

Rule (CE16) concludes case.

Case Line 8: From rule (CE7) and transitivity (CE16) we get that

$$\frac{A \vdash \mathbf{unfold}; (c \to c') = \mathbf{unfold}; \iota}{A \vdash \mathbf{unfold}; (c \to c') = \mathbf{unfold}}$$

Rule (CE13) and induction hypothesis $A \vdash c \to c' = \iota$ prove the premise of the judgement above and hence the case. ◇

Lemma 7.2.12 *For all assumption sets A and coercions $c, c' \in C_{nf}^{-1}$ we have*

$$\vdash c : \tau \leq \tau' \ \wedge \ \vdash c' : \tau \leq \tau' \quad \Rightarrow \quad \mathbf{E}(A, c, c') = \textsc{True}$$

PROOF Theorem 7.2.10 proves that $\mathbf{E}$ terminates with, say, altogether n recursive calls. We therefore prove lemma by induction on n.

Coercions in all sub calls have identical type signature and are expressible in the C_{nf}^{-1} grammar (inspect!) so the induction hypothesis is applicable at every recursive call. Lines 1–15 are proven directly by the induction hypothesis enabled by the mentioned observations. The only thing we need to show is that line 16 is never reached for any pair of coercions. To demonstrate this we consider all possible combinations of coercions.

C_{nf}^{-1}	
ι	d_1
$\bot$	d_2
$\top$	d_3
$c_1 \to c_2$	d_4
unfold	d_5
fold	d_6
unfold; $(c_1 \to c_2)$	d_7
$(c_1 \to c_2)$; **fold**	d_8
unfold; $(c_1 \to c_2)$; **fold**	d_9

c, c'	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9
d_1	15			3					1
d_2		15							
d_3			15						
d_4	4			5					
d_5					15		9		
d_6						15		12	
d_7					8		7		
d_8						11		10	
d_9	2								13

Field numbers correspond to algorithm lines handling that particular pair of coercions[1]. Empty fields mean that the pair of coercions cannot possibly have identical type signature and are thus not to be considered. From this table we conclude that all coercions with identical type signatures are handled before the exception rule at line 16. ◇

Theorem 7.2.13 *If* $\vdash c : \tau \leq \tau'$ *and* $\vdash c' : \tau \leq \tau'$ *then* $\vdash c = c'$.

PROOF By Lemma 7.2.2 we get that $\vdash c = \mathrm{nf}(c)$ and hence by Lemma 7.2.12 we finish proof. ◇

7.3 Soundness

We show that whenever two coercions are equal they have same type signature, which denote that they prove the same subtype judgement.

Theorem 7.3.1 *If* $\vdash c : \tau \leq \sigma$ *and* $\vdash c' : \tau' \leq \sigma'$ *then*

$$\forall A.\ A \vdash c = c' \quad \Rightarrow \quad \tau \equiv \tau' \text{ and } \sigma \equiv \sigma'$$

PROOF Induction on derivation of $A \vdash c = c'$. All cases except (CE14) and (CE15) are trivial.

Case (CE14) : We have that $(c = c') \in A$, but as noted in Section 7.1 the hypothesis rule has two implicit side conditions on the type-ability of c, c'. They state exactly what we require, namely that $\tau \equiv \tau'$ and $\sigma \equiv \sigma'$.

Case (CE15) : The rule says $A \vdash \mathbf{fix}\, f.c = c[\mathbf{fix}\, f.c/f]$. By Lemma 6.2.2 we know that

$$\vdash \mathbf{fix}\, f.c : \tau \leq \sigma \ \Leftrightarrow \vdash c[\mathbf{fix}\, f.c/f]) : \tau \leq \sigma$$

Since coercion typings are unique (Lemma 6.2.3) we may conclude case. ◇

[1] Some pairs might be handled by other lines as well depending on their type signatures

Chapter 8

Coercion Interpretation

We give a formal denotational semantics for coercions based on bc-domains. Besides giving a formal understanding of coercions the interpretation also gives us the opportunity to prove an interesting theorem about soundness of coercion equality. To motivate this theorem consider two coercions $E \vdash c : \tau \leq \sigma$, $E \vdash c' : \tau \leq \sigma$, that is $\vdash c = c'$. If $v : \tau$ we naturally have $c(v) : \sigma$ and $c'(v) : \sigma$ but are these values the same? We show that they are.

8.1 Denotational Semantics

We give an interpretation of types and coercions inspired by Gunter and Glynn [Gun92, Win93].

Definition 8.1.1 Types are interpreted as domains in the category of all domains, DOM.

$$\begin{aligned}
\mathrm{T}[\![\bot]\!]\,\rho &= \text{one point domain } \{\bot\} \\
\mathrm{T}[\![\top]\!]\,\rho &= \text{one point domain } \{\top\} \\
\mathrm{T}[\![\alpha]\!]\,\rho &= \rho(\alpha) \\
\mathrm{T}[\![\tau_1 \to \tau_2]\!]\,\rho &= \mathrm{T}[\![\tau_1]\!]\,\rho \xrightarrow{s} \mathrm{T}[\![\tau_2]\!]\,\rho \\
\mathrm{T}[\![\mu\alpha.\tau]\!]\,\rho &= T \text{ where } T \approx \mathrm{T}[\![\tau]\!]\,\rho(\alpha \mapsto T)
\end{aligned}$$

where ρ is an environment mapping type variables to domains. We denote the isomorphism $T \approx \mathrm{T}[\![\tau]\!]\,\rho(\alpha \mapsto T)$ with $(\mathrm{UF}_{\mu\alpha.\tau}, \mathrm{FD}_{\mu\alpha.\tau})$ where

$$\mathrm{UF}_{\mu\alpha.\tau} : T \xrightarrow{s} \mathrm{T}[\![\tau]\!]\,\rho(\alpha \mapsto T)$$

$$\mathrm{FD}_{\mu\alpha.\tau} : \mathrm{T}[\![\tau]\!]\,\rho(\alpha \mapsto T) \xrightarrow{s} T$$

For coercions we define

$$\mathrm{T}[\![(f_1 : \tau_1 \leq \sigma_1, \ldots, f_n : \tau_n \leq \sigma_n)]\!] = (\mathrm{T}[\![\tau_1]\!] \xrightarrow{s} \mathrm{T}[\![\sigma_1]\!]) \times \ldots \times (\mathrm{T}[\![\tau_n]\!] \xrightarrow{s} \mathrm{T}[\![\sigma_n]\!])$$

$$\mathrm{T}[\![E \vdash c : \tau \leq \sigma]\!] = \mathrm{T}[\![E]\!] \to \mathrm{T}[\![\tau]\!] \xrightarrow{s} \mathrm{T}[\![\sigma]\!]$$

$\diamond$

Proposition 8.1.2 $\mathrm{T}[\![\tau]\!]\ \rho$ *is well-defined for all* τ, ρ.

PROOF $\mathrm{T}[\![\tau]\!]\ \rho$ can be formulated as a set of recursive domain equations. Solutions to such systems exist and it is a well-known result in domain theory (see [Gun92, Win93]). ◇

Definition 8.1.3 Let $E \vdash c : \tau \leq \sigma$ be a type judgement. Assume $E = (f_1 : \tau_1 \leq \sigma_1, \ldots, f_n : \tau_n \leq \sigma_n)$ and define $\vec{E} = (f_1, \ldots, f_n)$. The interpretation of the type judgement is given by

$$\mathrm{C}[\![E \vdash \iota_\tau : \tau \leq \tau]\!] = \lambda\vec{E}.\,\lambda x.\,x$$

$$\mathrm{C}[\![E \vdash \bot_\tau : \bot \leq \tau]\!] = \lambda\vec{E}.\,\lambda x.\,\bot_{\mathrm{T}[\![\tau]\!]}$$

$$\mathrm{C}[\![E \vdash \top_\tau : \tau \leq \top]\!] = \lambda\vec{E}.\,\lambda x.\,\top$$

$$\mathrm{C}[\![E \vdash f_i : \tau_i \leq \sigma_i]\!] = \lambda\vec{E}.\,f_i$$

$$\mathrm{C}[\![E \vdash \mathbf{unfold}_{\mu\alpha.\tau} : \mu\alpha.\tau \leq \tau[\mu\alpha.\tau/\alpha]]\!] = \lambda\vec{E}.\,\mathrm{UF}_{\mu\alpha.\tau}$$

$$\mathrm{C}[\![E \vdash \mathbf{fold}_{\mu\alpha.\tau} : \tau[\mu\alpha.\tau/\alpha] \leq \mu\alpha.\tau]\!] = \lambda\vec{E}.\,\mathrm{FD}_{\mu\alpha.\tau}$$

$$\mathrm{C}[\![E \vdash (c; c') : \tau \leq \sigma]\!] = \lambda\vec{E}.\,\mathrm{C}[\![E \vdash c : \tau \leq \delta]\!]\ \vec{E}; \mathrm{C}[\![E \vdash c' : \delta \leq \sigma]\!]\ \vec{E}$$

$$\mathrm{C}[\![E \vdash (c \to c') : (\tau' \to \sigma) \leq (\tau \to \sigma')]\!] =$$

$$\lambda\vec{E}.\,\lambda x.\,(\mathrm{C}[\![E \vdash c : \tau \leq \tau']\!]\ \vec{E}; x; \mathrm{C}[\![E \vdash c' : \sigma \leq \sigma']\!]\ \vec{E})$$

$$\mathrm{C}[\![E \vdash \mathbf{fix}\ f.c \to c' : (\tau' \to \sigma) \leq (\tau \to \sigma')]\!] = \bigsqcup_i F^i(\bot)$$

$$F(v) = \lambda(f_1, .., f_n).\,\lambda x.\,\mathrm{C}[\![E' \vdash c : \tau \leq \tau']\!]\ (f_1, .., f_n, v); x; \mathrm{C}[\![E' \vdash c' : \sigma \leq \sigma']\!]\ (f_1, .., f_n, v)$$

$$E' = E(f : (\tau' \to \sigma) \leq (\tau \to \sigma'))$$

◇

Observe by inspection that $\mathrm{C}[\![E \vdash c : \tau \leq \sigma]\!]\ \vec{v} = \mathrm{C}[\![E' \vdash c : \tau \leq \sigma]\!]\ \vec{v'}$ where $E' \subseteq E$ and $\vec{v'} \subseteq \vec{v}$ as long as c still types under E'.

Proposition 8.1.4 *The interpreter* $\mathrm{C}[\![\cdot]\!]$ *is well-defined, that is for all type judgement* $E \vdash c : \tau \leq \sigma$ *there exists a unique* $\varphi \in \mathrm{T}[\![E \vdash c : \tau \leq \sigma]\!]$ *such that* $\mathrm{C}[\![E \vdash c : \tau \leq \sigma]\!] = \varphi$.

Note that $\mathrm{C}[\![E \vdash c : \tau \leq \sigma]\!]$ is a member of a strict, function domain and hence strict and continuous.

PROOF Induction on coercion c.

Case $c = \iota_\tau$**:** We trivially conclude that $\lambda\vec{E}.\,\lambda x.\,x \in \mathrm{T}[\![E]\!] \to (\mathrm{T}[\![\tau]\!] \xrightarrow{s} \mathrm{T}[\![\tau]\!])$. We have thus found a function in the right domain which clearly is the only one produced by $\mathrm{C}[\![E \vdash \iota_\tau : \tau \leq \tau]\!]$.

Case $c = \bot_\tau$**:** The interpretation is $\lambda\vec{E}.\,\lambda x.\,\bot$ which clearly maps between the right domains and it is obviously also continuous and strict. Uniqueness is evident.

Case $c = \top_\tau$**:** Similar to $\bot_\tau$, except for strictness. $\lambda x.\,\top$ is strict because $\bot_{\mathrm{T}[\![\top]\!]} = \top$.

Case $c = f_i$: Since c is typable we find that $f_i \in \mathrm{dom}(E)$ and $E(f_i) = (\tau_i \leq \sigma_i) = (\tau \leq \sigma)$. $\lambda\vec{E}.\, f_i$ is obviously a continuous function from $\mathrm{T}[\![E]\!]$ to $\mathrm{T}[\![\tau]\!] \to \mathrm{T}[\![\sigma]\!]$ and without doubt the only possible function returned by $\mathrm{C}[\![\,]\!]$. Since $\vec{E} \in \mathrm{T}[\![E]\!]$ we know that every function in it is strict and hence is $f_i \in \vec{E}$ strict.

Case $c =$ unfold or $c =$ fold: Trivial due to continuity and strictness of UF and FD.

Case $c = c'; c''$: We know that $E \vdash c' : \tau \leq \delta$ and $E \vdash c'' : \delta \leq \sigma$ for some δ because $E \vdash c : \tau \leq \sigma$. By induction hypothesis there exist unique functions

$$\varphi' \in \mathrm{T}[\![E \vdash c' : \tau \leq \delta]\!]$$

$$\varphi'' \in \mathrm{T}[\![E \vdash c'' : \delta \leq \sigma]\!]$$

such that $\mathrm{C}[\![E \vdash c' : \tau \leq \delta]\!] = \varphi'$ and $\mathrm{C}[\![E \vdash c'' : \delta \leq \sigma]\!] = \varphi''$. If $\vec{v} \in \mathrm{T}[\![E]\!]$ then $\varphi'(\vec{v}) \in (\mathrm{T}[\![\tau]\!] \to \mathrm{T}[\![\delta]\!])$ and $\varphi''(\vec{v}) \in (\mathrm{T}[\![\delta]\!] \to \mathrm{T}[\![\sigma]\!])$. The composition of two continuous functions is again continuous — $\varphi'(\vec{v}); \varphi''(\vec{v}) \in (\mathrm{T}[\![\tau]\!] \to \mathrm{T}[\![\sigma]\!])$. Joined with the results obtained above we conclude $\mathrm{C}[\![E \vdash c' : \tau \leq \delta]\!]\ \vec{v}; \mathrm{C}[\![E \vdash c'' : \delta \leq \sigma]\!]\ \vec{v} \in (\mathrm{T}[\![\tau]\!] \to \mathrm{T}[\![\sigma]\!])$ and finally that

$$\lambda\vec{E}.\, \mathrm{C}[\![E \vdash c' : \tau \leq \delta]\!]\ \vec{E}; \mathrm{C}[\![E \vdash c'' : \delta \leq \sigma]\!]\ \vec{E}$$

belongs to the required domain and it is uniquely determined by induction. Since $\mathrm{C}[\![E \vdash c' : \tau \leq \delta]\!]\ \vec{E}$ and $\mathrm{C}[\![E \vdash c'' : \delta \leq \tau]\!]\ \vec{E}$ are strict then their composition is strict.

Case $c = c' \to c''$: Let $\tau = \tau'' \to \sigma'$ and $\sigma = \tau' \to \sigma''$. By type-ability of c we conclude $E \vdash c' : \tau' \leq \tau''$ and $E \vdash c'' : \sigma' \leq \sigma''$ and furthermore by induction hypothesis that there exist unique

$$\varphi' \in \mathrm{T}[\![E \vdash c' : \tau' \leq \tau'']\!]$$

$$\varphi'' \in \mathrm{T}[\![E \vdash c'' : \sigma' \leq \sigma'']\!]$$

such that $\mathrm{C}[\![E \vdash c' : \tau' \leq \tau'']\!] = \varphi'$ and $\mathrm{C}[\![E \vdash c'' : \sigma' \leq \sigma'']\!] = \varphi''$. If $\vec{v} \in \mathrm{T}[\![E]\!]$ then $\varphi'(\vec{v}) \in \mathrm{T}[\![\tau']\!] \to \mathrm{T}[\![\tau'']\!]$ and $\varphi''(\vec{v}) \in \mathrm{T}[\![\sigma']\!] \to \mathrm{T}[\![\sigma'']\!]$. Now, say $x \in \mathrm{T}[\![\tau'']\!] \to \mathrm{T}[\![\sigma']\!]$ then

$$\varphi'(\vec{v}); x; \varphi''(\vec{v}) \in \mathrm{T}[\![\tau']\!] \to \mathrm{T}[\![\sigma'']\!]$$

and therefore

$$\lambda x.\, \varphi'(\vec{v}); x; \varphi''(\vec{v}) \in \mathrm{T}[\![\tau]\!] \to \mathrm{T}[\![\sigma]\!]$$

This function is by induction uniquely determined. We also have

$$\big(\lambda x.\, \varphi'(\vec{v}); x; \varphi''(\vec{v})\big)\bot = \varphi'(\vec{v}); \bot; \varphi''(\vec{v}) = \bot$$

where we used the induction hypothesis on c' to conclude $\varphi'(\vec{v})$ strict.

Case $c =$ fix $f.c' \to c''$: Here $\tau = \tau'' \to \sigma'$ and $\sigma = \tau' \to \sigma''$. By type-ability of c we know that $E(f : \tau \leq \sigma) \vdash c_1 : \tau' \leq \tau''$ and by induction we find unique function $\varphi' \in \mathrm{T}[\![E(f : \tau \leq \sigma) \vdash c_1 : \tau' \leq \tau'']\!]$ such that $\mathrm{C}[\![E(f : \tau \leq \sigma) \vdash c_1 : \tau' \leq \tau'']\!] = \varphi'$. Similarly we find unique φ'' such that

$\mathrm{C}[\![E(f:\tau\leq\sigma)\vdash c_2:\sigma'\leq\sigma'']\!] = \varphi''$. Assume $(v_1,\ldots,v_n)\in \mathrm{T}[\![E]\!]$ and $v\in \mathrm{T}[\![\tau]\!]\to\mathrm{T}[\![\sigma]\!]$ then $(v_1,\ldots,v_n,v)\in\mathrm{T}[\![E(f:\tau\leq\sigma)]\!]$ and hence

$$\lambda x.\,\varphi'(v_1,\ldots,v_n,v);x;\varphi''(v_1,\ldots,v_n,v)\in \mathrm{T}[\![\tau]\!]\xrightarrow{s}\mathrm{T}[\![\sigma]\!]$$

by similar arguments as the previous case. From this we learn that F is unique, strict and continuous; $F\in(\mathrm{T}[\![\tau]\!]\xrightarrow{s}\mathrm{T}[\![\sigma]\!])\to\left(\mathrm{T}[\![\tau]\!]\xrightarrow{s}\mathrm{T}[\![\sigma]\!]\right)$. The fix-point theorem of domain theory provides us with a unique fix-point computed by $\bigsqcup_i F^i(\bot)$ which is exactly the function produced by $\mathrm{C}[\![]\!]$. It is easily seen that $F^i(\bot)\vec{v}$ is strict for all i and $\vec{v}\in\mathrm{T}[\![E]\!]$ so we get

$$\Big(\bigsqcup_i F^i(\bot)\Big)\vec{v}\bot=\bigsqcup_i\big(F^i(\bot)\vec{v}\bot\big)=\bigsqcup_i\bot=\bot$$

We conclude that the produced function is strict. ◇

Lemma 8.1.5 ***(Substitution)*** *For $E(f:\tau'\leq\sigma')E'\vdash c:\tau\leq\sigma$ and $E''\vdash d:\tau'\leq\sigma'$,*

$$\mathrm{C}[\![E_1(f:\tau'\leq\sigma')E_2\vdash c:\tau\leq\sigma]\!]\ \vec{v_1}(f\mapsto\mathrm{C}[\![E_3\vdash d:\tau'\leq\sigma']\!]\ \vec{v_3})\vec{v_2}=$$
$$\mathrm{C}[\![E_1E_2\vdash c[d/f]:\tau\leq\sigma]\!]\ \vec{v_1}\vec{v_2}$$

where $\vec{v_1}\in\mathrm{T}[\![E_1]\!]$, $\vec{v_2}\in\mathrm{T}[\![E_2]\!]$, $\vec{v_3}\in\mathrm{T}[\![E_3]\!]$ and $E_3\subseteq E_1E_2$, $\vec{v_3}\subseteq\vec{v_1}\vec{v_2}$.

PROOF Induction on c. Let $E_4=E_1(f:\tau'\leq\sigma')E_2$, $\vec{v_4}=\vec{v_1}(f\mapsto\mathrm{C}[\![E_3\vdash d:\tau'\leq\sigma']\!]\ \vec{v_3})\vec{v_2}$.

Case $c=f$: We have

$$\mathrm{C}[\![E_4\vdash f:\tau\leq\sigma]\!]\ \vec{v_4}=\mathrm{C}[\![E_3\vdash d:\tau'\leq\sigma']\!]\ \vec{v_3}=\mathrm{C}[\![E_1E_2\vdash f[d/f]:\tau\leq\sigma]\!]\ \vec{v_1}\vec{v_2}$$

since $E_3\subseteq E_1E_2$ and $\vec{v_3}\subseteq\vec{v_1}\vec{v_2}$.

Case $c=g\neq f$:

$$\mathrm{C}[\![E_4\vdash g:\tau\leq\sigma]\!]\ \vec{v_4}=\vec{v_4}(g)=\vec{v_1}\vec{v_2}(g)=\mathrm{C}[\![E_1E_2\vdash g[d/f]:\tau\leq\sigma]\!]\ \vec{v_1}\vec{v_2}$$

Case $c=\mathbf{fix}\ g.c'\to c''$: Consider F:

$$F(v)\vec{v_4}=\lambda x.\,(\mathrm{C}[\![E_4(g)\vdash c':\tau'\leq\tau'']\!]\ (\vec{v_4}v));x;(\mathrm{C}[\![E_4(g)\vdash c'':\sigma'\leq\sigma'']\!]\ (\vec{v_4}v))$$

By induction hypothesis on c' we obtain

$$\mathrm{C}[\![E_4(g)\vdash c':\tau'\leq\tau'']\!]\ (\vec{v_4}v)=\mathrm{C}[\![E_1E_2(g)\vdash c'[d/f]:\tau'\leq\tau'']\!]\ (\vec{v_1}\vec{v_2}v)$$

and similar for c''. Consider

$$F'(v)(\vec{v_1}\vec{v_2})=\lambda x.\,(\mathrm{C}[\![E_1\mathbf{E}_2(g)\vdash c'[d/f]:\tau'\leq\tau'']\!]\ (\vec{E_1}\vec{E_2}v));x;(\mathrm{C}[\![E_1E_2(g)\vdash c''[d/f]:\sigma'\leq\sigma'']\!]\ (\vec{v_1}\vec{v_2}v))$$

such that $\bigsqcup_i F'^i(\bot)=\mathrm{C}[\![E_1E_2\vdash(\mathbf{fix}\ g.c'\to c'')[d/f]:\tau\leq\sigma]\!]\ \vec{v_1}\vec{v_2}$. We thus have $F(v)\vec{v_4}=F'(v)(\vec{v_1}\vec{v_2})$ by the induction hypothesis. We may now deduce

$$\mathrm{C}[\![E_4\vdash\mathbf{fix}\ g.c'\to c'':\tau\leq\sigma]\!]\ \vec{v_4}\quad=$$

$$
\begin{aligned}
(\bigsqcup_i F^i(\bot))\vec{v_4} &= \\
\bigsqcup_i (F^i(\bot)\vec{v_4}) &= \\
\bigsqcup_i (F'^i(\bot)(\vec{v_1}\vec{v_2})) &= \\
(\bigsqcup_i F'^i(\bot))(\vec{v_1}\vec{v_2}) &= \mathrm{C}[\![E_1 E_2 \vdash (\mathbf{fix}\ g.c' \to c'')[d/f] : \tau \leq \sigma]\!]\ \vec{v_1}\vec{v_2}
\end{aligned}
$$

Case $c = (c' \to c'')$, $(c'; c'')$**:** Simply apply induction hypothesis. ◇

8.2 Denotational Approximation Semantics

Before being able to prove the soundness theorem we need an inductively defined denotational semantics for coercions, because otherwise we have no foundation upon which to built the proof. We therefore construct a sequence of inductively defined approximations which converge toward the standard interpretation. The standard semantics does in principle unfold all fix coercions infinitely whereas the approximations limit the number of unfoldings.

The approximations are based on an algorithm that when given a coercion typing returns all the subterm coercions occurring in arrow constructs. Furthermore it builds a continuous function which takes denotations corresponding to the arrow subterms and produces the denotation of the entire coercion. The algorithm is presented in Figure 8.1.

The next two lemmas state some fundamental properties of the algorithm.

Lemma 8.2.1 *For coercion c and type judgement $E \vdash c : \tau \leq \sigma$ we have that $\mathcal{A}(E \vdash c : \tau \leq \sigma)$ terminates.*

PROOF The only problematic cases of $\mathcal{A}(\cdot)$ are composition $(c'; c'')$ and fix $(\mathbf{fix}\ f.c \to c')$. Termination of composition is easily accounted for because the argument size is reduced in the recursive calls. Fix is more complicated since it actually may increase the size of the argument used in the recursive call, but because the body of fix is $c \to c'$ we know that f only occurs (if at all) within an arrow coercion. Arrow coercions lead to immediate termination of $\mathcal{A}()$, so every fix coercion will at most be unfolded once and therefore does this case also terminate. ◇

Since we have now shown that algorithm $\mathcal{A}()$ always terminates we can do induction proofs on its execution, which formally means induction on the number of recursive calls issued during execution.

Lemma 8.2.2 *Let $E \vdash c : \tau \leq \sigma$ be a type judgement and $\mathcal{A}(E \vdash c : \tau \leq \sigma) = (\phi, (c_1, \ldots, c_k))$. The following holds*

1. *$E \vdash c_i : \tau_i \leq \sigma_i$ for some τ_i, σ_i.*
2. *$\phi \in \left(\mathrm{T}[\![E \vdash c_1 : \tau_1 \leq \sigma_1]\!] \times \ldots \times \mathrm{T}[\![E \vdash c_k : \tau_k \leq \sigma_k]\!]\right) \to \mathrm{T}[\![E \vdash c : \tau \leq \sigma]\!]$*
   Note that this implicitly means that ϕ is continuous.

PROOF

1: $\mathcal{A}(E \vdash \iota_{\tau\to\tau'} : (\tau \to \tau') \leq (\tau \to \tau')) = (\lambda(c_1, c_2).\ \lambda\vec{E}.\ \lambda x.\ (c_1\vec{E}); x; (c_2\vec{E})\ ,\ (\iota_\tau, \iota_{\tau'}))$

2: $\mathcal{A}(E \vdash \iota_\tau : \tau \leq \tau) = (\lambda().\ \lambda\vec{E}.\ \lambda x.\ x\ ,\ ())$

3: $\mathcal{A}(E \vdash \bot_\tau : \bot \leq \tau) = (\lambda().\ \lambda\vec{E}.\ \lambda x.\ \bot_{\mathrm{T}[\![\tau]\!]}\ ,\ ())$

4: $\mathcal{A}(E \vdash \top_\tau : \tau \leq \top) = (\lambda().\ \lambda\vec{E}.\ \lambda x.\ \top\ ,\ ())$

5: $\mathcal{A}(E \vdash c'; c'' : \tau \leq \sigma) =$

6: **let**

7: $(\phi', (c'_1, \ldots, c'_{k'})) = \mathcal{A}(E \vdash c' : \tau \leq \tau')$

8: $(\phi'', (c''_1, \ldots, c''_{k''})) = \mathcal{A}(E \vdash c'' : \tau' \leq \sigma)$

9: **in**

10: $\Big(\lambda(x_1, \ldots, x_{k'}, y_1, \ldots, y_{k''}).\ \lambda\vec{E}.\ \phi'(x_1, \ldots, x_{k'})\vec{E}; \phi''(y_1, \ldots, y_{k''})\vec{E}$

11: $, (c'_1, \ldots, c'_{k'}, c''_1, \ldots, c''_{k''})\Big)$

12: **end**

13: $\mathcal{A}(E \vdash c' \to c'' : \tau' \to \sigma \leq \tau \to \sigma') = \Big(\lambda(c', c'').\ \lambda\vec{E}.\ \lambda z.\ (c'\vec{E}); z; (c''\vec{E})\ ,\ (c', c'')\Big)$

14: $\mathcal{A}(E \vdash \mathbf{fix}\ f.c' : \tau \leq \sigma) = \mathcal{A}(E \vdash c'[\mathbf{fix}\ f.c'/f] : \tau \leq \sigma)$

15: $\mathcal{A}(E \vdash f_i : \tau \leq \sigma) = (\lambda().\ \lambda\vec{E}.\ f_i\ ,\ ())$

16: $\mathcal{A}(E \vdash \mathbf{unfold} : \mu\alpha.\tau \leq \tau[\mu\alpha.\tau/\alpha]) = (\lambda().\ \lambda\vec{E}.\ \mathrm{UF}\ ,\ ())$

17: $\mathcal{A}(E \vdash \mathbf{fold} : \tau[\mu\alpha.\tau/\alpha] \leq \mu\alpha.\tau) = (\lambda().\ \lambda\vec{E}.\ \mathrm{FD}\ ,\ ())$

Figure 8.1: Algorithm for computing arrow subcoercions.

Ad.1 Induction on the execution of $\mathcal{A}()$. Only interesting case is line 11 and it holds because $c' \to c''$ is typeable and hence are c', c'' also typeable.

Ad.2 By induction on the algorithm execution we show that the resulting function is a member of the required domain.

Case Ln.1,2,3,4,15,16,17: Trivial.

Case Ln.5: $c = c'; c''$. First, let us determine the nature of the function. By induction we find that

$$\phi' \in \Big(\mathrm{T}[\![E \vdash c'_1 : \tau'_1 \leq \sigma'_1]\!]\ \times \ldots \times \mathrm{T}[\![E \vdash c'_{k'} : \tau'_{k'} \leq \sigma'_{k'}]\!]\ \Big) \to \mathrm{T}[\![E \vdash c' : \tau \leq \tau']\!]$$

$$\phi'' \in \Big(\mathrm{T}[\![E \vdash c''_1 : \tau''_1 \leq \sigma''_1]\!] \times \ldots \times \mathrm{T}[\![E \vdash c''_{k''} : \tau''_{k''} \leq \sigma''_{k''}]\!]\ \Big) \to \mathrm{T}[\![E \vdash c'' : \tau' \leq \sigma]\!]$$

When we apply ϕ' on $(x_1, \ldots, x_{k'}) \in (\mathrm{T}[\![E \vdash c'_1 : \tau'_1 \leq \sigma'_1]\!] \times \ldots \times \mathrm{T}[\![E \vdash c'_{k'} : \tau'_{k'} \leq \sigma'_{k'}]\!]\ )$ and $\vec{v} \in \mathrm{T}[\![E]\!]$ we get a function in the domain $\mathrm{T}[\![\tau]\!] \xrightarrow{s} \mathrm{T}[\![\tau']\!]$. Similarly is $\phi''(y_1, \ldots, y_{k''})\vec{v}$ a member of $\mathrm{T}[\![\tau']\!] \xrightarrow{s} \mathrm{T}[\![\sigma]\!]$. Composition of these (strict and continuous) functions yield a (strict and continuous) function in the domain $\mathrm{T}[\![\tau]\!] \xrightarrow{s} \mathrm{T}[\![\sigma]\!]$. We conclude that ϕ maps between the required domains. Next, we show it continuous and hence in the function domain. Monotonic: Let $(x_1, \ldots, x_{k'}, y_1, \ldots, y_{k''}) \leq (w_1, \ldots, w_{k'}, z_1, \ldots, z_{k''})$, *i.e.* $x_1 \leq w_1$, $x_2 \leq w_2$ and so forth. Especially $(x_1, \ldots, x_{k'}) \leq (w_1, \ldots, w_{k'})$ and $(y_1, \ldots, y_{k''}) \leq$

$(z_1, \ldots, z_{k''})$.

$$\begin{aligned}\phi(x_1, \ldots, x_{k'}, y_1, \ldots, y_{k''})\vec{v} &= \phi'(x_1, \ldots, x_{k'})\vec{v}; \phi''(y_1, \ldots, y_{k''})\vec{v} \\ &\leq \phi'(w_1, \ldots, w_{k'})\vec{v}; \phi''(z_1, \ldots, z_{k''})\vec{v} \\ &= \phi(w_1, \ldots, w_{k'}, z_1, \ldots, z_{k''})\vec{v}\end{aligned}$$

Note application of induction hypothesis in second equation where we also used the well-known fact that composition is a continuous operator. To show ϕ limit preserving let $\vec{z}^i = (x_1^i, \ldots, x_{k'}^i, y_1^i, \ldots, y_{k''}^i) = (\vec{x}^i, \vec{y}^i)$ be a chain. For chains of vectors (tuples) we have

$$\bigsqcup_i \vec{z}^i = \left(\sqcup_i x_1^i, \ldots, \sqcup_i x_{k'}^i, \sqcup_i y_1^i, \ldots, \sqcup_i y_{k''}^i\right) = \left(\sqcup_i \vec{x}^i, \sqcup_i \vec{y}^i\right)$$

Let $\vec{v} \in \mathrm{T}[\![E]\!]$.

$$\begin{aligned}\phi(\sqcup_i \vec{z}^i)\vec{v} &= \phi\left(\left(\sqcup_i \vec{x}^i, \sqcup_i \vec{y}^i\right)\right)\vec{v} \\ &= \left(\phi'\left(\sqcup_i \vec{x}^i\right)\vec{v}\right); \left(\phi''\left(\sqcup_i \vec{y}^i\right)\vec{v}\right) \\ &= \left(\bigsqcup_i \phi'\left(\vec{x}^i\right)\vec{v}\right); \left(\bigsqcup_i \phi''\left(\vec{y}^i\right)\vec{v}\right) \\ &= \bigsqcup_i \left(\phi'(\vec{x}^i)\vec{v}; \phi''(\vec{y}^i)\vec{v}\right) \\ &= \bigsqcup_i \phi(\vec{z}^i)\vec{v} = \left(\bigsqcup_i \phi(\vec{z}^i)\right)\vec{v}\end{aligned}$$

Case Ln.13 : If $(c', c'') \in \mathrm{T}[\![E \vdash c' : \tau \leq \tau']\!] \times \mathrm{T}[\![E \vdash c'' : \sigma \leq \sigma']\!]$ then $c'(\vec{v}) \in \mathrm{T}[\![\tau]\!] \xrightarrow{s} \mathrm{T}[\![\tau']\!]$ and $c''(\vec{v}) \in \mathrm{T}[\![\sigma]\!] \xrightarrow{s} \mathrm{T}[\![\sigma']\!]$. If further more $z \in \mathrm{T}[\![\tau']\!] \xrightarrow{s} \mathrm{T}[\![\sigma]\!]$ then the composition $(c'\vec{v}); z; (c''\vec{v})$ is a member of $\mathrm{T}[\![\tau]\!] \xrightarrow{s} \mathrm{T}[\![\sigma]\!]$. From these observations we easily deduce that ϕ has the right type. Monotonic: let $(x_1, x_2) \leq (y_1, y_2)$. By continuity of ; is $((x_1\vec{v}); z; (x_2\vec{v})) \leq ((y_1\vec{v}); z; (y_2\vec{v}))$ and since z, $\vec{v}$ are arbitrary we get $\phi(x_1, x_2) \leq \phi(y_1, y_2)$. Analogously is $\phi\left(\sqcup_i (x_1^i, x_2^i)\right) = \sqcup_i \phi\left((x_1^i, x_2^i)\right)$ for a chain (x_1^i, x_2^i).

Case Ln.14 : Follows immediately from induction hypothesis. ◇

Below we prove the intuition intended of the algorithm.

Lemma 8.2.3 *If $E \vdash c : \tau \leq \sigma$ and $\mathcal{A}(E \vdash c : \tau \leq \sigma) = (\phi, (c_1, \ldots, c_k))$ then*

$$\phi(\mathrm{C}[\![c_1]\!], \ldots, \mathrm{C}[\![c_k]\!]) = \mathrm{C}[\![c]\!]$$

PROOF Induction on execution of $\mathcal{A}(E \vdash c : \tau \leq \sigma)$.

Case Ln.1 : Here $c = \iota_{\tau \to \tau'}$, $c_1 = \iota_\tau$, $c_2 = \iota_{\tau'}$ and ϕ is

$$\phi(\mathrm{C}[\![E \vdash \iota_\tau]\!], \mathrm{C}[\![E \vdash \iota_{\tau'}]\!]) = \lambda\vec{E}.\, \lambda x.\, (\mathrm{C}[\![E \vdash \iota_\tau]\!]\ \vec{E}); x; (\mathrm{C}[\![E \vdash \iota_{\tau'}]\!]\ \vec{E})$$

We have that $\mathrm{C}[\![E \vdash \iota_\tau]\!] = \lambda\vec{E}.\, \lambda z.\, z$ so

$$\begin{aligned}\phi(\mathrm{C}[\![E \vdash \iota_\tau]\!], \mathrm{C}[\![E \vdash \iota_{\tau'}]\!]) &= \lambda\vec{E}.\, \lambda x.\, (\lambda z.\, z); x; (\lambda z.\, z) \\ &= \lambda\vec{E}.\, \lambda x.\, x \\ &= \mathrm{C}[\![E \vdash \iota_{\tau \to \tau'}]\!]\end{aligned}$$

Case Ln.2 : We have $\phi() = \lambda\vec{E}.\,\lambda x.\,x = \mathrm{C}[\![\iota_\tau]\!]$.

Case Ln.3,4 : Similar to ln.2.

Case Ln.5 : Here $c = c'; c''$ and by induction hypothesis we conclude

$$\begin{aligned}\phi'(\mathrm{C}[\![E \vdash c'_1]\!], \ldots, \mathrm{C}[\![E \vdash c'_{k'}]\!]) &= \mathrm{C}[\![E \vdash c' : \tau \le \tau']\!]\\ \phi''(\mathrm{C}[\![E \vdash c''_1]\!], \ldots, \mathrm{C}[\![E \vdash c''_{k''}]\!]) &= \mathrm{C}[\![E \vdash c'' : \tau' \le \sigma]\!]\end{aligned}$$

The result follows immediately from these equations

$$\begin{aligned}\phi(\mathrm{C}[\![E \vdash c'_1]\!], \ldots, \mathrm{C}[\![E \vdash c'_{k'}]\!], \mathrm{C}[\![E \vdash c''_1]\!], \ldots, \mathrm{C}[\![E \vdash c''_{k''}]\!]) &=\\ \lambda\vec{E}.\,\phi'(\mathrm{C}[\![E \vdash c'_1]\!], \ldots, \mathrm{C}[\![E \vdash c'_{k'}]\!])\vec{E}; \phi''(\mathrm{C}[\![E \vdash c''_1]\!], \ldots, \mathrm{C}[\![E \vdash c''_{k''}]\!])\vec{E} &=\\ \lambda\vec{E}.\,(\mathrm{C}[\![E \vdash c' : \tau \le \tau']\!]\ \vec{E}); (\mathrm{C}[\![E \vdash c'' : \tau' \le \sigma]\!]\ \vec{E}) &=\\ \mathrm{C}[\![E \vdash c'; c'' : \tau \le \sigma]\!]\end{aligned}$$

Case Ln.13 : Now $c = c' \to c''$.

$$\begin{aligned}&\phi(\mathrm{C}[\![E \vdash c' : \tau \le \tau']\!], \mathrm{C}[\![E \vdash c'' : \sigma \le \sigma']\!])\\ &= \lambda\vec{E}.\,\lambda z.\,(\mathrm{C}[\![E \vdash c' : \tau \le \tau']\!]\ \vec{E}); z; (\mathrm{C}[\![E \vdash c'' : \sigma \le \sigma']\!]\ \vec{E})\\ &= \mathrm{C}[\![E \vdash c' \to c'' : (\tau' \to \sigma) \le (\tau \to \sigma')]\!]\end{aligned}$$

Case Ln.14 : In this case $c = \mathbf{fix}\ f.c'$ and $\mathcal{A}(c'[\mathbf{fix}\ f.c'/f]) = (\phi, (c_1, \ldots, c_k))$. Consider the definition of $\mathrm{C}[\![\,]\!]$.

$$\mathrm{C}[\![\mathbf{fix}\ f.c']\!] = \bigsqcup_i F^i(\bot)$$

with $F(v) = \lambda\vec{E}.\,\mathrm{C}[\![E(f : \tau \le \sigma) \vdash c' : \tau \le \sigma]\!]\ \vec{E}(f \mapsto v)$, *i.e.* it is a fix-point of F. Let $\vec{v} \in \mathrm{T}[\![E]\!]$.

$$\begin{aligned}&\phi(\mathrm{C}[\![E \vdash c_1 : \tau_1 \le \sigma_1]\!], \ldots, \mathrm{C}[\![E \vdash c_k : \tau_k \le \sigma_k]\!])\vec{v}\\ &= \mathrm{C}[\![E \vdash c'[\mathbf{fix}\ f.c'/f] : \tau \le \sigma]\!]\ \vec{v}\\ &= \mathrm{C}[\![E(f : \tau \le \sigma) \vdash c' : \tau \le \sigma]\!]\ \vec{v}(f \mapsto \mathrm{C}[\![\mathbf{fix}\ f.c']\!]\ \vec{v})\\ &= F(\mathrm{C}[\![E \vdash \mathbf{fix}\ f.c' : \tau \le \sigma]\!]\ \vec{v})\\ &= \mathrm{C}[\![E \vdash \mathbf{fix}\ f.c' : \tau \le \sigma]\!]\ \vec{v}\end{aligned}$$

where we concluded the first equation by induction hypothesis; second by Substitution lemma (Lemma 8.1.5); third by definition of F and finally the fourth by fix-point property of $\mathrm{C}[\![E \vdash c : \tau \le \sigma]\!]$.

Case Ln.15,16,17 : Similar to ln.2. ◇

Definition 8.2.4 For type judgements $E \vdash c : \tau \le \sigma$ we define for all $n \ge 0$ an approximated interpretation

$$\begin{aligned}\mathrm{A}^0[\![E \vdash c : \tau \le \sigma]\!] &= \bot\\ \mathrm{A}^{(n+1)}[\![E \vdash c : \tau \le \sigma]\!] &= \phi(\mathrm{A}^n[\![E \vdash c_1 : \tau_1 \le \sigma_1]\!], \ldots, \mathrm{A}^n[\![E \vdash c_k : \tau_k \le \sigma_k]\!])\end{aligned}$$

where $(\phi, (c_1, \ldots, c_k)) = \mathcal{A}(E \vdash c : \tau \le \sigma)$ and $E \vdash c_i : \tau_i \le \sigma_i$. ◇

We see that $\mathrm{A}^n[\![]\!]$ has the desired functionality – it only interprets arrow constructs to the depth of n and cuts off all other by interpreting them as $\perp$. Observe by inspection that $\mathrm{A}^n[\![E \vdash c : \tau \leq \sigma]\!]\ \vec{v} = \mathrm{A}^n[\![E' \vdash c : \tau \leq \sigma]\!]\ \vec{v'}$ where $E' \subseteq E$ and $v' \subseteq v$ as long as c still types under E'.

We now focus on proving that the approximations converge toward the standard interpretation.

Lemma 8.2.5 $\mathrm{A}^n[\![E \vdash c : \tau \leq \sigma]\!] \in \mathrm{T}[\![E \vdash c : \tau \leq \sigma]\!]$ *are well-defined for all* $n \geq 0$ *and they constitute a chain,*

$$\mathrm{A}^0[\![E \vdash c : \tau \leq \sigma]\!] \leq \ldots \leq \mathrm{A}^n[\![E \vdash c : \tau \leq \sigma]\!] \leq \ldots$$

PROOF The approximations are inductively defined and since $\mathcal{A}()$ produce well-defined, strict and continuous functions ϕ we conclude that the approximations are well-defined. We prove $\mathrm{A}^n[\![E \vdash c : \tau \leq \sigma]\!]$ a chain by induction on n. Base case: Trivial since $\mathrm{A}^0[\![E \vdash c : \tau \leq \sigma]\!] = \perp$. Inductive case: We need to show $\mathrm{A}^n[\![E \vdash c : \tau \leq \sigma]\!] \leq \mathrm{A}^{(n+1)}[\![E \vdash c : \tau \leq \sigma]\!]$, *i.e.*

$$\begin{array}{c}\phi(\mathrm{A}^{(n-1)}[\![E \vdash c_1 : \tau_1 \leq \sigma_1]\!], \ldots, \mathrm{A}^{(n-1)}[\![E \vdash c_k : \tau_k \leq \sigma_k]\!]) \leq \\ \phi(\mathrm{A}^n[\![E \vdash c_1 : \tau_1 \leq \sigma_1]\!], \ldots, \mathrm{A}^n[\![E \vdash c_k : \tau_k \leq \sigma_k]\!])\end{array}$$

but by induction we know $\mathrm{A}^{(n-1)}[\![E \vdash c_i : \tau_i \leq \sigma_i]\!] \leq \mathrm{A}^n[\![E \vdash c_i : \tau_i \leq \sigma_i]\!]$ for $1 \leq i \leq k$ and by continuity of ϕ we arrive at the above in-equation. $\diamond$

In domain theory every chain has a least upper bound — the approximations thus have an upper limit,

$$\bigsqcup_n \mathrm{A}^n[\![E \vdash c : \tau \leq \sigma]\!]$$

In the sequent we prove that this limit equals the standard interpretation $\mathrm{C}[\![E \vdash c : \tau \leq \sigma]\!]$, but first a few technical lemmas about the approximations.

Lemma 8.2.6 ***(Substitution)*** *Let* $E'' = E(f : \tau' \leq \sigma')E'$ *be an environment such that* $E'' \vdash c : \tau \leq \sigma$, $EE' \vdash d : \tau' \leq \sigma'$. *If* $\mathcal{A}(E'' \vdash c : \tau \leq \sigma) = (\phi, (c_1, \ldots, c_k))$ *then*

1. $\mathrm{A}^{(n+1)}[\![E'' \vdash c[d/f] : \tau \leq \sigma]\!]\ \vec{v''}$
   $\leq \phi\Big(\mathrm{A}^n[\![E'' \vdash c_1[d/f] : \tau_1 \leq \sigma_1]\!], \ldots, \mathrm{A}^n[\![E'' \vdash c_k[d/f] : \tau_k \leq \sigma_k]\!]\Big)\vec{v''}$
2. $\bigsqcup_n \mathrm{A}^n[\![E'' \vdash c[d/f] : \tau \leq \sigma]\!]\ \vec{v''}$
   $= \bigsqcup_n \phi\Big(\mathrm{A}^n[\![E'' \vdash c_1[d/f] : \tau_1 \leq \sigma_1]\!], \ldots, \mathrm{A}^n[\![E'' \vdash c_k[d/f] : \tau_k \leq \sigma_k]\!]\Big)\vec{v''}$
3. $\bigsqcup_n \mathrm{A}^n[\![E'' \vdash c : \tau \leq \sigma]\!]\ \vec{v''}$
   $= \bigsqcup_n \mathrm{A}^n[\![E'' \vdash c[d/f] : \tau \leq \sigma]\!]\ \vec{v''}$

where $\vec{v} \in \mathrm{T}[\![E]\!]$, $\vec{v'} \in \mathrm{T}[\![E']\!]$ *and* $\vec{v''} = \vec{v}(f \mapsto \sqcup_{n'} \mathrm{A}^{n'}[\![EE' \vdash d : \tau' \leq \sigma']\!]\ v\vec{v'})\vec{v'} \in \mathrm{T}[\![E'']\!]$.

PROOF First, we have to prove that all the judgements make sense. By Lemma 6.2.2 does $EE' \vdash c[d/f] : \tau \leq \sigma$ hold and since $f \notin \mathrm{fv}(c[d/f])$ and

$EE' \subseteq E''$ we get by Lemma 6.2.1 that $E'' \vdash c[d/f] : \tau \leq \sigma$ holds. Analogous arguments justify the other judgements as well.

We prove the first equation by induction on the computation $\mathcal{A}(c)$.

Case Ln.1,2,3,4,16,17: Trivial since the coercions do not contain any variables.

Case Ln.5: Let $c = c'; c''$. By induction hypothesis $\mathrm{A}^{(n+1)}[\![E'' \vdash c'[d/f] : \tau \leq \tau']\!]\ \vec{v''} \leq \phi'(\mathrm{A}^n[\![E'' \vdash c'_1[d/f] : \tau'_1 \leq \sigma'_1]\!], \ldots, \mathrm{A}^n[\![E'' \vdash c'_{k'}[d/f] : \tau'_{k'} \leq \sigma'_{k'}]\!])\vec{v''}$ and similar for ϕ'' and c''.

$$\begin{aligned}
&\mathrm{A}^{(n+1)}[\![E'' \vdash (c'; c'')[d/f] : \tau \leq \sigma]\!]\ \vec{v''} \\
&\quad = \mathrm{A}^{(n+1)}[\![E'' \vdash c'[d/f] : \tau \leq \tau']\!]\ \vec{v''}; \mathrm{A}^{(n+1)}[\![E'' \vdash c''[d/f] : \tau' \leq \sigma]\!]\ \vec{v''} \\
&\quad \leq \phi'(\{\mathrm{A}^n[\![E'' \vdash c_i[d/f] : \tau'_i \leq \sigma'_i]\!]\ \})\vec{v''}; \phi''(\{E'' \vdash \mathrm{A}^n[\![c_j[d/f] : \tau''_j \leq \sigma''_j]\!]\ \})\vec{v''} \\
&\quad = \phi(\{\mathrm{A}^n[\![E'' \vdash c'_i[d/f] : \tau'_i \leq \sigma'_i]\!]\ \}, \{\mathrm{A}^n[\![E'' \vdash c''_j[d/f] : \tau''_j \leq \sigma''_j]\!]\ \})\vec{v''}
\end{aligned}$$

by induction and continuity of ;.

Case Ln.13: We directly obtain

$$\begin{aligned}
&\mathrm{A}^{(n+1)}[\![E'' \vdash (c' \to c'')[d/f] : (\tau \to \tau') \leq (\sigma \to \sigma')]\!]\ \vec{v''} \\
&\quad = \lambda z.\, \mathrm{A}^n[\![E'' \vdash c'[d/f] : \tau' \leq \sigma]\!]\ \vec{v''}; z; \mathrm{A}^n[\![E'' \vdash c''[d/f] : \tau \leq \sigma']\!]\ \vec{v''} \\
&\quad = \phi(\mathrm{A}^n[\![E'' \vdash c'[d/f] : \tau' \leq \sigma]\!], \mathrm{A}^n[\![E'' \vdash c''[d/f] : \tau \leq \sigma']\!])\vec{v''}
\end{aligned}$$

Case Ln.14: From induction hypothesis and the simple observation $c[d/f][(e[d/f])/g] = c[e/g][d/f]$, $g \notin \mathrm{fv}(d)$ we get

$$\begin{aligned}
&\mathrm{A}^{(n+1)}[\![E'' \vdash (\mathbf{fix}\ g.c')[d/f] : \tau \leq \sigma]\!]\ \vec{v''} \\
&\quad = \mathrm{A}^{(n+1)}[\![E'' \vdash \mathbf{fix}\ g.(c'[d/f]) : \tau \leq \sigma]\!]\ \vec{v''} \\
&\quad = \mathrm{A}^{(n+1)}[\![E'' \vdash (c'[d/f])[\mathbf{fix}\ g.(c'[d/f])/g] : \tau \leq \sigma]\!]\ \vec{v''} \\
&\quad = \mathrm{A}^{(n+1)}[\![E'' \vdash (c'[\mathbf{fix}\ g.c'])[d/f] : \tau \leq \sigma]\!]\ \vec{v''} \\
&\quad \leq \phi(\mathrm{A}^n[\![E'' \vdash c_1[d/f] : \tau \leq \sigma]\!], \ldots, \mathrm{A}^n[\![E'' \vdash c_k[d/f] : \tau \leq \sigma]\!])\vec{v''}
\end{aligned}$$

Case Ln.15: If $c = g$ we easily get $\mathrm{A}^{(n+1)}[\![E'' \vdash g[d/f] : \tau \leq \sigma]\!]\ \vec{v''} = \phi()\vec{v''}$. Assume therefore $c = f$,

$$\begin{aligned}
&\mathrm{A}^{(n+1)}[\![E'' \vdash f[d/f] : \tau' \leq \sigma']\!]\ \vec{v''} \\
&\quad = \mathrm{A}^{(n+1)}[\![E'' \vdash d : \tau' \leq \sigma']\!]\ \vec{v''} \\
&\quad \leq \bigsqcup_{n'} \mathrm{A}^{n'}[\![E'' \vdash d : \tau' \leq \sigma']\!]\ \vec{v''} \\
&\quad = \bigsqcup_{n'} \mathrm{A}^{n'}[\![EE' \vdash d : \tau' \leq \sigma']\!]\ v\vec{v'} \\
&\quad = \vec{v''}(f) \\
&\quad = \phi()\vec{v''}
\end{aligned}$$

We have now proved equation 1, which implies $\bigsqcup_n \mathrm{A}^n[\![E'' \vdash c[d/f] : \tau \leq \sigma]\!]\ \vec{v''} \leq \bigsqcup_n \phi\Big(\{\mathrm{A}^n[\![E'' \vdash c_i[d/f] : \tau_i \leq \sigma_i]\!]\ \}\Big)\vec{v''}$

To prove equation 2 we thus only need to show the opposite direction. We prove for all n that

$$\phi\Big(\{\mathrm{A}^n[\![E'' \vdash c_i[d/f] : \tau_i \leq \sigma_i]\!]\ \}\Big)\vec{v''} \leq \bigsqcup_{n'} \mathrm{A}^{n'}[\![E'' \vdash c[d/f] : \tau \leq \sigma]\!]\ \vec{v''}$$

and hence that the limit of the left hand side is less than the right hand side. Base case $n = 0$ is obvious and the inductive case goes, again, by induction on the computation of $\mathcal{A}(E'' \vdash c : \tau \leq \sigma)$.

Case Ln.1,2,3,4,16,17 : Trivial.

Case Ln.5 : Easy application of induction hypothesis.

Case Ln.13 : For $c = c' \to c''$ with type $(\tau' \to \sigma) \leq (\tau \to \sigma')$ we find

$$\begin{aligned}
&\phi(\mathrm{A}^n[\![E'' \vdash c'[d/f] : \tau \leq \tau']\!]\ , \mathrm{A}^n[\![E'' \vdash c''[d/f] : \sigma \leq \sigma']\!]\ )\vec{v''} \\
&\quad = \quad \lambda z.\ \mathrm{A}^n[\![E'' \vdash c'[d/f] : \tau \leq \tau']\!]\ \vec{v''}; z; \mathrm{A}^n[\![E'' \vdash c''[d/f] : \sigma \leq \sigma']\!]\ \vec{v''} \\
&\quad = \quad \mathrm{A}^{(n+1)}[\![E'' \vdash (c' \to c'')[d/f] : (\tau' \to \sigma) \leq (\tau \to \sigma')]\!]\ \vec{v''} \\
&\quad \leq \quad \bigsqcup_{n'} \mathrm{A}^{n'}[\![E'' \vdash (c' \to c'')[d/f] : (\tau' \to \sigma) \leq (\tau \to \sigma')]\!]\ \vec{v''}
\end{aligned}$$

Case Ln.14 : Let $c = \mathbf{fix}\ g.c'$. Induction and the previous observation on substitution orders gives

$$\begin{aligned}
&\phi(\mathrm{A}^n[\![E'' \vdash c_1[d/f] : \tau_1 \leq \sigma_1]\!]\ , \ldots, \mathrm{A}^n[\![E'' \vdash c_k[d/f] : \tau_k \leq \sigma_k]\!]\ )\vec{v''} \\
&\quad \leq \quad \bigsqcup_{n'} \mathrm{A}^{n'}[\![E'' \vdash c'[\mathbf{fix}\ g.c'/g][d/f] : \tau \leq \sigma]\!]\ \vec{v''} \\
&\quad = \quad \bigsqcup_{n'} \mathrm{A}^{n'}[\![E'' \vdash c'[d/f][(\mathbf{fix}\ g.c')[d/f]/g] : \tau \leq \sigma]\!]\ \vec{v''} \\
&\quad = \quad \bigsqcup_{n'} \mathrm{A}^{n'}[\![E'' \vdash (\mathbf{fix}\ g.c')[d/f] : \tau \leq \sigma]\!]\ \vec{v''}
\end{aligned}$$

Case Ln.15 : If $c = f$ then $\tau = \tau'$, $\sigma = \sigma'$ and $\phi()\vec{v''} = (\lambda\vec{E''}.\ f)\vec{v''} = \bigsqcup_{n'} \mathrm{A}^{n'}[\![E'' \vdash d : \tau' \leq \sigma']\!]\ \vec{v''} = \bigsqcup_{n'} \mathrm{A}^{n'}[\![E'' \vdash f[d/f] : \tau' \leq \sigma']\!]\ \vec{v''}$. Otherwise if $c = g \neq f$ then $\phi()\vec{v''} = (\lambda\vec{E''}.\ g)\vec{v''} = \bigsqcup_{n'} \mathrm{A}^{n'}[\![E'' \vdash g[d/f] : \tau \leq \sigma]\!]\ \vec{v''}$.

To prove the final equation we need the following property: If $\xi_i, \psi_i \in \mathrm{T}[\![E'' \vdash c_i : \tau_i \leq \sigma_i]\!]$ and $\xi_i(\vec{v''}) = \psi_i(\vec{v''})$ for $1 \leq i \leq k$ then $\phi(\{\xi_i\})\vec{v''} = \phi(\{\psi_i\})\vec{v''}$ This property is trivially shown by induction on $\mathcal{A}(E \vdash c : \tau \leq \sigma)$. We prove by induction on n the following inequalities,

$$\mathrm{A}^n[\![E'' \vdash c : \tau \leq \sigma]\!]\ \vec{v''} \quad \leq \quad \bigsqcup_{n'} \mathrm{A}^{n'}[\![E'' \vdash c[d/f] : \tau \leq \sigma]\!]\ \vec{v''} \tag{8.1}$$

$$\mathrm{A}^n[\![E'' \vdash c : \tau \leq \sigma]\!]\ \vec{v''} \quad \geq \quad \bigsqcup_{n'} \mathrm{A}^{n'}[\![E'' \vdash c[d/f] : \tau \leq \sigma]\!]\ \vec{v''} \tag{8.2}$$

Since they are proved by the exact same schema we only show the first of them in detail.

Base case trivial, because $\bot$ is least element of domain. Inductive case:

$$\begin{aligned}
&\mathrm{A}^{(n+1)}[\![E'' \vdash c : \tau \leq \sigma]\!]\ \vec{v''} \\
&= \phi(\mathrm{A}^n[\![E'' \vdash c_1 : \tau_1 \leq \sigma_1]\!], \ldots, \mathrm{A}^n[\![E'' \vdash c_k : \tau_k \leq \sigma_k]\!])\vec{v''} \\
&\leq \phi(\bigsqcup_{n_1} \mathrm{A}^{n_1}[\![E'' \vdash c_1[d/f] : \tau_1 \leq \sigma_1]\!], \ldots, \bigsqcup_{n_k} \mathrm{A}^{n_k}[\![E'' \vdash c_k[d/f] : \tau_k \leq \sigma_k]\!])\vec{v''} \\
&= \bigsqcup_{n_1} \ldots \bigsqcup_{n_k} \phi(\mathrm{A}^{n_1}[\![E'' \vdash c_1[d/f] : \tau_1 \leq \sigma_1]\!], \ldots, \mathrm{A}^{n_k}[\![E'' \vdash c_k[d/f] : \tau_k \leq \sigma_k]\!])\vec{v''} \\
&= \bigsqcup_{n'} \phi(\mathrm{A}^{n'}[\![E'' \vdash c_1[d/f] : \tau_1 \leq \sigma_1]\!], \ldots, \mathrm{A}^{n'}[\![E'' \vdash c_k[d/f] : \tau_k \leq \sigma_k]\!])\vec{v''} \\
&= \bigsqcup_{n'} \mathrm{A}^{n'}[\![E'' \vdash c[d/f] : \tau \leq \sigma]\!]\ \vec{v''}
\end{aligned}$$

where we first applied induction hypothesis together with the property stated above, then continuity of ϕ (Lemma 8.2.2) and finally equation 2 of this lemma. $\diamond$

Proposition 8.2.7 *For $E \vdash c : \tau \leq \sigma$ is*

$$\mathrm{C}[\![E \vdash c : \tau \leq \sigma]\!] = \bigsqcup_n \mathrm{A}^n[\![E \vdash c : \tau \leq \sigma]\!]$$

PROOF First we prove for all n (by induction) that $\mathrm{A}^n[\![E \vdash c : \tau \leq \sigma]\!] \leq \mathrm{C}[\![E \vdash c : \tau \leq \sigma]\!]$. Base case $n = 0$ trivial. Inductive case $n > 0$. By definition is

$$\mathrm{A}^{(n+1)}[\![E \vdash c : \tau \leq \sigma]\!] = \phi(\mathrm{A}^n[\![E \vdash c_1 : \tau_1 \leq \sigma_1]\!], \ldots, \mathrm{A}^n[\![E \vdash c_k : \tau_k \leq \sigma_k]\!])$$

for some c_i and τ_i, σ_i. Induction hypothesis thus gives $\mathrm{A}^n[\![E \vdash c_i : \tau_i \leq \sigma_i]\!] \leq \mathrm{C}[\![E \vdash c_i \tau_i \leq \sigma_i]\!]$ for $1 \leq i \leq k$ and hence by continuity of ϕ

$$\begin{aligned}
&\phi(\mathrm{A}^n[\![E \vdash c_1 : \tau_1 \leq \sigma_1]\!], \ldots, \mathrm{A}^n[\![E \vdash c_k : \tau_k \leq \sigma_k]\!]) \\
&\quad \leq \phi(\mathrm{C}[\![E \vdash c_1 : \tau_1 \leq \sigma_1]\!], \ldots, \mathrm{C}[\![E \vdash c_k : \tau_k \leq \sigma_k]\!]) \\
&\quad = \mathrm{C}[\![E \vdash c : \tau \leq \sigma]\!]
\end{aligned}$$

Last equation holds because of Lemma 8.2.2.

Next we prove $\mathrm{C}[\![E \vdash c : \tau \leq \sigma]\!] \leq \sqcup_n \mathrm{A}^n[\![E \vdash c : \tau \leq \sigma]\!]$ by induction on c.

Case $c = \iota_\tau$, $c = \bot_\tau$, $c = \top_\tau$, $c =$ unfold, $c =$ fold: Trivial.

Case $c = c'; c''$: We have $E \vdash c' : \tau \leq \delta$ and $E \vdash c'' : \delta \leq \sigma$ for some δ. By induction we gather $\mathrm{C}[\![E \vdash c' : \tau \leq \delta]\!] \leq \sqcup_n \mathrm{A}^n[\![E \vdash c' : \tau \leq \delta]\!]$ and similar for c''.

$$\begin{aligned}
\mathrm{C}[\![E \vdash (c'; c'') : \tau \leq \sigma]\!] &= \mathrm{C}[\![E \vdash c' : \tau \leq \delta]\!]\ ; \mathrm{C}[\![E \vdash c'' : \delta \leq \sigma]\!] \\
&\leq (\sqcup_n \mathrm{A}^n[\![E \vdash c' : \tau \leq \delta]\!])\ ; (\sqcup_n \mathrm{A}^n[\![E \vdash c'' : \delta \leq \sigma]\!]) \\
&= \bigsqcup_n (\mathrm{A}^n[\![E \vdash c' : \tau \leq \delta]\!]\ ; \mathrm{A}^n[\![E \vdash c'' : \delta \leq \sigma]\!]) \\
&= \bigsqcup_n \mathrm{A}^n[\![E \vdash (c'; c'') : \tau \leq \sigma]\!]
\end{aligned}$$

Case $c = c' \to c''$: Here $\tau = \tau'' \to \sigma'$, $\sigma = \tau' \to \sigma''$ and $E \vdash c' : \tau' \leq \tau''$, $E \vdash c'' : \sigma' \leq \sigma''$. The induction hypothesis thus applies for c' and c''. Consider the derivation

$$\begin{aligned}
&\mathrm{C}[\![E \vdash (c' \to c'') : (\tau'' \to \sigma') \leq (\tau' \to \sigma'')]\!] \\
&\quad = \quad \lambda\vec{E}.\, \lambda z.\, \mathrm{C}[\![E \vdash c' : \tau' \leq \tau'']\!]\ \vec{E}; z; \mathrm{C}[\![E \vdash c'' : \sigma' \leq \sigma'']\!]\ \vec{E} \\
&\quad \leq \quad \lambda\vec{E}.\, \lambda z.\, \left(\sqcup_n \mathrm{A}^n[\![E \vdash c' : \tau' \leq \tau'']\!]\ \vec{E}\right); z; \left(\sqcup_n \mathrm{A}^n[\![E \vdash c'' : \sigma' \leq \sigma'']\!]\ \vec{E}\right) \\
&\quad = \quad \bigsqcup_n \lambda\vec{E}.\, \lambda z.\, \mathrm{A}^n[\![E \vdash c' : \tau' \leq \tau'']\!]\ \vec{E}; z; \mathrm{A}^n[\![E \vdash c'' : \sigma' \leq \sigma'']\!]\ \vec{E} \\
&\quad = \quad \bigsqcup_n \mathrm{A}^{(n+1)}[\![E \vdash (c' \to c'') : \tau \leq \sigma]\!] \\
&\quad = \quad \bigsqcup_n \mathrm{A}^n[\![E \vdash (c' \to c'') : \tau \leq \sigma]\!]
\end{aligned}$$

Case $c = \mathbf{fix}\ f.c'$: Due to type-ability of c is $E \vdash c'[\mathbf{fix}\ f.c/c] : \tau \leq \sigma$ and by Lemma 6.2.2 we get $E(f : \tau \leq \sigma) \vdash c' : \tau \leq \sigma$. Induction hypothesis therefore allows us to conclude $\mathrm{C}[\![E(f : \tau \leq \sigma) \vdash c' : \tau \leq \sigma]\!] \ \leq \sqcup_n \mathrm{A}^n[\![E(f : \tau \leq \sigma) \vdash c' : \tau \leq \sigma]\!]$. On the other hand is $\mathrm{A}^n[\![E(f : \tau \leq \sigma) \vdash c' : \tau \leq \sigma]\!] \ \leq \mathrm{C}[\![E(f : \tau \leq \sigma) \vdash c' : \tau \leq \sigma]\!]$ for all n (shown above), so the limit is less than $\mathrm{C}[\![E(f : \tau \leq \sigma) \vdash c' : \tau \leq \sigma]\!]$ and hence $\sqcup_n \mathrm{A}^n[\![E(f : \tau \leq \sigma) \vdash c' : \tau \leq \sigma]\!] \ = \mathrm{C}[\![E(f : \tau \leq \sigma) \vdash c' : \tau \leq \sigma]\!]$.

Recall definition of $\mathrm{C}[\![]$ in the case of **fix**:

$$\mathrm{C}[\![E \vdash \mathbf{fix}\ f.c' : \tau \leq \sigma]\!] \ = \bigsqcup_i F^i(\bot)$$

with $F(f) = \lambda\vec{E}.\, \mathrm{C}[\![E(f : \tau \leq \sigma) \vdash c' : \tau \leq \sigma]\!]\ \vec{E}(f)$ that is, $\mathrm{C}[\![E \vdash \mathbf{fix}\ f.c' : \tau \leq \sigma]\!]$ is the least fix-point of F. We show that $\sqcup_n \mathrm{A}^n[\![E \vdash \mathbf{fix}\ f.c' : \tau \leq \sigma]\!]$ is also a fix-point of F and therefore less than $\mathrm{C}[\![E \vdash \mathbf{fix}\ f.c' : \tau \leq \sigma]\!]$ since it is the *least* fix-point.

$$\begin{aligned}
&F\,(\sqcup_n \mathrm{A}^n[\![E \vdash \mathbf{fix}\ f.c' : \tau \leq \sigma]\!]\ ) \\
&\quad = \quad \lambda\vec{E}.\, \mathrm{C}[\![E(f : \tau \leq \sigma) \vdash c' : \tau \leq \sigma]\!]\ \vec{E}(\sqcup_n \mathrm{A}^n[\![E \vdash \mathbf{fix}\ f.c' : \tau \leq \sigma]\!]\ ) \\
&\quad = \quad \lambda\vec{E}.\, \sqcup_n\, \mathrm{A}^n[\![E(f : \tau \leq \sigma) \vdash c' : \tau \leq \sigma]\!]\ \vec{E}(\sqcup_n \mathrm{A}^n[\![E \vdash \mathbf{fix}\ f.c' : \tau \leq \sigma]\!]\ ) \\
&\quad = \quad \lambda\vec{E}.\, \sqcup_n\, \mathrm{A}^n[\![E \vdash c'[\mathbf{fix}\ f.c'/f] : \tau \leq \sigma]\!]\ \vec{E} \\
&\quad = \quad \sqcup_n \mathrm{A}^n[\![E \vdash c'[\mathbf{fix}\ f.c'/f] : \tau \leq \sigma]\!] \\
&\quad = \quad \sqcup_n \mathrm{A}^n[\![E \vdash \mathbf{fix}\ f.c' : \tau \leq \sigma]\!]
\end{aligned}$$

where the third equation holds because of Lemma 8.2.6. Conclusion:

$$\mathrm{C}[\![E \vdash \mathbf{fix}\ f.c' : \tau \leq \sigma]\!] \ \leq \bigsqcup_n \mathrm{A}^n[\![E \vdash \mathbf{fix}\ f.c' : \tau \leq \sigma]\!]$$

and finally

$$\mathrm{C}[\![E \vdash c : \tau \leq \sigma]\!] \ = \bigsqcup_n \mathrm{A}^n[\![E \vdash c : \tau \leq \sigma]\!]$$

$\diamond$

8.3 Coercion Equality Soundness

Based on the approximated coercion interpretation we construct a stratified interpretation of coercion equality. We call it stratified because it is built from a sequence of levels corresponding to approximated coercion interpretations.

Definition 8.3.1 (**Stratified Equality Interpretation**) For well-typed coercions c, d with type judgements $E \vdash c : \tau \leq \sigma$ and $E \vdash d : \tau \leq \sigma$ we define

1. $\models_n c = d$ iff $\mathrm{A}^n[\![E \vdash c : \tau \leq \sigma]\!] = \mathrm{A}^n[\![E \vdash d : \tau \leq \sigma]\!]$
2. $\models_n A$ iff $\models_n c = d$ for all $(c = d) \in A$.
3. $A \models_n c = d$ iff $(\models_n A \Rightarrow \models_n c = d)$
4. $A \models c = d$ iff $\forall n \geq 0.\, A \models_n c = d$

$\diamond$

Lemma 8.3.2 *If c, d are well-typed coercions with $E \vdash c : \tau \leq \sigma$ and $E \vdash d : \tau \leq \sigma$ then*

$$A \vdash c = d \quad \Rightarrow \quad A \models c = d$$

PROOF Induction on derivation of $A \vdash c = d$. Only the interesting cases are shown. Note that $A \models_0 c = d$ trivially holds for all c and d.

Case (CE1): Show $A \models \bot; c = \bot$. We have for $n \geq 1$

$$\begin{aligned}
&\mathrm{A}^n[\![E \vdash \bot_\tau; c : \bot \leq \sigma]\!] \\
&\quad = \lambda\vec{E}.\, \mathrm{A}^n[\![E \vdash \bot_\tau : \bot \leq \tau]\!]\ \vec{E}; \mathrm{A}^n[\![E \vdash c : \tau \leq \sigma]\!]\ \vec{E} \\
&\quad = \lambda\vec{E}.\, (\lambda x.\, \bot_{\mathrm{T}[\![\tau]\!]}); \mathrm{A}^n[\![E \vdash c : \tau \leq \sigma]\!]\ \vec{E} \\
&\quad = \lambda\vec{E}.\, \lambda x.\, \bot_{\mathrm{T}[\![\sigma]\!]} \\
&\quad = \mathrm{A}^n[\![E \vdash \bot_\sigma : \bot \leq \sigma]\!]
\end{aligned}$$

Third equation holds because $\mathrm{A}^n[\![E \vdash c : \tau \leq \sigma]\!]\ \vec{E}$ is strict.

Case (CE5): Show $A \models \bot_\top = \top_\bot$. Observe that $\bot_{\mathrm{T}[\![\top]\!]} = \top$ and from this the case is immediately concluded.

Case (CE6): Show $A \models \iota_\tau; c = c$. For $n \geq 1$ is $\mathrm{A}^n[\![E \vdash \iota_\tau; c : \tau \leq \sigma]\!] = \lambda\vec{E}.\, \mathrm{A}^n[\![E \vdash \iota_\tau : \tau \leq \tau]\!]\ \vec{E}; \mathrm{A}^n[\![E \vdash c : \tau \leq \sigma]\!]\ \vec{E} = \lambda\vec{E}.\, (\lambda x.\, x); \mathrm{A}^n[\![E \vdash c : \tau \leq \tau]\!]\ \vec{E} = \mathrm{A}^n[\![E \vdash c : \tau \leq \sigma]\!]\ \vec{E}$. The result follows trivially from this observation.

Case (CE8): We have for $n \geq 1$

$$\begin{aligned}
&\mathrm{A}^n[\![E \vdash \iota_\tau \to \iota_{\tau'} : (\tau \to \tau') \leq (\tau \to \tau')]\!] \\
&\quad = \lambda\vec{E}.\, \lambda z.\, (\mathrm{A}^{(n-1)}[\![E \vdash \iota_\tau : \tau \leq \tau]\!]\ \vec{E}); z; (\mathrm{A}^{(n-1)}[\![E \vdash \iota_{\tau'} : \tau' \leq \tau']\!]\ \vec{E}) \\
&\quad = \mathrm{A}^n[\![E \vdash \iota_{\tau \to \tau'} : (\tau \to \tau') \leq (\tau \to \tau')]\!]
\end{aligned}$$

Case (CE12): For all $n \geq 1$ we can derive

$$\begin{aligned}
&\mathrm{A}^{(n+1)}[\![E \vdash (c' \to d); (c \to d') : (\sigma_1 \to \tau_2) \leq (\tau_1 \to \sigma_2)]\!] \\
&\quad = \lambda\vec{E}.\, (\mathrm{A}^{(n+1)}[\![E \vdash c' \to d]\!]\ \vec{E}); (\mathrm{A}^{(n+1)}[\![E \vdash c \to d']\!]\ \vec{E})
\end{aligned}$$

where the interpretations of $c' \to d$ and $c \to d'$ are

$$\begin{aligned}&\mathrm{A}^{(n+1)}[\![E \vdash c' \to d : (\sigma_1 \to \tau_2) \leq (\delta_1 \to \delta_2)]\!]\ \vec{E}\\ &\quad = \quad \lambda z.\,(\mathrm{A}^n[\![E \vdash c' : \delta_1 \leq \sigma_1]\!]\ \vec{E}); z; (\mathrm{A}^n[\![E \vdash d : \tau_2 \leq \delta_2]\!]\ \vec{E})\end{aligned}$$

$$\begin{aligned}&\mathrm{A}^{(n+1)}[\![E \vdash c \to d' : (\delta_1 \to \delta_2) \leq (\tau_1 \to \sigma_2)]\!]\ \vec{E}\\ &\quad = \quad \lambda z.\,(\mathrm{A}^n[\![E \vdash c : \tau_1 \leq \delta_1]\!]\ \vec{E}); z; (\mathrm{A}^n[\![E \vdash d' : \delta_2 \leq \sigma_2]\!]\ \vec{E})\end{aligned}$$

The composition of these equations give

$$\lambda x.\,(\mathrm{A}^n[\![E \vdash c]\!]\ \vec{E}); (\mathrm{A}^n[\![E \vdash c']\!]\ \vec{E}); x; (\mathrm{A}^n[\![E \vdash d]\!]\ \vec{E}); (\mathrm{A}^n[\![E \vdash d']\!]\ \vec{E})$$

since $f; g = \lambda x.\, g(f(x))$. The composition to the left of x in the above expression evaluates to

$$\begin{aligned}&\mathrm{A}^n[\![E \vdash c : \tau_1 \leq \delta_1]\!]\ \vec{E}; \mathrm{A}^n[\![E \vdash c' : \delta_1 \leq \sigma_1]\!]\ \vec{E}\\ &\quad = \quad \mathrm{A}^n[\![E \vdash (c; c') : \tau_1 \leq \sigma_1]\!]\ \vec{E}\end{aligned}$$

and similar for the right composition,

$$\begin{aligned}&\mathrm{A}^n[\![E \vdash d : \tau_2 \leq \delta_2]\!]\ \vec{E}; \mathrm{A}^n[\![E \vdash d' : \delta_2 \leq \sigma_2]\!]\ \vec{E}\\ &\quad = \quad \mathrm{A}^n[\![E \vdash (d; d') : \tau_2 \leq \sigma_2]\!]\ \vec{E}\end{aligned}$$

If we insert these results in the very first expression we obtain

$$\begin{aligned}&\lambda\vec{E}.\,(\mathrm{A}^{(n+1)}[\![E \vdash c' \to d]\!]\ \vec{E}); (\mathrm{A}^{(n+1)}[\![E \vdash c \to d']\!]\ \vec{E})\\ &\quad = \quad \lambda\vec{E}.\,\lambda x.\,\mathrm{A}^n[\![E \vdash (c; c') : \tau_1 \leq \sigma_1]\!]\ \vec{E}; x; \mathrm{A}^n[\![E \vdash (d; d') : \tau_2 \leq \sigma_2]\!]\ \vec{E}\\ &\quad = \quad \mathrm{A}^{(n+1)}[\![E \vdash (c; c') \to (d; d') : (\sigma_1 \to \tau_2) \leq (\tau_1 \to \sigma_2)]\!]\end{aligned}$$

Case (CE14): Show $A(c = d)A' \models c = d$, but since we assume $\models_n A(c = d)A'$ (Definition 8.3.1,3.) we can immediately conclude $\models_n c = d$.

Case (CE16): Let $A' = A((c \to d) = (c' \to d'))$. Induction hypothesis is that $A' \models c = c'$ and $A' \models d = d'$. We show $A \models_n (c \to d) = (c' \to d')$ for all n by a minor induction on n.

Case $n = 0$: Trivial.

Case $n > 0$: Show $A \models_n (c \to d) = (c' \to d')$, *i.e.* assume $\models_n A$ holds and prove $\models_n (c \to d) = (c' \to d')$. If $\models_n A$ then obviously $\models_{(n-1)} A$, because we do not require equality to depth n but only $n-1$. By the minor induction hypothesis (on n) we get that $A \models_{(n-1)} (c \to d) = (c' \to d')$ and since $\models_{(n-1)} A$ we find that $\models_{(n-1)} (c \to d) = (c' \to d')$ and hence $\models_{(n-1)} A'$ where A' is the extended assumption set. This enables us to conclude $\models_{(n-1)} c = c'$ and $\models_{(n-1)} d = d'$ due to $\models_{(n-1)} A'$ and the major induction hypothesis. The interpretation of $\models_{(n-1)} c = c'$ is $\mathrm{A}^{(n-1)}[\![E \vdash c : \tau \leq \sigma]\!] = \mathrm{A}^{(n-1)}[\![E \vdash c' : \tau \leq \sigma]\!]$ and of course the same for $\models_{(n-1)} d = d'$. Now we deduce

$$\begin{aligned}&\mathrm{A}^n[\![E \vdash c \to d : (\sigma \to \tau') \leq (\tau \to \sigma')]\!]\\ &\quad = \quad \lambda z.\,(\mathrm{A}^{(n-1)}[\![E \vdash c : \tau \leq \sigma]\!]\ ); z; (\mathrm{A}^{(n-1)}[\![E \vdash d : \tau' \leq \sigma']\!]\ )\\ &\quad = \quad \lambda z.\,(\mathrm{A}^{(n-1)}[\![E \vdash c' : \tau \leq \sigma]\!]\ ); z; (\mathrm{A}^{(n-1)}[\![E \vdash d' : \tau' \leq \sigma']\!]\ )\\ &\quad = \quad \mathrm{A}^n[\![E \vdash c' \to d' : (\sigma \to \tau') \leq (\tau \to \sigma')]\!]\end{aligned}$$

which completes the case. ◇

Lemma 8.3.3 *For $E \vdash c : \tau \leq \sigma$, $E \vdash d : \tau \leq \sigma$*

$$\models c = d \quad \Rightarrow \quad \mathrm{C}[\![E \vdash c : \tau \leq \sigma]\!] = \mathrm{C}[\![E \vdash d : \tau \leq \sigma]\!]$$

PROOF Because $\models c = d$ we have that $\models_n c = d$ for all $n \geq 0$. This means that $\mathrm{A}^n[\![E \vdash c : \tau \leq \sigma]\!] = \mathrm{A}^n[\![E \vdash d : \tau \leq \sigma]\!]$ for all $n \geq 0$ and hence $\sqcup_n \mathrm{A}^n[\![E \vdash c : \tau \leq \sigma]\!] = \sqcup_n \mathrm{A}^n[\![E \vdash d : \tau \leq \sigma]\!]$. Proposition 8.2.7 completes the proof: $\mathrm{C}[\![E \vdash c : \tau \leq \sigma]\!] = \sqcup_n \mathrm{A}^n[\![E \vdash c : \tau \leq \sigma]\!] = \sqcup_n \mathrm{A}^n[\![E \vdash d : \tau \leq \sigma]\!] = \mathrm{C}[\![E \vdash d : \tau \leq \sigma]\!]$. ◇

Theorem 8.3.4 ***(Coercion Equality Soundness)*** *For coercions c, d where $E \vdash c : \tau \leq \sigma$ we find that*

$$\vdash c = d \quad \Rightarrow \quad \mathrm{C}[\![E \vdash c : \tau \leq \sigma]\!] = \mathrm{C}[\![E \vdash d : \tau \leq \sigma]\!]$$

PROOF By Theorem 7.3.1 we know that $E \vdash d : \tau \leq \sigma$ since $\vdash c = d$. The result then follows directly from Lemma 8.3.2 and Lemma 8.3.3. ◇

Computational Adequacy

We briefly discuss computational adequacy of our denotational semantics, but since we have not given any formal operational semantics it does not make sense to prove anything.

Theorem 8.3.5 ***(Computational Adequacy)*** *Let $v : \tau$, $w : \sigma$ and $[\![v]\!] \in \mathrm{T}[\![\tau]\!]$, $[\![w]\!] \in \mathrm{T}[\![\sigma]\!]$ be the corresponding denotations. We have*

$$\mathrm{C}[\![\epsilon \vdash c : \tau \leq \sigma]\!]\, \epsilon\, [\![v]\!] = [\![w]\!] \text{ and } [\![w]\!] \neq \bot \quad \Leftrightarrow \quad (c\ v) \Rightarrow w$$

and furthermore

$$\mathrm{C}[\![\epsilon \vdash c : \tau \leq \sigma]\!]\, \epsilon\, [\![v]\!] = \bot \quad \Leftrightarrow \quad (c\ v) \text{ does not terminate.}$$

where $e \Rightarrow v$ means some big step operational semantics for coercions.

It would be interesting to prove such a theorem relating denotational and operational semantics.

Observational Congruence

We will also briefly mention an alternative approach to proving soundness of coercion equality.

Observational congruence of coercions (= programs) $c \approx c'$ means that c and c' behave the same in all well-formed contexts, especially $(c\ v) = (c'\ v)$ for all values v of appropriate type. Observational congruence have been studied by Gordon and Rees in [Gor95, GR96] where they use bisimulations[1] to characterize congruence. They show that if two programs are related by a bisimulation then they are observational congruent. Our coercion equality relation is tightly coupled with bisimulation, which might make the following proposition easy to prove.

[1] their notion of bisimulation can easily be related to ours, which again is related to Fiore's categorical bisimulations (see related works or [Fio94])

Proposition 8.3.6 *If* $\vdash c = c'$ *then* $c \approx c'$.

With this proposition and computational adequacy we may conclude soundness of coercion equality (Theorem 8.3.4), because $\vdash c = c'$ implies $c \approx c'$ and hence $(cv) = (c'v)$ for all v which by adequacy leads to $\mathrm{C}[\![\vdash c : \tau \leq \sigma]\!] = \mathrm{C}[\![\vdash c' : \tau \leq \sigma]\!]$. It would be interesting to pursue this idea in detail.

Chapter 9

Optimal Coercions

We have dealt with coercions as programs and we have discussed their operational semantics, but until now nothing as been said about efficiency. It is well-known that there are both good (efficient) and bad (inefficient) programs achieving the same observational behavior. This fact certainly also holds for coercions. Consider the following examples,

$$\iota ; c \qquad \textbf{unfold}; \textbf{fold} \qquad \iota \to \iota$$

These are all inefficient programs, but all well-typed and correct. First example requires a composition and a no-op (ι) more than actually required. Second example has three useless operations, **unfold**, **fold** and composition. Same observational behavior could be obtained by a simple identity coercion. The last example suffers the same inefficiencies and could also be replaced by a single ι. The last two examples come from a hole class of inefficient coercions: non-identity coercions with identity type signature $\tau \leq \tau$. Coercions from this class could, and should, be replaced by their corresponding identity coercion ι_τ, which definitely is the most efficient coercion for the job.

In this chapter we develop a theory of optimal coercions. First, we define what optimality formally means and then we show that all coercions have an optimal counterpart with the same observational behavior. During this proof we construct an algorithm for producing optimal coercions of any valid type signature!

9.1 Definition

We define in Figure 9.1 an order on coercions. If coercion c is less than coercion c' in this order it means that c is more efficient than c'. This intuition is essentially described by the rules (CS1-3), which state that ι_τ is a no-op and that it is the most efficient coercion with identity type signature. This order is actually derived from the coercion equality relation where we have directed some of the rules, especially the rules for identity (CS1-3). The advantage is of course that we affect the equality classes as little as possible. Based on this order we define the concept of optimal coercion.

Definition 9.1.1 A coercion c is *optimal* iff for all coercions c'

$$A \vdash c = c' \;\Rightarrow\; A \vdash c \leq c'$$

◇

In other words, a coercion is optimal if all other coercions from its equivalence class are larger than it.

In the following sections we prove that optimal coercions exist and we do so by giving an algorithm for constructing them.

9.2 Coercion Equality Normalization

To prove optimality we need a detailed understanding of coercion equality (see Figure 7.1). The tricky part of equality is, as always, transitivity. Transitivity is not syntax directed and is hence very difficult to control in inductive proofs. We would therefore like a notion of equality in which we can avoid transitivity. Fortunately, we have already done the hard work when we proved coercion coherence, namely the algorithm **E**. This algorithm proves two coercions equal merely by examining their syntax. In Figure 9.2 we present a normalized equality axiomatization derived directly from algorithm **E**.

Lemma 9.2.1

$$A \vdash c = d \quad \Leftrightarrow \quad A \vdash_n c = d$$

PROOF It is quite easy to realize that $A \vdash_n c = d$ implies $A \vdash c = d$ — you simply go thru all the rules of normalized equality and show that the conclusion can be obtained by ordinary equality. To prove the other way around we first note that $A \vdash c = d$ especially implies that c, d have the same type signature (Lemma 7.3.1) and hence **E** returns TRUE (shown in Lemma 7.3.1). We can then directly translate the execution of **E** to a correct derivation in $\vdash_n$ and thus prove $A \vdash_n c = d$. ◇

Lemma 9.2.2

$$\vdash nf(c) \leq c$$

PROOF Consider $c' \Rightarrow c''$ as defined in section 7.2. By rule induction on this derivation it is easily seen that $\vdash c'' \leq c'$. The property for multiple reduction steps are obtained by induction on the number of steps and the property for single step reduction. ◇

9.3 Optimal Coercion Algorithm

The algorithm presented in this chapter is not exactly new. In the chapter about completeness of the subtype relation we encountered the algorithm **S**, which given an assumption set and two recursive types produced a proof of them being in the subtype relation (if possible). Subtype proofs are, as we saw in Chapter 6, coercions, that is we may think of **S** as producing coercions instead of subtype proofs. The question is then whether **S** produces optimal coercions or not? It does not, but a few obvious adjustments can easily remedy the problems. In Figure 9.3 we present algorithm **O** which given an environment

$$A \vdash \bot_\tau \leq \bot_\tau ; c \,:\, \bot \leq \sigma \quad \text{(CS1)} \qquad A \vdash \top_\tau \leq c ; \top_\tau \,:\, \sigma \leq \top \quad \text{(CS2)}$$

$$A \vdash \bot_\top \leq \top_\bot \,:\, \bot \leq \top \quad \text{(CS3)} \qquad A \vdash \top_\bot \leq \bot_\top \,:\, \bot \leq \top \quad \text{(CS4)}$$

$$A \vdash c \leq \iota_\tau ; c \,:\, \tau \leq \sigma \quad \text{(CS5)} \qquad A \vdash c \leq c ; \iota_\sigma \,:\, \tau \leq \sigma \quad \text{(CS6)}$$

$$A \vdash \iota_\tau \leq c \,:\, \tau \leq \tau \quad \text{(CS7)}$$

$$A \vdash c ; (c' ; c'') \leq (c ; c') ; c'' \,:\, \tau \leq \sigma \quad \text{(CS8)} \qquad A \vdash (c ; c') ; c'' \leq c ; (c' ; c'') \,:\, \tau \leq \sigma \quad \text{(CS9)}$$

$$A \vdash (c' \to d) ; (c \to d') \leq (c ; c') \to (d ; d') \,:\, \tau \leq \sigma \quad \text{(CS10)}$$

$$A \vdash (c ; c') \to (d ; d') \leq (c' \to d) ; (c \to d') \,:\, \tau \leq \sigma \quad \text{(CS11)}$$

$$A \vdash \mathbf{fix}\, f.c \leq c[\mathbf{fix}\, f.c/f] \,:\, \tau \leq \sigma \quad \text{(CS12)} \qquad A \vdash c[\mathbf{fix}\, f.c/f] \leq \mathbf{fix}\, f.c \,:\, \tau \leq \sigma \quad \text{(CS13)}$$

$$A(c \leq d)A' \vdash c \leq d \,:\, \tau \leq \sigma \quad \text{(CS14)}$$

$$\frac{A(c \to d \leq c' \to d') \vdash c \leq c' \,:\, \tau_1 \leq \tau_2 \qquad A(c \to d \leq c' \to d') \vdash d \leq d' \,:\, \sigma_1 \leq \sigma_2}{A \vdash (c \to d) \leq (c' \to d') \,:\, (\tau_2 \to \sigma_1) \leq (\tau_1 \to \sigma_2)} \quad \text{(CS15)}$$

$$\frac{A \vdash c \leq c' \,:\, \tau \leq \delta \quad A \vdash d \leq d' \,:\, \delta \leq \sigma}{A \vdash c ; d \leq c' ; d' \,:\, \tau \leq \sigma} \quad \text{(CS16)}$$

$$A \vdash c \leq c \,:\, \tau \leq \sigma \quad \text{(CS17)}$$

$$\frac{A \vdash c \leq c' \,:\, \tau \leq \sigma \qquad A \vdash c' \leq c'' \,:\, \tau \leq \sigma}{A \vdash c \leq c'' \,:\, \tau \leq \sigma} \quad \text{(CS18)}$$

Figure 9.1: Coercion Sub (CS) relation

$$\frac{A \vdash_n \mathbf{unfold}; \iota_{\tau[\mu\alpha.\tau/\alpha]}; \mathbf{fold} = c}{A \vdash_n \iota_{\mu\alpha.\tau} = c} \quad \text{(CEN1)} \qquad \frac{A \vdash_n c = \mathbf{unfold}; \iota_{\tau[\mu\alpha.\tau/\alpha]}; \mathbf{fold}}{A \vdash_n c = \iota_{\mu\alpha.\tau}} \quad \text{(CEN2)}$$

$$\frac{A \vdash_n \iota_{\tau_1} \to \iota_{\tau_2} = c}{A \vdash_n \iota_{\tau_1 \to \tau_2} = c} \quad \text{(CEN3)} \qquad \frac{A \vdash_n c = \iota_{\tau_1} \to \iota_{\tau_2}}{A \vdash_n c = \iota_{\tau_1 \to \tau_2}} \quad \text{(CEN4)}$$

$$A(c =_n d)A' \vdash_n c = d \quad \text{(CEN5)} \qquad A \vdash_n c = c \quad \text{(CEN6)}$$

$$\frac{A(c \to c' =_n d \to d') \vdash_n \mathrm{nf}(c) = \mathrm{nf}(d) \qquad A(c \to c' =_n d \to d') \vdash_n \mathrm{nf}(c') = \mathrm{nf}(d')}{A \vdash_n c \to c' = d \to d'} \quad \text{(CEN7)}$$

$$\frac{A \vdash_n c \to c' = d \to d'}{A \vdash_n (\mathbf{unfold}; c \to c') = (\mathbf{unfold}; d \to d')} \quad \text{(CEN8)}$$

$$\frac{A \vdash_n c \to c' = \iota}{A \vdash_n (\mathbf{unfold}; c \to c') = \mathbf{unfold}} \quad \text{(CEN9)}$$

$$\frac{A \vdash_n \iota = c \to c'}{A \vdash_n \mathbf{unfold} = (\mathbf{unfold}; c \to c')} \quad \text{(CEN10)}$$

$$\frac{A \vdash_n c \to c' = d \to d'}{A \vdash_n (c \to c'; \mathbf{fold}) = (d \to d'; \mathbf{fold})} \quad \text{(CEN11)}$$

$$\frac{A \vdash_n c \to c' = \iota}{A \vdash_n (c \to c'; \mathbf{fold}) = \mathbf{fold}} \quad \text{(CEN12)}$$

$$\frac{A \vdash_n \iota = c \to c'}{A \vdash_n \mathbf{fold} = (c \to c'; \mathbf{fold})} \quad \text{(CEN13)}$$

$$\frac{A \vdash_n c \to c' = d \to d'}{A \vdash_n (\mathbf{unfold}; c \to c'; \mathbf{fold}) = (\mathbf{unfold}; d \to d'; \mathbf{fold})} \quad \text{(CEN14)}$$

Figure 9.2: Coercion Equality Normalized (CEN)

E, mapping coercion variables to type assumptions, and two recursive types τ, σ produces an *optimal* coercion $c : \tau \leq \sigma$ whenever $E_A \vdash \tau \leq \sigma$, where E_A denotes the assumptions embedded in E. To prove this conjecture we have formalized the execution of **O** just as we did with coercion equality in the previous section (see Figure 9.4). What results from this is actually a normalized axiomatization of the subtype relation. As with coercion equality the advantage is that we get control of the transitivity rule.

Consider the following coercion grammar

$$\begin{array}{lcl} C_{\min} & ::= & \iota_\tau \mid \bot_\tau \mid \top_\tau \mid \mathbf{unfold} \mid \mathbf{fold} \mid C'_{\min} \\ & & \mid \mathbf{unfold}; C'_{\min} \mid C'_{\min}; \mathbf{fold} \mid \mathbf{unfold}; C'_{\min}; \mathbf{fold} \\ C'_{\min} & ::= & f \mid \mathbf{fix}\, f.C_{\min} \to C_{\min} \end{array}$$

with the syntactic restrictions

1. $C'_{\min} \neq \mathbf{fix}\, f.\iota \to \iota$
2. $\bot_\tau \neq \bot_\bot$
3. $\top_\tau \neq \top_\top$

Lemma 9.3.1 *The following properties hold for the normalized subtype relation and the normalized coercion grammar:*

1. *If* $E \vdash_{min} c : \tau \leq \sigma$ *then* $c \in C_{min}$.
2. *If* $c \in C_{min}$ *and* $E \vdash c : \tau \leq \sigma$ *then* $E \vdash_{min} c : \tau \leq \sigma$.
3. $E \vdash_{min} c : \tau \leq \tau$ *iff* $c \equiv \iota_\tau$.

where E is well-formed, i.e. for $(f : \tau \leq \sigma) \in E$ we have $\tau \neq \sigma$ and $\tau \equiv \tau_1 \to \tau_2$, $\sigma \equiv \sigma_1 \to \sigma_2$.

PROOF

Ad 1. Rule induction on derivation of $E \vdash_{\min} c : \tau \leq \sigma$. We only show the interesting cases.

Case (SN6) : By induction hypothesis we conclude that $c \in C_{\min}$. To demonstrate that $(\mathbf{unfold}; c) \in C_{\min}$ we do a case analysis on the structure of c.

Case $c \equiv \iota$**:** Not possible due to side condition of rule (SN6).

Case $c \equiv \bot$**:** Not possible because $\mathbf{unfold} : \mu\alpha.\tau \leq \tau[\mu\alpha.\tau/\alpha]$ and $\tau[\mu\alpha.\tau/\alpha] \neq \bot$.

Case $c \equiv \top$**:** Side condition prevents this case.

Case $c \equiv \mathbf{unfold}$**:** Since we only work with contractive recursive types we can not possible have $\tau[\mu\alpha.\tau/\alpha] \equiv \mu\beta.\sigma$ and thus this case is not feasible.

Case $c \equiv \mathbf{fold}$**:** N/A due to side condition of (SN6)

Case $c \in C'_{\mathbf{min}}$**:** Ok, because $\mathbf{unfold}; C'_{\min} \in C_{\min}$.

Case $c \equiv \mathbf{unfold}; c'$**:** Contractivity of $\mu\alpha.\tau$ prevents this case.

Case $c \equiv c'; \mathbf{fold}$**:** With $c' \in C'_{\min}$ we get that $\mathbf{unfold}; c'; \mathbf{fold} \in C_{\min}$.

Case $c \equiv \mathbf{unfold}; c'; \mathbf{fold}$**:** Not possible for the same reason as $c \equiv \mathbf{unfold}; c'$.

Case (SN7): Analogous to (SN6).

Case (SN9): By induction hypothesis we get that $c_1 \in C_{\min}$ and $c_2 \in C_{\min}$. The side condition of (SN9) guarantees that c_1 and c_2 *not* both are identity, that is $\mathbf{fix}\, f.c_1 \to c_2 \in C_{\min}$.

Ad. 2 Induction on the structure of $c \in C_{\min}$.

Case $c \in \{\iota, \bot, \top, \mathbf{unfold}, \mathbf{fold}\}$**:** Trivial, since the exact same typing rules exist in the normalized subtype system.

Case $c \equiv f$**:** We have due to type-ability $E(f : \tau \leq \sigma)E' \vdash f : \tau \leq \sigma$. According to (SN8) we directly conclude the result.

Case $c \equiv \mathbf{fix}\, f.c_1 \to c_2$**:** Due to type-ability we know that $E' \vdash c_1 : \sigma_1 \leq \tau_1$ and $E' \vdash c_2 : \tau_2 \leq \sigma_2$, with $E' = E(f : \tau_1 \to \tau_2 \leq \sigma_1 \to \sigma_2)$. The syntactic restriction on $C'_{\min}$ guarantees us that $\tau_1 \to \tau_2 \neq \sigma_1 \to \sigma_2$ and hence is E' well-formed. The induction hypothesis then provides us with proofs of $E' \vdash_{\min} c_1 : \sigma_1 \leq \tau_1$ and $E' \vdash_{\min} c_2 : \tau_2 \leq \sigma_2$. Rule (SN9) then immediately concludes case.

Ad 3. The direction from right to left is easy and proven by (SN1). Direction from left to right is proven by a case analysis on the derivation of $E \vdash_{\min} c : \tau \leq \tau$.

Case (SN1): Ok.

Case (SN2–9): Not possible since the type signatures are not identical. Note that rule (SN8) can not have identical type signatures because E is well–formed. ◇

1: $\mathbf{O}(E, \tau, \tau) = \iota_\tau$ [1]
2: $\mathbf{O}(E, \bot, \tau) = \bot$
3: $\mathbf{O}(E, \tau, \top) = \top$
4: $\mathbf{O}(E, \mu\alpha.\tau, \tau[\mu\alpha.\tau/\alpha]) = \mathbf{unfold}$
5: $\mathbf{O}(E, \mu\alpha.\tau, \sigma) =$
6: $\quad \mathbf{unfold}; \mathbf{O}(E, \tau[\mu\alpha.\tau/\alpha], \sigma)$
7: $\mathbf{O}(E, \sigma[\mu\beta.\sigma/\beta], \mu\beta.\sigma) = \mathbf{fold}$
8: $\mathbf{O}(E, \tau, \mu\beta.\sigma) =$
9: $\quad \mathbf{O}(E, \tau, \sigma[\mu\beta.\sigma/\beta]); \mathbf{fold}$
10: $\mathbf{O}(E(f : \tau \leq \sigma), \tau, \sigma) = f$
11: $\mathbf{O}(E, \tau_1 \to \tau_2, \sigma_1 \to \sigma_2) =$
12: $\quad \mathbf{let}$
13: $\quad\quad E' = E(f : \tau_1 \to \tau_2 \leq \sigma_1 \to \sigma_2)$
14: $\quad \mathbf{in}$
15: $\quad\quad \mathbf{fix}\, f.\mathbf{O}(E', \sigma_1, \tau_1) \to \mathbf{O}(E', \tau_2, \sigma_2)$
16: $\quad \mathbf{end}$
17: $\mathbf{O}(E, \tau, \sigma) =$ **exception**

Figure 9.3: Optimal Coercion Algorithm

$$E \vdash_{\min} \iota_\tau \ : \ \tau \leq \tau \qquad \text{(SN1)}$$

$$E \vdash_{\min} \bot_\tau \ : \ \bot \leq \tau \quad (\tau \neq \bot) \qquad \text{(SN2)} \qquad E \vdash_{\min} \top_\tau \ : \ \tau \leq \top \quad (\tau \neq \top) \qquad \text{(SN3)}$$

$$E \vdash_{\min} \mathbf{unfold} \ : \ \mu\alpha.\tau \leq \tau[\mu\alpha.\tau/\alpha] \qquad \text{(SN4)} \qquad E \vdash_{\min} \mathbf{fold} \ : \ \tau[\mu\alpha.\tau/\alpha] \leq \mu\alpha.\tau \qquad \text{(SN5)}$$

$$\frac{E \vdash_{\min} c \ : \ \tau[\mu\alpha.\tau/\alpha] \leq \sigma}{E \vdash_{\min} \mathbf{unfold}; c \ : \ \mu\alpha.\tau \leq \sigma} \quad (\sigma \notin \{\top, \mu\alpha.\tau, \tau[\mu\alpha.\tau/\alpha]\}) \qquad \text{(SN6)}$$

$$\frac{E \vdash_{\min} c \ : \ \sigma \leq \tau[\mu\alpha.\tau/\alpha]}{E \vdash_{\min} c; \mathbf{fold} \ : \ \sigma \leq \mu\alpha.\tau} \quad (\sigma \notin \{\bot, \mu\alpha.\tau, \tau[\mu\alpha.\tau/\alpha]\}) \qquad \text{(SN7)}$$

$$E(f : \tau \leq \sigma)E' \vdash_{\min} f \ : \ \tau \leq \sigma \qquad \text{(SN8)}$$

$$\frac{\begin{array}{l} E' = E(f : \tau_1 \to \tau_2 \leq \sigma_1 \to \sigma_2) \\ E' \vdash_{\min} c_1 \ : \ \sigma_1 \leq \tau_1 \quad E' \vdash_{\min} c_2 \ : \ \tau_2 \leq \sigma_2 \end{array}}{E \vdash_{\min} \mathbf{fix}\, f.c_1 \to c_2 \ : \ \tau_1 \to \tau_2 \leq \sigma_1 \to \sigma_2} \quad (\tau_1 \to \tau_2 \neq \sigma_1 \to \sigma_2) \qquad \text{(SN9)}$$

Figure 9.4: Subtype relation Normalized (SN)

$$\frac{c \sqsubseteq c_1}{c \sqsubseteq c_1 \to c_2} \qquad \frac{c \sqsubseteq c_2}{c \sqsubseteq c_1 \to c_2}$$

$$\frac{c \sqsubseteq c_1}{c \sqsubseteq c_1; c_2} \qquad \frac{c \sqsubseteq c_2}{c \sqsubseteq c_1; c_2}$$

$$c \sqsubseteq c \qquad \frac{c \sqsubseteq c'[\mathbf{fix}\, f.c'/f]}{c \sqsubseteq \mathbf{fix}\, f.c'}$$

Figure 9.5: Coercion subterm relation

The next step on our way of proving optimality of **O** is to define coercion subterms (Figure 9.5). This trick as been very helpful in the preceding chapters and so will it also in this proof.

Observe that $c' \sqsubseteq c$ where $c \in C_{\min}$ implies that $c' \in C^{sub}_{\min}$, where $C^{sub}_{\min}$ is defined by

$$C^{sub}_{\min} ::= C_{\min} \mid C_{\min} \to C_{\min}$$

with the same restriction as $C_{\min}$. We can also prove the identity lemma for subterms of minimal coercions.

Lemma 9.3.2 *If $c \in C^{sub}_{min}$ and $E \vdash c : \tau \leq \tau$ then $c \equiv \iota_\tau$.*

PROOF Case analysis of $C^{sub}_{\min}$.

Case $c \in C_{\min}$: By property 2 of Lemma 9.3.1 we get $E \vdash_{\min} c : \tau \leq \tau$. Property 3 then concludes case by stating $c \equiv \iota_\tau$.

Case $c \in C_{\min} \to C_{\min}$: From arrow type rule we conclude that the branches also have identity type signatures. The argument of the previous case states that the branches are identity coercions, but then the syntactic restriction prohibits c. Case can not occur. ◇

With these identity results at hand we are able to prove that coercions typed with $\vdash_{\min}$ are optimal, *i.e.* coercions produced by **O** are optimal.

Lemma 9.3.3 *Let $\vdash_{min} c : \tau \leq \sigma$ be given. For all c', d with $c' \sqsubseteq c$ and assumptions A we have that*

$$A \vdash_n c' = d \quad \Rightarrow \quad \widetilde{A} \vdash c' \leq d$$

where $\widetilde{A} = \{(c \leq d) \mid (c = d) \in A\}$.

PROOF Rule induction on derivation of $A \vdash_n c' = d$.

Case (CEN1,3): Since $\iota \leq d$ for all d by (CS7) we can immediately conclude case.

Case (CEN2,4): We have $A \vdash_n d = \iota$ and thus $d : \tau \leq \tau$ because of soundness. Furthermore $d \sqsubseteq c$ so by Lemma 9.3.2 we get $d \equiv \iota$. Reflexivity (CS17) finally completes case.

Case (CEN5): If $(c = d) \in A$ then $(c \leq d) \in \widetilde{A}$ so by (CS14) this case is concluded.

Case (CEN6): Trivial by (CS17).

Case (CEN7): In this case we have $A \vdash c_1 \to c_2 = d_1 \to d_2$ and by definition of coercion subterms are $c_1 \sqsubseteq c_1 \to c_2$ and $c_2 \sqsubseteq c_1 \to c_2$. Note that

$$\widetilde{A'} = \widetilde{A}(c_1 \to c_2 \leq d_1 \to d_2) = \widetilde{A}'$$

We now prove that $\widetilde{A}' \vdash c_1 \leq \mathrm{nf}(d_1)$. We know that $c_1 \in C^{sub}_{\min}$ so we do a case analysis on its structure. Note that c_1 is closed, *i.e.* there does not occur any free variables in it.

Case $c_1 = \iota, c_1 =$ unfold, $c_1 =$ fold: We have that $\mathrm{nf}(c_1) = c_1$ so the induction hypothesis applies and gives $\widetilde{A}' \vdash \mathrm{nf}(c_1) \leq \mathrm{nf}(d_1)$.

Case $c_1 = \mathbf{fix}\ f.c_1 \to c_2$**:** Here $\mathrm{nf}(c_1) \sqsubseteq c_1$ and hence by induction hypothesis as above we conclude case.

Case $c_1 = \mathbf{unfold}; \mathbf{fix}\, f.c_1' \to c_2'$**:** The normal form of c_1 is $\mathbf{unfold}; c_1'' \to c_2''$ where c_1'', c_2'' are the unfolded versions of c_1', c_2'. At this point we do a case analysis of $A' \vdash \mathbf{unfold}; c_1'' \to c_2'' = \mathrm{nf}(d_1)$.

Case (CEN2): If this rule proves $A' \vdash \mathbf{unfold}; c_1'' \to c_2'' = \mathrm{nf}(d_1)$ then $\mathbf{unfold}; c_1'' \to c_2'' : \tau \le \tau$ and hence by Lemma 9.3.2 $\mathbf{unfold}; c_1'' \to c_2'' : \tau \le \tau \equiv \iota_\tau$, but this obviously is not possible. We conclude that this rule can not have proven our judgement.

Case (CEN4): Not possible because $\mathbf{unfold}; c_1'' \to c_2''$ can not have an arrow type signature.

Case (CEN5): Easy since equality assumption is converted to inequality assumption in $\widetilde{A}$.

Case (CEN6): Trivial.

Case (CEN8): We learn that $A' \vdash c_1'' \to c_2'' = d_1' \to d_2'$ where $\mathrm{nf}(d_1) = \mathbf{unfold}; d_1' \to d_2'$. With this coercion (left hand side) we are back in business,

$$c_1'' \to c_2'' \sqsubseteq \mathbf{fix}\, f.c_1' \to c_2' \sqsubseteq \mathbf{unfold}; \mathbf{fix}\, f.c_1' \to c_2' \sqsubseteq c$$

Our induction hypothesis applies and yields $\widetilde{A}' \vdash c_1'' \to c_2'' \le d_1' \to d_2'$ and by (CS16,17)

$$\widetilde{A}' \vdash \mathbf{unfold}; c_1'' \to c_2'' \le \mathrm{nf}(d_1)$$

Case (CEN9): If this rule proves judgement then we arrive at a contradiction just as in case (CEN2) above.

Case (CEN10): We may immediately conclude $\widetilde{A}' \vdash \iota \le d_1' \to d_2'$ due to rule (CS7) and hence also desired judgement.

Case $c_1 = \mathbf{fix}\ f.c_1' \to c_2'; \mathbf{fold}$**:** A similar analysis to the one performed above is needed here.

Case $c_1 = \mathbf{unfold}; \mathbf{fix}\, f.c_1' \to c_2'; \mathbf{fold}$**:** As previous two cases.

Case $c_1 = c_1' \to c_2'$**:** We see that $\mathrm{nf}(c_1) = c_1$ and then induction hypothesis directly finish case.

We have just proved that $\widetilde{A}' \vdash c_1 \le \mathrm{nf}(d_1)$ and similarly is $\widetilde{A}' \vdash c_2 \le \mathrm{nf}(d_2)$ proven. From Lemma 9.2.2 we get that $\vdash \mathrm{nf}(d) \le d$ and therefore $\widetilde{A}' \vdash c_1 \le d_1$, $\widetilde{A}' \vdash c_2 \le d_2$. Rule (CS15) finally completes the case by stating that $\widetilde{A} \vdash c_1 \to c_2 \le d_1 \to d_2$.

Case (CEN8): Since $c_1 \to c_2 \sqsubseteq (\mathbf{unfold}; c_1 \to c_2)$ we may apply hypothesis to obtain $\tilde{A} \vdash c_1 \to c_2 \le d_1 \to d_2$. The final result is achieved by compatibility of composition (CS16) and reflexivity (CS17).

Case (CEN9): The premise of (CEN9) states that $A \vdash c_1 \to c_2 = \iota$ and hence by soundness that $c_1 \to c_2 : \tau' \le \tau'$ for some τ'. Because

$$c_1 \to c_2 \sqsubseteq (\mathbf{unfold}; c_1 \to c_2) \sqsubseteq c$$

we get from Lemma 9.3.2 that $c_1 \to c_2 \equiv \iota$ which clearly is not true. We have thus reached a contradiction which can only mean that (CEN9) can not have been applied.

Case (CEN10): Premise of rule states $A \vdash \iota = d_1 \to d_2$ and by (CS7) we get $\tilde{A} \vdash \iota \leq d_1 \to d_2$. From (CS6) we know that $\tilde{A} \vdash \mathbf{unfold} \leq \mathbf{unfold}; \iota$. The remainder of the case is handled by (CS16,17) as in case (CEN8).

Case (CEN11-14): Analogous to (CEN8-10). ◇

We can now formulate and prove the main result of this chapter, namely that **O** and $\vdash_{\text{min}}$ produce optimal coercions.

Theorem 9.3.4 *(Optimal Coercions) If* $\vdash_{min} c : \tau \leq \sigma$ *or* $c = \mathbf{O}(\epsilon, \tau, \sigma)$ *then* c *is optimal.*

PROOF Since $c \sqsubseteq c$ we can immediately conclude that c is optimal by Lemma 9.3.2. ◇

Chapter 10

Related Work

We will in this chapter make a survey of related works and possible applications of our work in other scenarios. Such surveys will almost always be incomplete, which is indeed true of ours.

10.1 Recursive Subtyping

As mentioned plenty of times is this report heavily inspired by the work of Amadio, Cardelli ([AC91, AC93]). They present definitions of recursive type equality (largely due to Cardone, Coppo [CC91]) and subtyping. Furthermore they give sound and complete axiomatizations of these relations by means of the (Contract) rule. A soundness property in a cpo-based interpretation model is also shown for the two relations. Amadio, Cardelli moreover address the issue of coercions, by given mathematical functions acting as coercions in their interpretation model. Note that coercions are not given any operational semantics or interpretation, which makes them hard to relate to actual programming languages.

Kozen, Palsberg and Schwartzbach continues the work of Amadio, Cardelli by implementing an efficient ($\mathbf{O}(n^2)$) decision algorithm for recursive subtyping [KPS93, KPS95]. Their work is interesting, besides the algorithm, because they present a simple characterization of recursive subtyping. They state for recursive types τ, σ that $\tau \leq_{\text{AC}} \sigma$ if no common (finite) path detects a label mismatch in Tree(τ) and Tree(σ). If you thing about this somewhat informal definition you quickly realize that they are actually giving a coinductive characterization of subtyping — if you can not find a counter example then it holds! This is exactly then paradigm underpinning coinduction. Kozen, Palsberg and Schwartzbach does not pursue and investigate this coinductive characterization further, they only use the operational contents of it to built an efficient algorithm.

Many people have given type inference systems for recursive subtyping, based more or less on Amadio, Cardelli's work. Sekiguchi and Yonezawa show in [SeYo94] a complete and sound inference system for lambda calculus with recursive types, structural subtyping and parametric polymorphism. They do not use the axiomatization of Amadio, Cardelli and hence not the (Contract) rule, but they do regard recursive types as infinite regular trees and thus base their subtype relation on the definition of Amadio, Cardelli. Damm presents

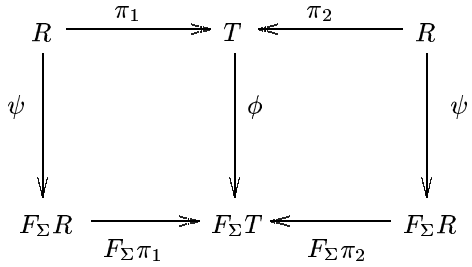
in [Dam94] a type inference system with subtyping of union-, intersection- and recursive types, which is also based on the Amadio, Cardelli definition of recursive subtyping. Both of these works does not mention coercions at all, but focuses entirely on type inference.

10.2 Bisimulation

Fiore develops in [Fio94] a coinduction proof principle based on bisimulation. Below we sketch his presentation and we conclude with his coinduction principle for regular trees.

Every alphabet Σ with rank function $\rho : \Sigma \to \mathbb{N}$ (also known as a one-sorted signature) induces an endofunctor $F_\Sigma(\_) = \coprod_n \Sigma_n \times (\_)^n$ on the category **Set**, where $\coprod$ and $\times$ respectively denotes the coproduct and product functors. Σ_n denotes all operator symbols $o \in \Sigma$ with $\rho(o) = n$. Fiore then defines categorical F_Σ-bisimulations.

Definition 10.2.1 A relation $R \subseteq T \times T$ is a categorical F_Σ-bisimulation on the F_Σ-coalgebra (T, ϕ) if there exists an F_Σ-coalgebra structure ψ on R such that



commutes, where π_1, π_2 are the canonical projections. ◇

Next, he shows an adequate and necessary condition on relations for being bisimulations.

Proposition 10.2.2 *A relation $R \subseteq T \times T$ is a categorical bisimulation on $\phi : T \to F_\Sigma T$ if and only if $t \, R \, t'$ implies*

$$\mathbf{root}_\phi(t) = \mathbf{root}_\phi(t') \in \Sigma_n \ \wedge \ \forall 1 \leq i \leq n.\ \mathbf{subtree}_\phi(t, i) \, R \, \mathbf{subtree}_\phi(t', i)$$

where $\mathbf{root}_\phi(t) = o$, $\mathbf{subtree}_\phi(t, i) = t_i$ *for* $\phi(t) = \big(o, (t_1, \ldots, t_n)\big) \in \Sigma_n \times T^n$.

We begin to see some resemblance with our definition of bisimulation on recursive types. Fiore gives a categorical bisimulation characterization of regular trees (finite and infinite). Fiore makes the set of regular trees, T^Σ, a *final* F_Σ-coalgebra, T^Σ_ω, by introducing the following structure map,

$$\mathrm{T}^\Sigma_\omega \to \coprod_n \Sigma_n \times (\mathrm{T}^\Sigma_\omega)^n \ : t \mapsto \big(t(\epsilon), \big\{\lambda\pi \in \mathbb{N}^*.\ t(i\pi)\big\}_i\big)$$

For this F_Σ-coalgebra we have the interesting coinduction principle on regular trees.

Proposition 10.2.3 *If* $\Sigma = \{\bot_0, \top_0, \to_2\}$ *and* R *is a binary relation on* T_ω^Σ *satisfying*

1. $t_1 \to t_2 \, R \, s_1 \to s_2 \;\Rightarrow\; t_1 \, R \, s_1$ *and* $t_2 \, R \, s_2$
2. $t \, R \, s \;\Rightarrow\; t(\epsilon) = s(\epsilon)$

then R *is a categorical bisimulation and furthermore* $t \, R \, t' \;\Rightarrow\; t = t'$.

It is an interesting exercise to see that this definition of bisimulation on regular trees is equivalent to our definition on recursive types, but we will not go into details here.

10.3 Program Correctness

Nielson studies in [Nie85] correctness of nested recursive procedures. The interesting aspect of her work in connection to ours is the paradigm of "proofs from assumptions". She formalizes the paradigm with a very familiar rule (original from [Hoa71]),

$$\frac{\phi_1, \ldots, \phi_n, P[\textsc{Call}\ p]Q \vdash P[c]Q}{\phi_1, \ldots, \phi_n \vdash P[\textsc{Call}\ p]Q}$$

where p is a procedure with body c. The construct $P[c]Q$ says that the program c is correct with respect to the pre-conditions P and post-conditions Q. The judgement then intuitively says that if $P[c]Q$ can be shown from the assumption that $P[\textsc{Call}\ p]Q$ holds for all recursive calls of p in c (plus the assumptions $\phi_1, \ldots, \phi_n$) then it actually holds for every call of p. The notion of coinduction is not pursued in [Nie85], but it would be interesting to know if such a correlation exists. We see from Nielson's work (and others in that field, [Hoa71, Sok77]) that the paradigm "proofs from assumptions" and our (Fix) rule are well-known concepts. We do, however, believe that they are new in the setting of recursive subtyping and type theory in general.

10.4 Static Program Analysis

Static program analysis is a discipline with many promising directions, *e.g.* boxing analysis, binding-time analysis and strictness analysis to name a few. A technique common to a hole class of analyses is to devise a type system expressing features of the terms being examined. Consider boxing analysis. A recursive type system suitable for this analysis is

$$\tau \equiv \alpha \mid \tau \to \tau \mid \mu\alpha.\tau \mid [\tau]$$

The intuition of $[\tau]$ is that a term typed $[\tau]$ is a *boxed* value of type τ. All other types are *unboxed*. This kind of box types are used in [Jør96], but without recursive types. Boxed types express representation details about terms, but the values are fundamentally the same and may thus be used interchangeable, hence some kind of equality relation is needed.

$$\tau \leq [\tau] \qquad [\tau] \leq \tau$$

We have chosen to present the relation as inequality to make an analogy with the remainder of this report. Besides these two rules we need rules for coping with recursive types (rule (Fix)) and the other type constructs (rules (Arrow), (Ref) *etc.*). The next naturally step, in analogy with this report, is to give each rule a coercion encoding and associate an operational semantics to each coercion. It is shown that the coercions encoding the coinductive subtype relation for recursive types can be given a canonical operational semantics and so can the two rules presented above also.

$$\mathbf{box} : \tau \leq [\tau] \qquad \mathbf{unbox} : [\tau] \leq \tau$$

where **box** and **unbox** respectively boxes and unboxes their argument. By associating a coercion to every rule of the relation a representation translation is produced automatically. In boxing analysis it is of course very important that such representation translations are as efficient as possible. To this end a coercion calculus is typically developed and an optimality criterion formulated. One then attempts to extract formally optimal coercions from the equivalence classes. Such a scheme is used in [Jør96]. It should be evident that this is exactly what we have been doing in the preceding chapters on coercions and recursive types. We therefore feel that our approach can be extended to cope with a variety of static program analyses in order to include recursive types in the languages being analyzed.

Chapter 11

Conclusion

We have presented new axiomatizations of recursive type equality and subtyping. The axiomatizations are novel inasmuch they are based on a coinductive rule and formulated strictly by syntax, where standard axiomatizations are based on a contraction rule. Our axiomatizations are proven complete and sound with respect to their tree based definitions and hence also with respect to Amadio, Cardelli's and others axiomatizations.

A coercion language has been constructed to encode derivations in the recursive subtype relation. Coercions are presented as a *real* programming language with type system and operational semantics. The coercions are inherently recursive since they have to encode the coinductive rule. The equality relation developed for coercions is therefore not surprisingly also based on coinduction. Coercions are shown coherent with subtype derivations – coercion equality being the connecting concept. An interpretation model is given for coercions and equality is proven sound in this model. Finally, a criteria for optimal coercions is formulated and it is shown that optimal coercions exist for all subtype derivations and an explicit algorithm for determining them is presented.

Our work has several advantages compared to other related works:

- The original motivation for developing new axiomatizations was to avoid the problematic (Contract) rule. This objective is definitely achieved. An immediate advantage of not having a contraction rule is that an operational interpretation of the axiomatization is easy to give, which is evident in the presentation of coercions.

- Another advantage of the coinductive rule is that it is strictly syntax oriented and computational constructive. These facts implies that algorithms for determining the equality or subtype relations are simple and very naturally organized. The naive algorithms presented in this report (algorithm **S** in Figure 5.2.2 and **O** in Figure 9.3) can easily be implemented to achieve $\mathcal{O}(n^2)$ time complexity (n syntactic size of input types).

- Our axiomatizations are more natural and self-contained, *i.e.* no external concepts like contractions or regular trees are needed. We feel that axiomatizations should be as simple as possible such that properties about the relations are easily proven.

- Each rule of our axiomatizations (that is coercions) corresponds exactly to one language transformation in real programming languages. Our coercions are therefore directly applicable in modern programming languages as programs for representation transformation.
- Our work provides a good foundation for static program analyses with recursive types. This is the case because many program analyses consists of the same constructions as presented in this report (type system, coercions, coercion calculus and coherence with type system, formal optimal coercions).

11.1 Future Work

We give a small selection of possible interesting directions for future work.

1. Pursue and prove the ideas of computional adequacy and observational congruence presented in Section 8.3.
2. Extend type system with simple constructions, such as $\times$, $+$ and basic ground types. We expect no problems and only minor changes are required of theorems and proofs.
3. Extend type system with more complex constructions, such as type schemas $\forall\alpha.\tau$, union-, intersection- and object types. We expect some complications, especially concerning polymorphism.
4. Implement efficient ($\mathcal{O}(n^2)$) version of algorithm **O** for computing optimal coercions.
5. Develop type inference system for recursive types based on new subtype axiomatization and algorithm **O**.
6. Apply theory to concrete static program analyses to obtain recursive types in these.
7. Apply idea of contractive rule to other coinductively defined relations, *e.g.* extensional equality of Böhm trees or observational equivalence of programs.

Bibliography

[AbCa96] M. Abadi and L. Cardelli. *A Theory of Objects*, Monographs in Computer Science. Spring Verlag, 1996.

[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] R. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(4):575–631, September 1993.

[AF96] M. Abadi and M.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] Z.M. Ariola and J.W. Klop. Equational term graph rewriting. Technical report, University of Oregon, 1995. To appear in Acta Informatica.

[Ama90] R. Amadio. Typed equivalence, type assignment and type containment. *Proc. CTRS90*, Montreal, 1990.

[ASU86] A.V. Aho, R. Sethi and J.D. Ullman Compilers. Principles, Techniques and Tools. Addison-Wesley, 1986

[Berg92] C. Berg. Lecture Notes in Matematically Analysis, "Metric Spaces". University of Copenhagen, 1992.

[Bra97] M. Brandt. Coinductive Axiomatization of Recursive Type Equality. DIKU student project, January 1997.

[BH97] M. Brandt, F. Henglein. Coinductive Axiomatization of Recursive Type Equality and Subtyping. *Proc. 3d Int'l Conf. on Typed Lambda Calculi and Applications (TLCA), LNCS??.* Springer 1997.

[BrTa89] V. Breazu-Tannen, C. Gunter, A. Scedrov. Denotational Semantics for subtyping between recursive types. Report MS–CIS 89 63, Login of Computation 12, Dept. Comp. Sci. of Pennsylvania.

[CC91] F. Cardone and M. Coppo. Type inference with recursive types: Syntax and semantics. *Information and Computation*, 92(1):48–80, May 1991.

[Cour83] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.

[Dam94] F. Damm. Subtyping with Union Types, Intersection Types and Recursive Types. *Lecture Notes in Computer Science*, 789:687–706, 1994

[Fio94] M.P. Fiore. A Coinduction Principle for Recursive Data Types Based on Bisimulation. *Proc. to 8th LICS Conf. 1993, 110–119* or in *iandcomp Journal 127:186–198, 1996.*

[FM90] Y-C. Fuh, P. Mishra. Type Inference with Subtypes. *Theoretical Computer Science*, 73:155–175, 1990

[Gor95] A.D. Gordon. Functional Programming, Ayr, Scotland 1994. *Workshops in Computing*, Springer, 1995.

[GR96] A.D. Gordon, G.D. Rees. Bisimilarity for a first-order calculus of objects with subtyping. *Proceedings of the Twenty–Third Annual ACM Symposium on Principles of Programming Languages*, 1996.

[Gun92] C.A. Gunter. Semantics of Programming Languages: Structures and Techniques *Foundations of Computing*, MIT press, 1992.

[HS86] R. Hindley, J.P. Seldin. Introduction to Combinators and lambda–calculus *Cambridge Univ. Press*, 1986

[Hoa71] C.A.R. Hoare. Procedures and parameters: an axiomatic approach. *Lecture Notes in Matematics*, 188:102–116, 1971.

[Hoa75] C.A.R. Hoare. Recursive Data Structures. *Int. Journal of Computer and Information Sciences*, Vol. 4, No. 2, 1975.

[Jør96] J. Jørgensen. *A Calculus for Boxing Analysis of Polymorphically Typed Languages.* PhD thesis, DIKU report 96/28, University of Copenhagen, May 1996.

[KPS93] D. Kozen, J. Palsberg, and M. 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] D. Kozen, J. Palsberg, and M. Schwartzbach. Efficient recursive subtyping. *Mathematical Structures in Computer Science (MSCS)*, 5(1), 1995.

[Mil84] R. Milner. A complete inference system for a class of regular behaviours. *Journal of Computer and System Sciences (JCSS)*, 28:439–466, 1984.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989

[Mit91] J.C. Mitchell Type Inference with Simple Subtypes *Journal of Functional Programming*, 1(3):245–285, July 1991

[Nie85] H.R. Nielson. A Hoare–like Proof System for Total Correctness of Nested Recursive Procedures. *Proc. 4th Hung.Computer Science Conf., 1985.*

[Par81] D.M. Park. Concurrency and automata on infinite sequences. *Lecture Notes in Computer Science*, 104:167–183,1981.

[Reh96] J. Rehof. Introduction to Subtyping. Notes, DIKU, 1996

[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.

[Sco72] D. Scott. Continuous Lattices. *Lecture Notes in Matematics*, 274:97–136, 1972

[SeYo94] T. Sekiguchi, A. Yonezawa. A Complete Type Inference System for Subtyped Recursive Types. *Lecture Notes in Computer Science*, 789:667–685, 1994

[Sok77] S. Sokolowski. Total Correctness for Procedures. *Lecture Notes in Computer Science (LNCS), Proc. 6th Symp. Matematical Foundations of Computer Science*, 53:475–483, 1977.

[Win93] W. Glynn. The Formal Semantics of Programming Languages: An Introduction. *MIT Press*, Cambridge, Massachusetts, 1993